ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Containerized Database Indexing Service

Κωνσταντίνος Δ. Γεωργαντόπουλος

Επιβλέπων: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2018

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Containerized Database Indexing Service

Κωνσταντίνος Δ. Γεωργαντόπουλος
Α.Μ.: 1115201300029

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ

ΠΕΡΙΛΗΨΗ

Η παρούσα εργασία αποσκοπεί στην δημιουργία μίας υπηρεσίας ευρετηρίου βάσης δεδομένων η οποία θα φιλοξενείται μέσα σε έναν περιέκτη (container) και θα εξυπηρετεί πολλούς χρήστες ταυτόχρονα. Η εξυπηρέτηση των χρηστών θα γίνεται απομακρυσμένα σύμφωνα με το μοντέλο πελάτη-εξυπηρετητή (server-client).

Αρχικά, θα εισάγουμε την έννοια της εικονικοποίησης και θα εξερευνήσουμε δύο βασικές εκφάνσεις της. Από τη μία θα εξετάσουμε τις εικονικές μηχανές και από την άλλη την τεχνολογία των περιεκτών. Ακόμη, θα κάνουμε μία σύγκριση μεταξύ των δύο.

Στη συνέχεια, θα αναλύσουμε διεξοδικά την υπηρεσία ευρετηρίου που θα χαρακτηρίζει την παρούσα εργασία. Θα δούμε αναλυτικά τις δομές και τις λειτουργίες τους και θα κάνουμε αξιολόγηση της απόδοσης της υπηρεσίας σχετικά με τους χρόνους απόκρισης και το μέγεθος των δεδομένων.

Τέλος, θα εξάγουμε κάποια συμπεράσματα για την υπηρεσία και θα προτείνουμε εναλλακτικούς τρόπους υλοποίησής της.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συστήματα Βάσεων Δεδομένων, Εικονικοποίηση

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: περιέκτης, εικονικοποίηση, ευρετήριο, βάση δεδομένων, υπηρεσία

ABSTRACT

This work aims to create a database indexing service hosted inside a container and serving multiple users at once. The customer service will be remote based on the client-server model.

Initially, we will introduce the concept of virtualization and explore two key aspects of it. On the one hand we will look at virtual machines and on the other hand the technology of the containers. We will also make a comparison between the two.

We will then thoroughly analyze the indexing service that characterizes this work. We will look closely at their structures and operations, and we will evaluate service performance on response times and data size.

Finally, we will draw some conclusions about the service and propose alternative ways of implementing it.

SUBJECT AREA: Database Systems, Virtualization

KEYWORDS: container, virtualization, index, database, service

ΕΥΧΑΡΙΣΤΙΕΣ

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τον επιβλέπων, καθ. Αλέξη Δελή, για την ευκαιρία που μου έδωσε για συνεργασία και την πολύτιμη συμβολή του στην ολοκλήρωση της παρούσας πτυχιακής εργασίας.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ **10**

1. ΕΙΣΑΓΩΓΗ **11**

2. ΕΙΚΟΝΙΚΟΠΟΙΗΣΗ **12**

2.1 Εικονικοποίηση Πλατφόρμας 12

2.2 Εικονικοποίηση Επιπέδου ΛΣ 14

2.2.1 Linux Containers (LXC) 15

2.2.2 Docker 17

2.3 Εικονικές Μηχανές έναντι Containers 21

3. CONTAINERIZED ΥΠΗΡΕΣΙΑ ΕΥΡΕΤΗΡΙΟΥ ΒΔ **23**

3.1 Δομή Ευρετηρίου 24

3.1.1 B^{link}-Δέντρο 24

3.1.2 Υλοποίηση B^{link}-Δέντρου 25

3.2 Πρόγραμμα Εξυπηρετητή 33

3.2.1 Δομικά στοιχεία Εξυπηρετητή 34

3.2.2 Κύκλος ζωής μιας σύνδεσης χρήστη στον εξυπηρετητή 37

3.3 Container Υπηρεσίας 37

3.4 Πρόγραμμα Πελάτη 41

4. ΑΞΙΟΛΟΓΗΣΗ ΥΠΗΡΕΣΙΑΣ **42**

4.1 Δοκιμαστικό σενάριο ερωτημάτων αναζήτησης 42

4.2 Δοκιμαστικό σενάριο μεικτών ερωτημάτων 42

5. ΣΥΜΠΕΡΑΣΜΑΤΑ **44**

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ **45**

ΣΥΝΤΜΗΣΕΙΣ, ΑΡΚΤΙΚΟΛΕΞΑ ΚΑΙ ΑΚΡΩΝΥΜΙΑ **46**

ΑΝΑΦΟΡΕΣ **47**

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

Σχήμα 1: Μοντέλα συστημάτων. a) Μη εικονική μηχανή. b) Εικονική μηχανή. 13

Σχήμα 2: Σύγκριση μεταξύ τεχνικών εικονικοποίησης. Αριστερά: Εικονικές μηχανές. Δεξιά: Containers. 14

Σχήμα 3: Δημιουργία LXC Container. 16

Σχήμα 4: Τρέχουσα κατάσταση ενεργών LXC Container. 16

Σχήμα 5: Άνοιγμα κελύφους σε ενεργό LXC Container. 16

Σχήμα 6: Καταστροφή LXC Container. 17

Σχήμα 7: Αρχιτεκτονική του Docker. 19

Σχήμα 8: Τα επίπεδα του συστήματος αρχείων του Docker. 20

Σχήμα 9: Οι βασικοί άξονες της υπηρεσίας 23

Σχήμα 10: Δομή εγγραφής δεδομένων. 25

Σχήμα 11: Δομή εγγραφής εσωτερικού κόμβου. 26

Σχήμα 12: Δομή κεφαλίδας μπλοκ. 26

Σχήμα 13: Δομή κλειδωνιάς. 26

Σχήμα 14: Ρουτίνα απόκτησης κλειδωνιάς τύπου SHARE. 27

Σχήμα 15: Ρουτίνα άφεσης κλειδωνιάς τύπου SHARE. 27

Σχήμα 16: Αναζήτηση δείκτη σε μπλοκ εσωτερικού κόμβου. 28

Σχήμα 17: Εισαγωγή εγγραφής σε μπλοκ δεδομένων. 29

Σχήμα 18: Διαχωρισμός μπλοκ εσωτερικού κόμβου. 30

Σχήμα 19: Διαχωρισμός μπλοκ δεδομένων. 30

Σχήμα 20: Διαγραφή εγγραφής. 31

Σχήμα 21: Αναζήτηση εγγραφής με τελεστή ίσον ($=$). 31

Σχήμα 22: Αναζήτηση εγγραφής με τελεστή διάφορο ($\neq$). 32

Σχήμα 23: Αναζήτηση εγγραφής με τελεστές μικρότερο ($<$) και μικρότερο-ίσο ($\leq$). . . . 32

Σχήμα 24: Αναζήτηση εγγραφής με τελεστές μεγαλύτερο ($>$) και μεγαλύτερο-ίσο ($\geq$). . . . 33

Σχήμα 25: Συνάρτηση παρακολούθησης δικτύου για αιτήματα σύνδεσης. . . . 35

Σχήμα 26: Δομή job scheduler. . . . 35

Σχήμα 27: Φάση προετοιμασίας. Εδώ γίνεται η διαλογή του πεδίου κλειδιού και του πεδίου φορτίου. . . . 36

Σχήμα 28: Το μοντέλο του εξυπηρετητή. Ο πελάτης επικοινωνεί με την υποδοχή παρακολούθησης, το νήμα συνδέσεων δημιουργεί μία νέα υποδοχή την οποία τοποθετεί στην ουρά αναμονής και τα νήματα εξυπηρέτησης λαμβάνουν αυτές τις υποδοχές από την ουρά για την εξυπηρέτηση του πελάτη. . . . 37

Σχήμα 29: Σχηματική αναπαράσταση της θέσης του container σχετικά με την υπηρεσία και το σύστημα οικοδεσπότη. . . . 38

Σχήμα 30: Dockerfile υπηρεσίας. . . . 39

Σχήμα 31: Αποτέλεσμα κατασκευής εικόνας του container. . . . 40

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Πίνακας 1: Μέσος Χρόνος Απόκρισης Υπηρεσίας (AVR) σε ερωτήματα αναζήτησης 42
Πίνακας 2: Μέσος Χρόνος Απόκρισης Υπηρεσίας (AVR) σε μεικτά φορτία ερωτημάτων 43

ΠΡΟΛΟΓΟΣ

Η παρούσα εργασία με τίτλο Containerized Database Indexing Service εκπονήθηκε στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών. Η εκπόνηση της πτυχιακής εργασίας έγινε στα πλαίσια συνεργασίας με τον καθηγητή κ. Αλέξη Δελή στον τομέα των Βάσεων Δεδομένων και των Λειτουργικών Συστημάτων. Σκοπός της παρούσας εργασίας είναι η δημιουργία μιας υπηρεσίας ευρετηρίου βάσης δεδομένων η οποία θα είναι ενσωματωμένη σε μια ανερχόμενη και ελαφρύτερη τεχνολογία εικονικοποίησης, το container.

1. ΕΙΣΑΓΩΓΗ

Η παρούσα εργασία αφορά στην δημιουργία μιας υπηρεσίας ευρετηρίου βάσης δεδομένων και στην μελέτη μιας ανερχόμενης τεχνολογίας εικονικοποίησης λειτουργικών συστημάτων, των containers. Η μελέτη αυτή της τεχνολογίας θα γίνει τόσο σε θεωρητικό επίπεδο με παρουσίαση βασικών χαρακτηριστικών και λειτουργιών, όσο και σε πρακτικό επίπεδο με τη φιλοξενία της υπηρεσίας ευρετηρίου σε ένα container.

Αρχικά, θα εξετάσουμε την τεχνολογία των containers. Πρόκειται για αυτοδύναμα περιβάλλοντα εκτέλεσης με δικά τους μερίδια υπολογιστικών πόρων που χρησιμοποιούν τον πυρήνα του φιλοξενούντος λειτουργικού συστήματος. Λειτουργούν ως εικονικές μηχανές αλλά πολύ ελαφρύτερα σε θέμα κατανάλωσης πόρων και σε θέμα κλιμάκωσης καθώς χρησιμοποιούν ένα υποσύνολο του ΛΣ και του συστήματος γενικότερα. Πειράματα δείχνουν ότι το πλήθος των containers που μπορεί να φιλοξενήσει ένα σύστημα εκτείνεται από 6 έως 8 φορές μεγαλύτερο από αυτό των εικονικών μηχανών [6]! Οι υλοποιήσεις αυτής της τεχνολογίας είναι σχετικά πρόσφατες με το πρώτο ολοκληρωμένο περιβάλλον να είναι το LXC και το πιο δημοφιλές να είναι το μεταγενέστερο, και κατά κάποιο τρόπο απόγονο του LXC, το Docker.

Στη συνέχεια, θα παρουσιάσουμε την υπηρεσία ευρετηρίου βήμα-βήμα. Η υπηρεσία αυτή θα χρησιμοποιεί μία παραλληλοποιημένη μορφή B^{+}-δέντρου, το B^{link}-δέντρο, η οποία εξυπηρετεί ταυτόχρονα πολλές αιτήσεις, θα έχει μία δαίμονα διεργασία (daemon process) η οποία θα δέχεται αιτήματα από πολλές client διεργασίες και θα απαντάει σε κάθε μία με τις εγγραφές της βάσης που αντιστοιχούν στο εκάστοτε ερώτημα. Η δαίμονας διεργασία θα φιλοξενείται σε ένα container, ενώ οι client διεργασίες μπορεί να εκτελούνται σε οποιοδήποτε σύστημα από τον χρήστη.

Τέλος, θα εξάγουμε κάποια συμπεράσματα και θα αναφέρουμε μελλοντικές διαφοροποιήσεις της υπηρεσίας όπως διαφορετική δομή ευρετηρίου από το B^{link}-Δέντρο.

2. ΕΙΚΟΝΙΚΟΠΟΙΗΣΗ

Στην πληροφορική, ο όρος εικονικοποίηση αναφέρεται στην δημιουργία μιας εικονικής έκδοσης ενός αντικειμένου, είτε αυτό λέγεται υλικό υπολογιστή, είτε αποθηκευτικός πόρος, είτε δίκτυο υπολογιστών. Στην πιο συχνή της μορφή αναφέρεται στην εκτέλεση πολλών λειτουργικών συστημάτων στο ίδιο υπολογιστικό σύστημα ταυτόχρονα. Οι εφαρμογές που εκτελούνται επάνω από την εικονικοποιημένη μηχανή θεωρούν ότι βρίσκονται σε μια αποκλειστικά δικιά τους μηχανή, όπου το λειτουργικό σύστημα, οι βιβλιοθήκες και τα άλλα προγράμματα είναι μοναδικά στο φιλοξενούμενο σύστημα και ασύνδετα με το σύστημα οικοδεσπότη που βρίσκεται από κάτω [8].

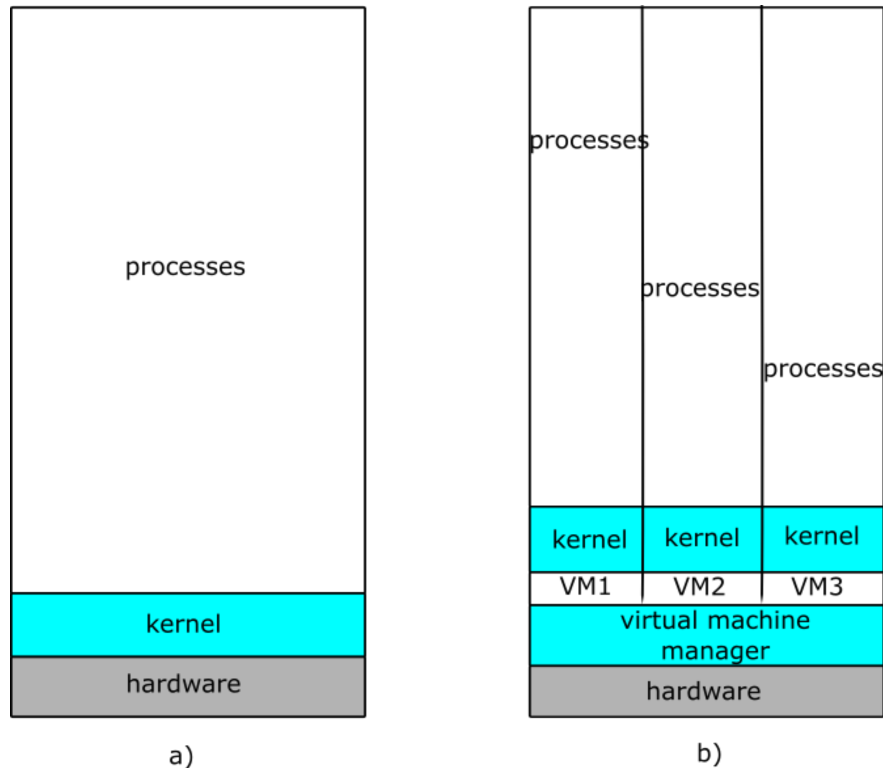
Υπάρχουν πολλοί λόγοι για την χρήση της εικονικοποίησης στους υπολογιστές. Για τους απλούς καθημερινούς χρήστες η πιο κοινή χρήση είναι η δυνατότητα να μπορούν να εκτελούν εφαρμογές που έχουν γραφτεί για διαφορετικά συστήματα χωρίς να χρειάζεται να αλλάξουν μηχανή ή να επανεκκινήσουν την δικιά τους με άλλο σύστημα. Για διαχειριστές εξυπηρετητών η εικονικοποίηση προσφέρει την δυνατότητα να εκτελούνται διαφορετικά λειτουργικά συστήματα, αλλά ίσως πιο σημαντικό είναι ότι προσφέρει έναν τρόπο να κατακερματιστεί ένα σύστημα σε μικρότερα κομμάτια, επιτρέποντας στον εξυπηρετητή να χρησιμοποιείται πιο αποδοτικά από έναν αριθμό διαφορετικών χρηστών ή από εφαρμογές με διαφορετικές ανάγκες. Επίσης, επιτρέπει την απομόνωση, κρατώντας τα προγράμματα που εκτελούνται σε μια εικονική μηχανή ασφαλή από διεργασίες που εκτελούνται σε άλλες εικονικές μηχανές στο ίδιο σύστημα οικοδεσπότη [2].

2.1 Εικονικοποίηση Πλατφόρμας

Ίσως η πιο γνωστή μορφή εικονικοποίησης είναι η εικονικοποίηση πλατφόρμας. Η εικονικοποίηση πλατφόρμας αναφέρεται στην δημιουργία μιας εικονικής μηχανής η οποία συμπεριφέρεται σαν κανονικός υπολογιστής με το δικό του λειτουργικό σύστημα που όμως φιλοξενείται σε έναν υπολογιστή οικοδεσπότη (host). Η θεμελιώδης ιδέα πίσω από μια εικονική μηχανή είναι να μεταφέρεται μια προσομοίωση του υλικού ενός υπολογιστή σε αρκετά και διαφορετικά περιβάλλοντα εκτέλεσης, δημιουργώντας τη ψευδαίσθηση ότι κάθε ξεχωριστό περιβάλλον εκτελείται στο δικό του ιδιωτικό υπολογιστή. Το λογισμικό που εκτελείται σε μια εικονική μηχανή βρίσκεται σε ένα τέτοιο περιβάλλον εκτέλεσης το οποίο φαίνεται σε αυτό ότι είναι το εγγενές υλικό της φυσικής μηχανής, ενώ στην ουσία το απομονώνει από αυτό [2]. Το περιβάλλον αυτό συμπεριφέρεται στο λογισμικό όπως θα του συμπεριφερόταν το εγγενές υλικό, αλλά επίσης το προστατεύει, το διαχειρίζεται και το περιορίζει.

Οι υλοποιήσεις εικονικών μηχανών περιλαμβάνουν αρκετά συστατικά. Στην βάση βρίσκεται το φιλόξενο σύστημα (host), το υποκείμενο σύστημα υλικού, που εκτελεί τις εικονικές μηχανές. Ο διαχειριστής εικονικής μηχανής (virtual machine manager, VMM), που είναι επίσης γνωστός ως hypervisor, δημιουργεί και εκτελεί τις εικονικές μηχανές, παρέχοντας μια διεπαφή, η οποία είναι πανομοιότυπη με το φιλόξενο σύστημα [2]. Κάθε διεργασία επισκέπτης (guest) παρέχεται μαζί μ’ ένα εικονικό αντίγραφο του φιλόξενου συστήματος.

Συνήθως, η διεργασία επισκέπτης είναι στην ουσία ένα λειτουργικό σύστημα. Μια φυσική μηχανή μπορεί λοιπόν να εκτελεί πολλαπλά λειτουργικά συστήματα ταυτόχρονα, όπου το καθένα βρίσκεται μέσα στη δική του εικονική μηχανή.



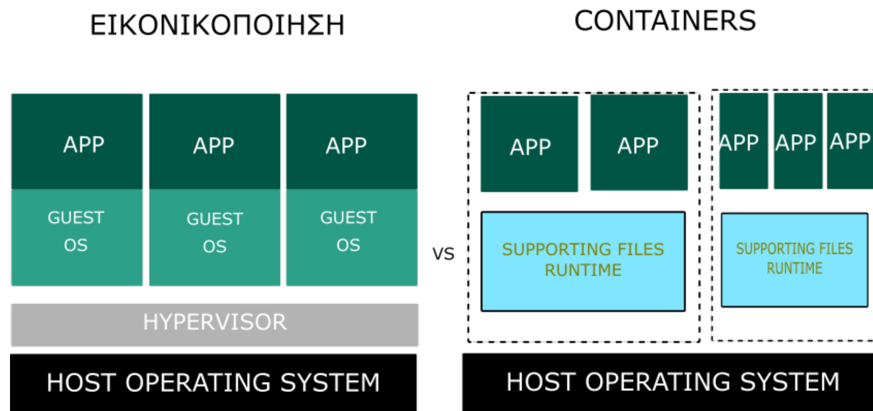
Σχήμα 1: Μοντέλα συστημάτων. a) Μη εικονική μηχανή. b) Εικονική μηχανή.

Ως ένα πρακτικό παράδειγμα χρήσης της εικονικοποίησης πλατφόρμας ας σκεφτούμε μια εταιρεία η οποία διαθέτει τους εξής πόρους: μία ιστοσελίδα για την προώθηση προϊόντων και την ενημέρωση των πελατών, ηλεκτρονικό ταχυδρομείο για την επικοινωνία της και κάποιες εφαρμογές για την εσωτερική λειτουργία της. Έστω ότι για το καθένα από τα παραπάνω διαθέτει μία φυσική μηχανή, δηλαδή έναν web server, έναν mail server και έναν υπολογιστή για τις εφαρμογές της. Κάθε μηχανή από αυτές τρέχει περίπου στο 30% των δυνατοτήτων της, ένα πολύ μικρό ποσοστό φυσικά. Εφόσον όμως οι εφαρμογές της εταιρείας είναι σημαντικές για την λειτουργία της δεν θα πρέπει να τις κρατήσουμε μαζί με την φυσική μηχανή τους; Παραδοσιακά η απάντηση θα ήταν ναι γιατί είναι συχνά πιο εύκολο και αξιόπιστο να τρέχουμε ξεχωριστές δραστηριότητες σε ξεχωριστές μηχανές. Όμως με την εικονικοποίηση και τις εικονικές μηχανές μπορούμε σε μία μηχανή να λειτουργήσουμε π.χ. τόσο τον mail server όσο και τις εσωτερικές εφαρμογές χρησιμοποιώντας έτσι πιο αποδοτικά την συγκεκριμένη μηχανή. Η τεχνολογία της εικονικοποίησης μας προσφέρει τόσο την ασφάλεια με το γεγονός ότι κάθε κομμάτι λειτουργεί ανεξάρτητα και απομονωμένα από το άλλο, όσο και την απόδοση αφού κάθε κομμάτι παίρνει το ίδιο ποσοστό των πόρων της μηχανής που είχε και πριν. Μετά από αυτή την αλλαγή οι μηχανές που δεν χρησιμοποιούνται πια μπορούν είτε να χρησιμοποιηθούν για κάποια άλλη εργασία είτε να παροπλιστούν μειώνοντας έτσι και τα κόστη λειτουργίας και συντήρησης [11].

2.2 Εικονικοποίηση Επιπέδου ΛΣ

Η εικονικοποίηση επιπέδου ΛΣ, γνωστή και ως εγκλεισμός εφαρμογής, αναφέρεται σε ένα χαρακτηριστικό των ΛΣ στο οποίο ο πυρήνας επιτρέπει την ύπαρξη πολλών απομονωμένων στιγμιότυπων χώρων-χρήστη (user spaces) ταυτόχρονα [7]. Αυτά τα στιγμιότυπα, που ονομάζονται containers, μπορεί να φαίνονται ως αληθινοί αυτόνομοι υπολογιστές στα προγράμματα που εκτελούνται μέσα σε αυτά. Ουσιαστικά, ένα container είναι ένα σύνολο από διεργασίες απομονωμένες από το υπόλοιπο σύστημα και εκτελούμενες από μία διαφορετική εικόνα λογισμικού που παρέχει όλα τα αρχεία που είναι απαραίτητα για την υποστήριξη αυτών των διεργασιών. Με την παροχή μίας εικόνας λογισμικού που περιέχει όλες τις εξαρτήσεις μιας εφαρμογής, αυτή μετατρέπεται σε φορητή και συνεπή καθώς μεταφέρεται από την ανάπτυξη, στον έλεγχο και τέλος στην παραγωγή της.

Είναι, όμως, τα container πλήρης εικονικοποίηση; Η απάντηση σε αυτό το ερώτημα είναι διττή: Ναι, γιατί επιτρέπει την παράλληλη εκτέλεση πολλών συνόλων διεργασιών απομονωμένων μεταξύ τους - Όχι, γιατί χρησιμοποιεί τον ίδιο πυρήνα ΛΣ για αυτή τη δουλειά και όχι διαφορετικά ΛΣ όπως η πλήρης εικονικοποίηση. Το γεγονός αυτό καθιστά τα containers πιο ελαφριά πρακτική [12] [13]. Σε αντίθεση με τις εικονικές μηχανές δεν χρειάζονται ούτε ένα επιπλέον επίπεδο hypervisor για να τρέξουν, ούτε την ανάπτυξη πολλών ΛΣ, αλλά χρησιμοποιούν την ίδια τη διεπαφή του ΛΣ για τη λειτουργία τους. Αυτό μειώνει το φορτίο γύρω από τα containers και επιτρέπει μια μεγαλύτερη πυκνότητα από αυτά να εκτελούνται σε έναν οικοδεσπότη.



Σχήμα 2: Σύγκριση μεταξύ τεχνικών εικονικοποίησης. Αριστερά: Εικονικές μηχανές. Δεξιά: Containers.

Μέσα σ' ένα σύστημα εικονικοποίησης ΛΣ είναι εγκατεστημένος μόνο ένας πυρήνας και το υλικό δεν είναι εικονικοποιημένο. Αντίθετα, το ΛΣ και οι συσκευές του είναι εικονικοποιημένες, δίνοντας στις διεργασίες μέσα σε έναν περιέκτη την εντύπωση ότι είναι οι μόνες διεργασίες μέσα στο σύστημα. Ένας ή περισσότεροι περιέκτες μπορούν να δημιουργηθούν σε ένα σύστημα και ο καθένας μπορεί να έχει τις δικές του εφαρμογές, στοίβες δικτύου, διεύθυνση δικτύου και θύρες, λογαριασμούς χρηστών, συσκευές, υποσύνολο πόρων κλπ. Κάθε περιέκτης μπορεί να περιλαμβάνει αρκετά προγράμματα τα οποία μπορούν να εκτελούνται ταυτόχρονα, ξεχωριστά ή να αλληλεπιδρούν μεταξύ τους. Ακόμη, κάποιες φορές

ένας περιέκτης μπορεί να διατηρεί δικό του χρονοπρογραμματιστή για να βελτιστοποιεί την απόδοση των εφαρμογών στους πόρους που του έχουν δοθεί.

Η ιδιότητα των containers να λειτουργούν κατά κάποιο τρόπο ως φιλοξενούμενοι του ΛΣ έχει εγείρει κάποιες αμφιβολίες για την αξιοπιστία τους στην επιστημονική κοινότητα[4]. Πρώτον, θεωρούνται ως μη ευέλικτα επειδή εκμεταλλεύονται χαρακτηριστικά του πυρήνα πάνω στον οποίο αναπτύσσονται και έτσι πρέπει να έχουν ως βάση ίδια ή παρεμφερή έκδοση ΛΣ με αυτή του οικοδεσπότη. Δεύτερον, θεωρούνται λιγότερο ασφαλή σε σχέση με την πλήρη απομόνωση που προσφέρει το επίπεδο του hypervisor. Ως αντεπιχείρημα σε αυτό στέκει το ότι η ελαφρύτερη τεχνική των containers προσφέρει λιγότερη «επιφάνεια» επίθεσης από ότι το πλήρες ΛΣ που απαιτείται από μια εικονική μηχανή μαζί με τις παθογένειες που μπορεί να κρύβει και το ίδιο το επίπεδο του hypervisor.

Η πιο βασική χρήση της τεχνικής αυτής είναι σε περιβάλλοντα virtual hosting όπου είναι χρήσιμη για την ανάθεση ενός συγκεκριμένου αριθμού πόρων ανάμεσα σε μεγάλο πλήθος από αμοιβαία δύσπιστους χρήστες. Επίσης, τυπική χρήση της τεχνικής αυτής είναι ο εγκλεισμός διαφορετικών εφαρμογών σε διαφορετικούς περιέκτες μέσα σε έναν server π.χ. που προσφέρει αυξημένη ασφάλεια, ανεξαρτησία πόρων και προχωρημένες τεχνικές διαχείρισής τους. Ακόμη, η τεχνική αυτή μπορεί να χρησιμοποιηθεί και ως ένα δοκιμαστικό περιβάλλον για εφαρμογές υπό ανάπτυξη το οποίο τις απομονώνει από το υπόλοιπο σύστημα και το προστατεύει από τυχόν αστοχίες ή κενά ασφαλείας των εφαρμογών αυτών. Τέλος, προσφέρει εύκολη μεταφορά και ανάπτυξη εφαρμογών από το ένα σύστημα στο άλλο χωρίς περιττές ρυθμίσεις και ενέργειες.

Παρακάτω, θα ασχοληθούμε με περιέκτες των εκδόσεων Linux και θα δούμε δύο υλοποιήσεις της εικονικοποίησης επιπέδου ΛΣ για αυτούς: το LXC και το Docker.

2.2.1 Linux Containers (LXC)

Το LXC είναι μία υλοποίηση της εικονικοποίησης επιπέδου ΛΣ για την εκτέλεση πολλαπλών απομονωμένων συστημάτων Linux (containers) σε έναν υπολογιστή χρησιμοποιώντας έναν πυρήνα Linux. Ο πυρήνας αυτός προσφέρει τόσο την λειτουργία των cgroups η οποία επιτρέπει την οριοθέτηση και την απονομή προτεραιότητας των πόρων χωρίς την ανάγκη για δημιουργία μιας εικονικής μηχανής, όσο και τη λειτουργία της απομόνωσης χώρου ονομάτων (namespace isolation) που επιτρέπει την πλήρη απομόνωση των εφαρμογών από το ΛΣ. Το LXC δημιουργήθηκε το 2008 και ήταν η πρώτη ολοκληρωμένη υλοποίηση ενός Linux container διαχειριστή [9].

Τα cgroups είναι ένα χαρακτηριστικό του πυρήνα Linux που οριοθετεί, απομονώνει και μεριμνεί για την χρήση των πόρων (ΚΜΕ, μνήμη Ε/Ε δίσκου, δίκτυο κλπ.) από μια συλλογή διεργασιών. Ένα σύνολο ελέγχου (control group - cgroup) είναι μία συλλογή από διεργασίες που δεσμεύονται από ίδια κριτήρια και σχετίζονται με ένα σύνολο από παραμέτρους και όρια. Τα σύνολα αυτά μπορεί να είναι ιεραρχικά, δηλαδή μπορεί κάθε σύνολο να κληρονομεί όρια από το σύνολο-«πατέρα» του. Ο πυρήνας παρέχει πρόσβαση σε πολλαπλούς

ελεγκτές μέσω της διεπαφής των cgroups [6][9].

Η απομόνωση χώρου ονομάτων είναι ένα χαρακτηριστικό του πυρήνα Linux όπου σύνολα από διεργασίες διαχωρίζονται έτσι ώστε να μη μπορούν να «δουν» πόρους από άλλα σύνολα. Για παράδειγμα ο χώρος ονομάτων PID παρέχει μια διαφορετική απαρίθμηση από αναγνωριστικά διεργασιών μέσα σε κάθε διαφορετικό χώρο ονομάτων. Άλλοι γνωστοί χώροι ονομάτων είναι οι mount, UTS, network και IPC [9][10].

Εφόσον έχει εγκατασταθεί η βιβλιοθήκη αυτή στον υπολογιστή, χρησιμοποιώντας την γραμμή εντολών, η δημιουργία ενός container είναι εύκολη με την εντολή: lxc-create -t download -n my-container. Η παράμετρος download μας δείχνει μια λίστα από εκδόσεις Linux για να επιλέξουμε. Μια συνηθισμένη επιλογή είναι η ”ubuntu”.

```
kostas@kostas-VirtualBox:~$ sudo lxc-create -t download -n my-container
Setting up the GPG keyring
Downloading the image index
```

```
Distribution: ubuntu
Release: xenial
Architecture: amd64

Downloading the image index
Downloading the rootfs
Downloading the metadata
The image cache is now ready
Unpacking the rootfs

---
You just created an Ubuntu xenial amd64 (20180918_07:43) container.

To enable SSH, run: apt install openssh-server
No default root or user password are set by LXC.
```

Σχήμα 3: Δημιουργία LXC Container.

Εφόσον λοιπόν έχει δημιουργηθεί το container μπορούμε να το εκκινήσουμε με την εντολή: lxc-start -n my-container -d

Έπειτα, μπορούμε να δούμε την τρέχουσα κατάσταση του container με οποιαδήποτε από τις δύο εντολές: lxc-info -n my-container ή lxc-ls -f

```
kostas@kostas-VirtualBox:~$ sudo lxc-ls -f
NAME         STATE   AUTOSTART GROUPS IPV4       IPV6
my-container RUNNING 0         -      10.0.3.129 -
```

Σχήμα 4: Τρέχουσα κατάσταση ενεργών LXC Container.

Και να ανοίξουμε ένα κέλυφος μέσα στο container, π.χ. bash, με την εντολή: lxc-attach -n my-container

```
kostas@kostas-VirtualBox:~$ sudo lxc-attach -n my-container
root@my-container:/#
```

Σχήμα 5: Άνοιγμα κελύφους σε ενεργό LXC Container.

Μπορούμε να το σταματήσουμε με την εντολή: lxc-stop -n my-container

Και τέλος να το διαγράψουμε με την εντολή: lxc-destroy -n my-container

```
kostas@kostas-VirtualBox:~$ sudo lxc-destroy -n my-container
Destroyed container my-container
```

Σχήμα 6: Καταστροφή LXC Container.

2.2.2 Docker

Το Docker είναι ένα έργο ανοιχτού-κώδικα (open-source project) του 2013 που ως στόχο έχει την αυτοματοποίηση της τοποθέτησης εφαρμογών σε containers. Το Docker προσθέτει μια μηχανή εγκλεισμού εφαρμογών επάνω από εικονικοποιημένο περιβάλλον εκτέλεσης container. Έχει σχεδιαστεί τόσο για να παρέχει ένα ελαφρύ και γρήγορο περιβάλλον για την εκτέλεση του κώδικα μέσα σε αυτό όσο και μια αποδοτική μέθοδο για τη μεταφορά αυτού του κώδικα από το ένα σύστημα στο άλλο. Είναι εξαιρετικά απλό και το μόνο που χρειάζεται δεν είναι τίποτα παραπάνω από έναν συμβατό πυρήνα Linux και το εκτελέσιμο του.

Το Docker, το οποίο ξεκίνησε ως έργο για την κατασκευή LXC containers μονών εφαρμογών, παρουσίασε αρκετές σημαντικές αλλαγές στο LXC που κάνουν τα containers πιο μεταφέρσιμα και ευέλικτα στη χρήση. Χρησιμοποιώντας το Docker, μπορεί να αναπτυχθεί, κλωνοποιηθεί, μεταφερθεί και αποθηκευτεί ένα φορτίο εργασίας ακόμη πιο εύκολα και γρήγορα από ότι με μια εικονική μηχανή. Βασικά, το Docker, φέρνει μία ευελιξία τύπου cloud σε οποιαδήποτε υποδομή ικανή να εκτελέσει containers.

Θεμελιωδώς, τόσο τα Docker όσο και τα LXC containers είναι ελαφριοί μηχανισμοί εικονικοποίησης χώρων-χρήστη που χρησιμοποιούν την λειτουργία των cgroups και των χώρων ονομάτων για να διαχειριστούν την απομόνωση των πόρων. Υπάρχουν, όμως, κάποιες σημαντικές διαφορές ανάμεσά τους. Τι είναι αυτό που κάνει το Docker πιο εξελιγμένο από τον προκάτοχό του; Ας δούμε πιο συγκεκριμένα:

- **Μονοδιεργασιακό vs Πολυδιεργασιακό.** Το Docker περιορίζει τα containers να εκτελούνται ως μία διεργασία. Αυτό σημαίνει ότι αν μία εφαρμογή αποτελείται από X ταυτόχρονες διεργασίες, το Docker θα τις εκτελέσει σε X containers, το καθένα με μία ξεχωριστή διεργασία. Αντίθετα, τα LXC containers περιλαμβάνουν την παραδοσιακή διεργασία init και μπορούν να τρέξουν πολλαπλές διεργασίες. Τα πλεονεκτήματα των μονοδιεργασιακών containers είναι πολλά, συμπεριλαμβανόμενων των εύκολων και πιο λεπτομερών ενημερώσεων. Υπάρχουν όμως και περιορισμοί σε αυτά. Για παράδειγμα, δεν μπορούν να εκτελεστούν agents, logging scripts ή SSH daemons σε ένα τέτοιο container. Επίσης, δεν είναι εύκολο να υποβάλεις μικρές αλλαγές σε μια εφαρμογή χωρίς την ανάγκη για εκκίνηση ενός νέου ενημερωμένου container.

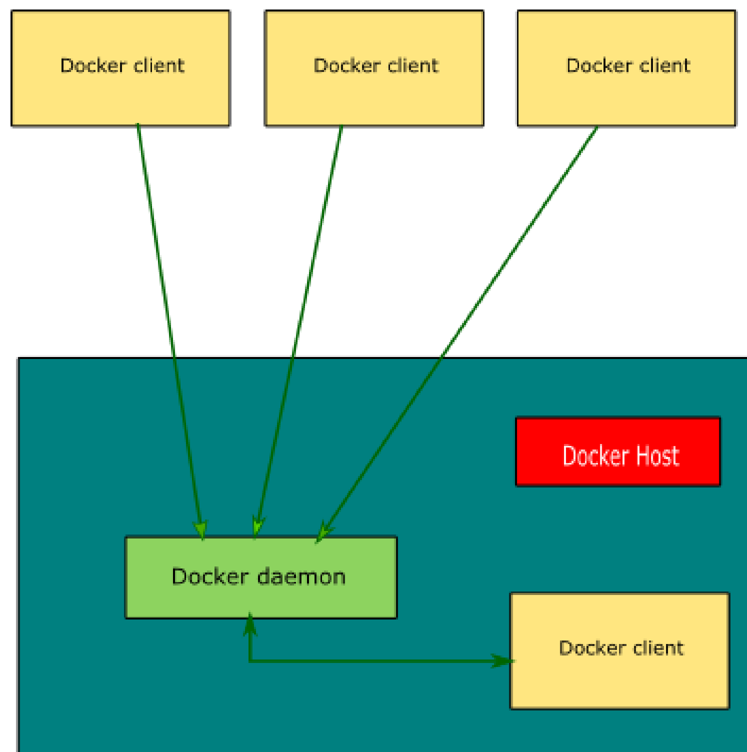
- **Ακαταστατικό vs Καταστατικό**. Τα Docker containers έχουν σχεδιαστεί να είναι ακαταστατικά. Πρώτον, το Docker δεν υποστηρίζει μόνιμη αποθήκευση. Για να ξεπεράσει αυτό το εμπόδιο το Docker επιτρέπει στον χρήστη να συνδέσει αποθηκευτικό χώρο από το σύστημα οικοδεσπότη ως Docker volume στο container. Ακριβώς επειδή αυτός ο χώρος είναι συνδεδεμένος δεν είναι ακριβώς μέρος του container.

Δεύτερον, τα containers του Docker αποτελούνται από επίπεδα που είναι μόνο για ανάγνωση. Αυτό σημαίνει ότι, από τη στιγμή που ένα container έχει σχεδιαστεί, δεν μπορεί να αλλάξει μορφή. Αν κατά την εκτέλεση η διεργασία μέσα στο container κάνει κάποιες εσωτερικές αλλαγές τότε δημιουργείται ένα νέο σχέδιο container διαφορετικό από το αρχικό. Ένα ακαταστατικό container είναι αρκετά ενδιαφέρον για αυτόν ακριβώς τον λόγο: όσες ενημερώσεις και να πραγματοποιηθούν, άλλα τόσα σχέδια containers θα δημιουργηθούν κάνοντας εύκολη την επαναφορά του συστήματος.

- **Φορητότητα**. Αυτό είναι ίσως και το πιο σημαντικό χαρακτηριστικό που ξεχωρίζει το Docker από το LXC. Το Docker προσθέτει μεγαλύτερο επίπεδο αφαίρεσης σε δικτυακές, αποθηκευτικές και λειτουργικές λεπτομέρειες σε μια εφαρμογή απ' ότι το LXC. Με το Docker, μια εφαρμογή είναι πραγματικά ανεξάρτητη από τις ρυθμίσεις αυτών των χαμηλού επιπέδου πόρων. Όταν ένα Docker container μεταφέρεται από έναν οικοδεσπότη με Docker σε μια άλλη μηχανή με υποστήριξη Docker, το Docker εξασφαλίζει ότι τι περιβάλλον της εφαρμογής θα παραμείνει το ίδιο. Ένα άμεσο πλεονέκτημα αυτής της προσέγγισης είναι ότι επιτρέπει στους προγραμματιστές να δημιουργούν τοπικά περιβάλλοντα ανάπτυξης ίδια με αυτά ενός server παραγωγής. Ακόμη και με το LXC, ένας προγραμματιστής μπορεί να πάρει κάτι που εκτελείται στο τοπικό του περιβάλλον και να διαπιστώσει ότι δεν εκτελείται σωστά στο περιβάλλον του server καθώς αυτό είναι διαφορετικό και να χαλάσει ώρες για να διορθώσει το πρόβλημα. Το Docker αφαίρεσε αυτή την πολυπλοκότητα. Αυτό κάνει τα Docker containers τόσο φορητά και εύκολα στη χρήση μεταξύ διαφορετικών συστημάτων [6].

Παραπάνω αναλύθηκε τόσο η γενική ιδέα του Docker όσο και οι διαφορές του με τον προκάτοχό του, LXC. Πώς λειτουργεί όμως το Docker; Πώς υλοποιούνται τα παραπάνω πλεονεκτήματα του; Το Docker, λοιπόν, σαν υπηρεσία αποτελείται από τρία βασικά στοιχεία: τις διεργασίες εξυπηρετητή και πελάτη, τις εικόνες και, φυσικά, τα ίδια τα containers.

Αρχικά, το Docker είναι μία εφαρμογή πελάτη-εξυπηρετητή. Η διεργασία πελάτης επικοινωνεί με την διεργασία εξυπηρετητή ή δαίμονα η οποία κάνει όλη τη δουλειά: εκκίνηση, δημιουργία, καταστροφή, ενημέρωση κατάστασης των containers κ.α. Η πλατφόρμα του Docker έρχεται μαζί με δικό της εκτελέσιμο γραμμής εντολών (CLI binary) και δικό της RESTful API. Οι διεργασίες εξυπηρετητή και πελάτη μπορούν να βρίσκονται τόσο στο ίδιο μηχάνημα όσο και να συνδέονται απομακρυσμένα.



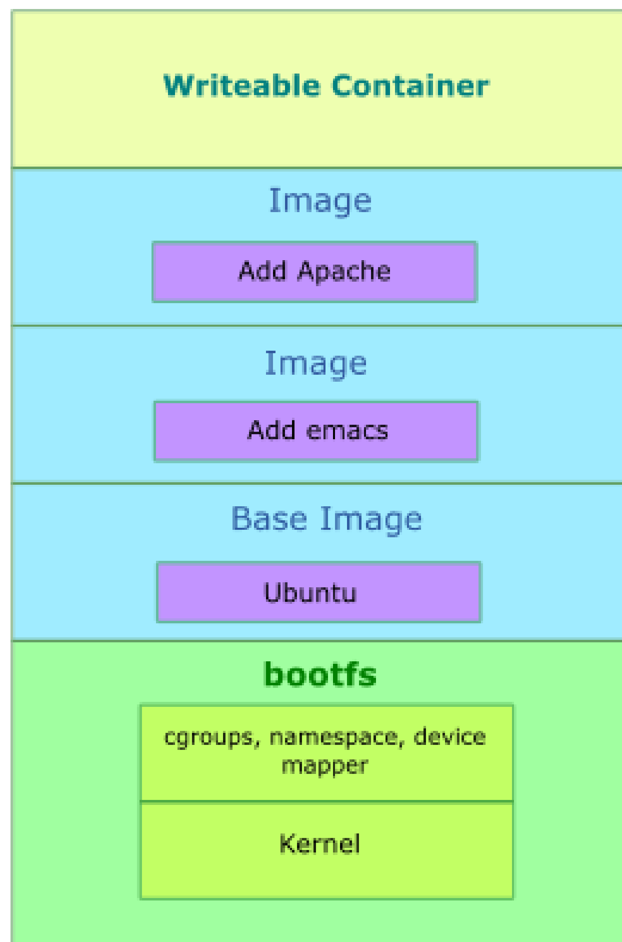
Σχήμα 7: Αρχιτεκτονική του Docker.

Έπειτα, έχουμε τις εικόνες (Docker images). Οι εικόνες είναι ουσιαστικά οι θεμέλιοι λίθοι του κόσμου του Docker. Η εκκίνηση των containers γίνεται από αυτές. Οι εικόνες είναι το κατασκευαστικό κομμάτι του κύκλου ζωής του Docker και τα σχέδια που αναφέραμε ότι δεν αλλάζουν στην ιδιότητα της ακαταστατικότητας. Είναι ένας σχηματισμός από επίπεδα που κατασκευάζονται βήμα-βήμα χρησιμοποιώντας μια σειρά από εντολές όπως:

1. Πρόσθεσε ένα αρχείο.
2. Εκτέλεσε μια εντολή.
3. Άνοιξε μία θύρα.

Κάθε επίπεδο του σχηματισμού αυτού αποτελείται από ενοποιημένα συστήματα αρχείων (file systems) το ένα πάνω από το άλλο [4]. Κάθε τέτοιο σύστημα αποτελεί μία εικόνα. Στην βάση βρίσκεται ένα σύστημα αρχείων boot, το bootffs, το οποίο είναι υπεύθυνο για την εκκίνηση του container όπως ένα τυπικό Linux/Unix boot file system. Στο επόμενο επίπεδο, τοποθετείται ένα σύστημα αρχείων root, το rootffs, το οποίο μπορεί να είναι ένα ή περισσότερα ΛΣ (π.χ. Ubuntu ή Debian file systems). Σε μια παραδοσιακή εκκίνηση του Linux, το root file system φορτώνεται ως μόνο για ανάγνωση και περνάει σε κατάσταση ανάγνωσης-εγγραφής μετά την εκκίνηση. Στον κόσμο του Docker όμως παραμένει σε κατάσταση μόνο ανάγνωσης και το Docker εκμεταλλεύεται την τεχνική ενωτικής φόρτωσης (union-mount) για να προσθέσει περισσότερα read-only file systems επάνω από το root. Η τεχνική της ενωτικής φόρτωσης είναι μία τεχνική που επιτρέπει σε αρκετά συστήματα αρχείων να είναι φορτωμένα την ίδια στιγμή αλλά να φαίνονται ως ένα. Τελικά, όταν ένα container εκκινείται φορτώνεται ένα σύστημα-αρχείων ανάγνωσης εγγραφής επάνω από όλα τα επίπεδα. Εκεί

είναι που εκτελούνται οι εφαρμογές του χρήστη. Το επίπεδο αυτό αρχικά είναι άδειο. όσο γίνονται αλλαγές αποθηκεύονται σε αυτό το επίπεδο. Για παράδειγμα, αν γίνει αλλαγή σε ένα αρχείο, το αρχείο αυτό αντιγράφεται από το κατώτερο, μόνο για ανάγνωση, επίπεδο στο επίπεδο ανάγνωσης-εγγραφής. Η μόνο για ανάγνωση έκδοση του αρχείου υπάρχει, αλλά πλέον έχει κρυφτεί «κάτω» από το αντίγραφο. Αυτή η τεχνική ονομάζεται αντιγραφή κατά την εγγραφή (copy on write) και είναι ένα από τα πιο σημαντικά χαρακτηριστικά του Docker καθώς επιτρέπει σε όλα τα κατώτερα, μόνο για ανάγνωση, επίπεδα να μην αλλάζουν μορφή βοηθώντας έτσι στην ταχύτερη εκκίνηση των containers από τις ίδιες εικόνες σε κάθε σύστημα.



Σχήμα 8: Τα επίπεδα του συστήματος αρχείων του Docker.

Τέλος, έχουμε, φυσικά, τα containers. Όπως είδαμε προηγουμένως τα containers δημιουργούνται από τις εικόνες και μπορούν να περιέχουν μία ή περισσότερες εκτελούμενες διεργασίες. Πρακτικά οι εικόνες είναι η κατασκευαστική πλευρά του Docker και τα containers η εκτελεστική του πλευρά. Ένα container είναι:

1. Ένα σχέδιο εικόνων.
2. Ένα σύνολο από βασικές λειτουργίες.
3. Ένα περιβάλλον εκτέλεσης.

Το Docker δανείζεται την ιδέα του κλασσικού εμπορευματοκιβωτίου, που χρησιμοποιείται

για την μεταφορά αγαθών παγκοσμίως, ως το μοντέλο για τα containers του. Αλλά, αντί για μεταφορά αγαθών, τα containers του Docker μεταφέρουν λογισμικό. Κάθε container περιέχει μία εικόνα λογισμικού -το φορτίο του- και, όπως ο φυσικός ομόλογός του, επιτρέπει ένα σύνολο από λειτουργίες να λάβει χώρα. Για παράδειγμα, μπορεί να δημιουργηθεί, να καταστραφεί, να εκκινηθεί, να σταματήσει και να επανεκκινηθεί. Όπως και ένα εμπορευματοκιβώτιο έτσι και το Docker δεν ενδιαφέρεται για τα περιεχόμενα ενός container όταν πραγματοποιεί τις παραπάνω λειτουργίες. Κάθε container φορτώνεται ακριβώς όπως κάθε άλλο. Επίσης το Docker δεν ενδιαφέρεται για το που μεταφέρεται ένα container. Μπορεί να δημιουργηθεί σε ένα laptop, να μεταφερθεί σε ένα server, εικονικό ή φυσικό και να εκτελεστεί. Όπως και ένα κλασικό εμπορευματοκιβώτιο, είναι ανταλλάξιμο, φορητό και όσο πιο γενικό γίνεται [4].

2.3 Εικονικές Μηχανές έναντι Containers

Για πολλά χρόνια, οι εικονικές μηχανές ήταν η αιχμή του δόρατος της εικονικοποίησης. Είχαν το τέλειο λογικό υπόβαθρο: αντί για ένα δωμάτιο γεμάτο από εξυπηρετητές, μπορεί να υπάρξει ένας μόνο εξυπηρετητής που να προσομοιώνει όλα αυτά τα συστήματα. Αλλά τι γίνεται με τα containers; Η ιδέα τους δεν είναι τόσο καινούρια αλλά οι τελευταίες υλοποιήσεις τους τα έχουν καταστήσει μια ανερχόμενη δύναμη στον τομέα της πληροφορικής και υπάρχει σοβαρός λόγος για αυτό: είναι αρκετά φθηνά, εύκολα στη χρήση, φορητά και έχουν υψηλή κλιμάκωση.

Εδώ ανέρχεται λοιπόν το ερώτημα: ποιο από τα δύο είναι πιο κατάλληλο; η απάντηση δεν είναι ιδιαίτερα προφανής αν δεν αναλυθούν πρώτα κάποιες λεπτομέρειες.

Για τις εικονικές μηχανές τα πράγματα είναι λίγο πολύ γνωστά. Η τεχνολογία αυτή είναι αρκετά οικεία στο ευρύ κοινό, έχει μικρή καμπύλη εκμάθησης και μπορεί να προσομοιώσει οποιοδήποτε σύστημα σε μικρό χρονικό διάστημα. Επίσης, οι εικονικές μηχανές δίνουν τη δυνατότητα για εγκατάσταση πολλών εφαρμογών μαζί σε μία μηχανή καθώς η λειτουργία τους περιλαμβάνει την προσομοίωση ολόκληρου λειτουργικού συστήματος. Με αυτές μπορεί να δημιουργηθεί ένα συμπαγές σύστημα που προσφέρει όλες τις υπηρεσίες που είναι αναγκαίες στην στιγμή.

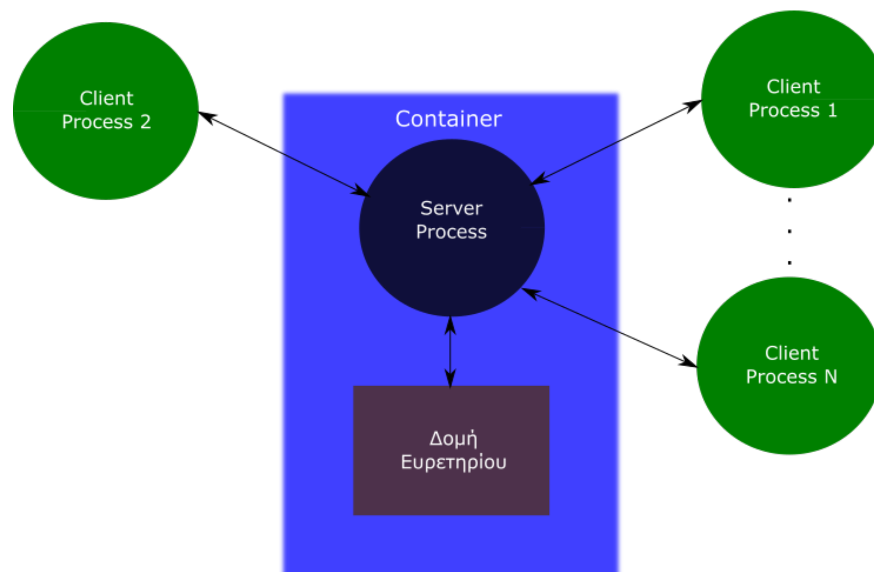
Για τα containers, η κατάσταση είναι πιο πρόσφατη. Μπορούν να κάνουν την ζωή εύκολη, ειδικά όταν πρέπει να αναπτυχθούν πολλαπλά στιγμιότυπα μιας εφαρμογής ή υπηρεσίας, πολύ πιο αποδοτικά από τις εικονικές μηχανές. Για παράδειγμα, μπορεί να χρειάζεται να αναπτυχθούν πολλαπλοί Apache servers γρήγορα. Επειδή μιλάμε για ανάπτυξη απλών υπηρεσιών, τα containers απαιτούν μακράν μικρότερη χρήση υλικών πόρων απ' ότι μια εικονική μηχανή. Επίσης, ο χρόνος εκκίνησης τους είναι συνήθως δευτερόλεπτα σε αντίθεση με ένα πλήρως εικονικοποιημένο σύστημα που χρειάζεται κάποια λεπτά.

Μετά και από αυτές τις λεπτομέρειες η απάντηση στο ερώτημα γίνεται, αν όχι εξαιρετικά απλή, εξαιρετικά εμφανής. Αν η ανάγκη είναι η δημιουργία μιας πλήρους πλατφόρμας με πλήρη απομόνωση, δικούς της πόρους και πολλαπλές υπηρεσίες εγκατεστημένες, τότε η

πιο κατάλληλη επιλογή είναι οι εικονικές μηχανές. Αντίθετα, αν η ανάγκη είναι η ανάπτυξη και η απομόνωση πολλών στιγμιότυπων υπηρεσιών ταυτόχρονα, τότε η πιο κατάλληλη επιλογή είναι τα containers [12][14].

3. CONTAINERIZED ΥΠΗΡΕΣΙΑ ΕΥΡΕΤΗΡΙΟΥ ΒΔ

Σε αυτό το κεφάλαιο στόχος είναι η δημιουργία μίας containerized υπηρεσίας ευρετηρίου ΒΔ χρησιμοποιώντας το υπόβαθρο για την εικονικοποίηση που αναλύσαμε μέχρι τώρα. Θα δούμε αναλυτικά όλα τα βασικά στοιχεία της υπηρεσίας ευρετηρίου ΒΔ, τις δομές, τις συναρτήσεις, την εκτέλεση και την συμπεριφορά. Η υπηρεσία είναι γραμμένη στην γλώσσα C, φιλοξενείται σε ένα Docker container και αποτελείται από τέσσερεις βασικούς άξονες: το πρόγραμμα πελάτη, το πρόγραμμα του εξυπηρετητή, το container και τη δομή ευρετηρίου.



Σχήμα 9: Οι βασικοί άξονες της υπηρεσίας

Τα πρόγραμμα πελάτη είναι αυτό που χειρίζεται ο χρήστης. Πρόκειται για την διεπαφή μεταξύ του χρήστη και του ευρετηρίου. Ο χρήστης πληκτρολογεί τις εντολές του και το πρόγραμμα πελάτη της μεταβιβάζει στον εξυπηρετητή.

Το πρόγραμμα εξυπηρετητή είναι ουσιαστικά το επίπεδο ανάμεσα στον χρήστη και το ευρετήριο. Είναι αυτό που λαμβάνει τις εντολές από το χρήστη και υποβάλει τις αλλαγές στη βάση. Όπως κάθε εξυπηρετητής μπορεί να εξυπηρετεί πολλούς χρήστες ταυτόχρονα μέσω νημάτων. Επίσης, είναι ο μόνος υπεύθυνος για την διαχείριση της βάσης δεδομένων και του σχετικού αρχείου.

Το container που περιέχει την υπηρεσία είναι αυτό που εγγυάται την απρόσκοπτη λειτουργία της. Αυτό επιτρέπει στην υπηρεσία να είναι απομονωμένη από το υπόλοιπο σύστημα. Επίσης της δίνει τη δυνατότητα να αναπτυχθεί σε οποιοδήποτε σύστημα έχοντας ακριβώς τις ίδιες προδιαγραφές περιβάλλοντος ανεξαρτήτως υλικού, ΛΣ ή ρυθμίσεων συστήματος.

Η δομή ευρετηρίου είναι το χαμηλότερο επίπεδο της υπηρεσίας και υπεύθυνη για όλη την «γραφειοκρατία». Εισάγει και διαγράφει εγγραφές από τη βάση και συλλέγει σύνολα από εγγραφές που ικανοποιούν τα εκάστοτε ερωτήματα των χρηστών. Η λειτουργία της διαφοροποιείται λίγο από τις κλασικες δομές ευρετηρίων για να εξυπηρετηθεί η ταυτόχρονη προσπέλασή της από πολλούς χρήστες.

Στην συνέχεια θα δούμε τους παραπάνω άξονες σε μία από τα χαμηλά προς τα ψηλά (bottom-up) προσέγγιση με παράθεση, όπου αυτό κριθεί αναγκαίο, κομμάτια από τον δικό μου κώδικα της υπηρεσίας.

3.1 Δομή Ευρετηρίου

Η δομή ευρετηρίου βρίσκεται στην βάση της ”ιεραρχίας” και αποτελεί ανεξάρτητο κομμάτι της υπηρεσίας. Δέχεται τα ερωτήματα των χρηστών από το επίπεδο του εξυπηρετητή και επιστρέφει τις εγγραφές της βάσης που ικανοποιούν αυτά τα ερωτήματα. Φυσικά η δομή αυτή θα είναι παραλλαγμένη έτσι ώστε να εξυπηρετεί πολλούς χρήστες ταυτόχρονα.

3.1.1 B^{link}-Δέντρο

Η δομή που θα χρησιμοποιηθεί θα είναι ένα B^+-Δέντρο με κάποιες παραλλαγές. Οι παραλλαγές αυτές θα επιτρέπουν την ταυτόχρονη προσπέλαση της δομής από πολλούς χρήστες χωρίς την ανάγκη για αμοιβαίως αποκλειόμενη προσέγγιση. Η νέα αυτή δομή ονομάζεται B^{link}-Δέντρο και μελετήθηκε από τον D. Lomet. Προσφέρει μια απλή, δυναμική και υψηλά ταυτόχρονη προσέγγιση. Κύριο μέλημα της δομής αυτής είναι να παρέχει έλεγχο ταυτόχρονης προσπέλασης και ανάκαμψης του συστήματος αφήνοντας τις λεπτομέρειες της αναζήτησης και της ενημέρωσης μεταξύ των κόμβων του δέντρου να αποφασίζονται από τα συγκεκριμένα δεδομένα που ευρετηριοποιούνται.

Ένα B^{link}-Δέντρο είναι μία δενδρική δομή ευρετηρίου αποτελούμενη από πολλά επίπεδα. Το κατώτερο επίπεδο αποτελείται από κόμβους φύλλα οι οποίοι περιέχουν μία καταχώρηση για κάθε τιμή του πεδίου κλειδιού μαζί με ένα δείκτη προς την αντίστοιχη εγγραφή (ή προς το μπλοκ που περιέχει την εγγραφή). Οι κόμβοι-φύλλα συνήθως συνδέονται μεταξύ τους προκειμένου να παράσχουν διατεταγμένη ως προς το πεδίο αναζήτησης προσπέλαση των εγγραφών. Πιο αναλυτικά η δομή των κόμβων-φύλλων ενός B^{link}-Δέντρου έχει ως εξής: Κάθε κόμβος-φύλλο είναι της μορφής $< <K_1, P_1>, <K_2, P_2>, \ldots, <K_n, P_n>, P_{next}>$ όπου κάθε P_i είναι ένας δείκτης δεδομένων που δείχνει στην εγγραφή της οποίας η τιμή στο πεδίο αναζήτησης είναι K_i, το P_{next} δείχνει στον επόμενο κόμβο-φύλλο του B^{link}-Δέντρου και ισχύει $K_1 < K_2 < \ldots < K_n$.

Τα ανώτερα επίπεδα του δέντρου αποτελούνται από εσωτερικούς κόμβους οι οποίοι είναι οι οδηγοί προς τους κόμβους-φύλλα. Οι κόμβοι αυτοί περιέχουν ζεύγη κλειδιών και δεικτών δέντρου. Οι δείκτες δέντρου είναι δείκτες προς μπλοκ που είναι κόμβοι δέντρου, δηλαδή είτε άλλοι εσωτερικοί είτε φύλλα. Συνήθως οι εσωτερικοί κόμβοι έχουν και αυτοί δείκτες προς τους διπλανούς κόμβους του ίδιου επιπέδου. Πιο αναλυτικά η δομή των εσωτερικών κόμβων ενός B^{link}-Δέντρου έχει ως εξής: Κάθε εσωτερικός κόμβος είναι της μορφής $< P_0, < K_1, P_1 >, < K_2, P_2 >, \ldots, < K_n, P_n >, P_n ext>$ όπου κάθε P_i είναι ένας δείκτης δέντρου και ισχύει $K_1 < K_2 < \ldots < K_n$. Για όλες τις τιμές V του πεδίου κλειδιού ισχύει $K_{i-1} \leq V < K_{i+1}$ για $1 < i < n$, $V < K_i$ για i=1 και $K_n \leq V$ [1].

Το στοιχείο που κάνει το B^{link}-Δέντρο να ξεχωρίζει από άλλες δομές ευρετηρίου είναι η

δυνατότητά του να εξυπηρετεί πολλούς χρήστες ταυτόχρονα. Αυτό επιτυγχάνεται μέσω μιας προσέγγισης με κλειδωνιές (latches). Οι κλειδωνιές είναι μηχανισμοί συγχρονισμού μεταξύ νημάτων υπεύθυνες για την επίλυση race conditions και αδιεξόδων. Είναι πιο ελαφριές από τις κλειδαριές (locks). Κάθε κόμβος του δέντρου έχει και μια δική του κλειδωνιά και κάθε κλειδωνιά έχει τρεις μορφές: share, update και exclusive [5]. Οι κλειδωνιές τύπου share είναι συμβατές μεταξύ τους και με τις update και αφορούν στο διάβασμα δεδομένων. Οι κλειδωνιές τύπου update δεν είναι συμβατές μεταξύ τους και αφορούν στην ενημέρωση δεδομένων. Οι κλειδωνιές τύπου exclusive δεν είναι συμβατές με καμία άλλη κλειδωνιά και αφορούν στην εισαγωγή ή διαγραφή δεδομένων και σε τροποποιήσεις της δομής.

Αυτό λοιπόν που επιτυγχάνουν οι κλειδωνιές είναι η ταυτόχρονη προσπέλαση του δέντρου από πολλούς χρήστες μέσω της τεχνικής των συζυγών κλειδωνιών (latch coupling) [5]. Με αυτή την τεχνική από ένα κλειδωμένο κόμβο n_1, ένας κόμβος n_2 που δεικτοδοτείται από τον n_1 κλειδώνεται πριν απελευθερωθεί η κλειδωνιά του n_1. Αυτό μας εξασφαλίζει τόσο το ότι ο κόμβος n_2 δεν μπορεί να διαγραφεί μεταξύ της αναφοράς και της πρόσβασης σε αυτόν, όσο και το ότι ο καθένας που χρησιμοποιεί τη δομή κρατάει διαδοχικά μόνο τους κόμβους που χρειάζεται πρόσβαση σε μια δεδομένη στιγμή αφήνοντας το υπόλοιπο δέντρο ελεύθερο. Σε περίπτωση από κάτω προς τα πάνω διάσχισης του δέντρου είναι εύλογο ότι η τεχνική αυτή δεν μπορεί να χρησιμοποιηθεί, καθώς μπορούν να δημιουργηθούν αδιέξοδα.

3.1.2 Υλοποίηση B^{link}-Δέντρου

Στο κεφάλαιο αυτό θα δούμε μερικά σημαντικά σημεία του κώδικά μου για το B^{link}-Δέντρο.

Αρχικά, έχουμε τη δόμηση των εγγραφών στα μπλοκ του ευρετηρίου. Η δομή που χαρακτηρίζει μια εγγραφή είναι η εξής:

```
typedef struct DataRecord {
	int key;
	char payload[64];
} DataRecord;
```

Σχήμα 10: Δομή εγγραφής δεδομένων.

Όπως φαίνεται από την παραπάνω εικόνα κάθε εγγραφή δεδομένων αποτελείται από 2 πεδία σταθερού μήκους: ένα πεδίο ακεραίου που είναι και το κλειδί του ευρετηρίου και μπορεί να εκφράζει id, ηλικία, μισθό κ.α. και ένα πεδίο συμβολοσειρά μήκους 64 χαρακτήρων που είναι και το φορτίο (payload) της εγγραφής. Αντίστοιχα, κάθε εγγραφή εσωτερικού κόμβου του ευρετηρίου αποτελείται από δύο πεδία: έναν ακέραιο κλειδί και έναν δείκτη προς μπλοκ.

```
typedef struct IndexRecord {
        int key;
        int pointer;
} IndexRecord;
```

Σχήμα 11: Δομή εγγραφής εσωτερικού κόμβου.

Έπειτα, κάθε μπλοκ δέχεται ακέραιο αριθμό εγγραφών και διαθέτει κεφαλίδα που περιέχει τον τύπο του μπλοκ (’d’ για μπλοκ δεδομένων, ’i’ για εσωτερικό μπλοκ), αριθμό εγγραφών και τον πλάγιο δείκτη σε αδελφό μπλοκ. Η διαχείριση του επιπέδου μπλοκ γίνεται από την βιβλιοθήκη BF η οποία δίνεται ως βοηθητικό υλικό του μαθήματος ”Υλοποίηση Συστημάτων Βάσεων Δεδομένων” του τμήματος Πληροφορικής και Τηλεπικοινωνιών.

```
typedef struct BlockHeader {
        char type;
        int next;
        int records;
} BlockHeader;
```

Σχήμα 12: Δομή κεφαλίδας μπλοκ.

Όσον αφορά στο συγχρονισμό του δέντρου με τις κλειδωνιές που περιγράφηκε παραπάνω, ακολουθείται μία προσέγγιση με mutexes της βιβλιοθήκης pthread. Πιο συγκεκριμένα, διατηρείται μία δομή η οποία περιλαμβάνει ένα mutex και έναν πίνακα ακεραίων μεγέθους όσο και ο αριθμός των μπλοκ του αρχείου. Κάθε θέση του πίνακα αρχικοποιείται στο 0 που σημαίνει ότι το αντίστοιχο μπλοκ είναι ελεύθερο.

```
typedef struct Latch {
        int blocks;
        int *block_latch;
        pthread_mutex_t mtx;
        pthread_cond_t cond_share;
        pthread_cond_t cond_exclusive;
} Latch;
```

Σχήμα 13: Δομή κλειδωνιάς.

Όταν ένα νήμα χρειάζεται πρόσβαση τύπου SHARE, τότε ελέγχει αν η θέση του πίνακα είναι μεγαλύτερη ή ίση του μηδέν και αυξάνει τον μετρητή αυτόν κατά ένα. Ένας θετικός ακέραιος αριθμός σε μια θέση του πίνακα συμβολίζει πόσα νήματα έχουν αποκτήσει πρόσβαση τύπου SHARE στο συγκεκριμένο μπλοκ. Όταν ένα νήμα χρειάζεται πρόσβαση τύπου EXCLUSIVE, τότε ελέγχει αν η θέση του πίνακα είναι ακριβώς μηδέν και τότε και μόνο τότε αποκτάει πρόσβαση στο μπλοκ αναθέτοντας στον μετρητή την τιμή -1. Σε περίπτωση που οι παραπάνω συνθήκες δεν ισχύουν τότε τα νήματα περιμένουν πάνω σε μία condition variable μέχρι να ελευθερωθεί το μπλοκ που επιθυμούν.

```
void acquireShareLatch(LatchPtr latch, int block) {
	pthread_mutex_lock(&(latch->mtx));
	while (latch->block_latch[block - 1] == UPDATE
	|| latch->block_latch[block - 1] == EXCLUSIVE) {
		pthread_cond_wait(&(latch->cond_share), &(latch->mtx));
	}
	latch->block_latch[block - 1]++;
	pthread_mutex_unlock(&(latch->mtx));
}
```

Σχήμα 14: Ρουτίνα απόκτησης κλειδωνιάς τύπου SHARE.

Κατά την απελευθέρωση μιας κλειδωνιάς το νήμα που το κάνει ενημερώνει μέσω broadcast τα υπόλοιπα νήματα που περιμένουν για την πιθανή αλλαγή στην κατάσταση ενός μπλοκ.

```
void releaseShareLatch(LatchPtr latch, int block) {
	pthread_mutex_lock(&(latch->mtx));
	latch->block_latch[block - 1]--;
	pthread_cond_broadcast(&(latch->cond_exclusive));
	pthread_cond_broadcast(&(latch->cond_share));
	pthread_mutex_unlock(&(latch->mtx));
}
```

Σχήμα 15: Ρουτίνα άφεσης κλειδωνιάς τύπου SHARE.

Παρακάτω θα δούμε τις βασικές συναρτήσεις του δέντρου: εισαγωγή, διαγραφή, ερωτήματα.

Εισαγωγή εγγραφής

Για την εισαγωγή μιας εγγραφής στο δέντρο ακολουθείται μια αναδρομική προσέγγιση. Ξεκινάμε την κατά βάθος διάσχιση του δέντρου από τον κόμβο ρίζα ακολουθώντας τους κατάλληλους δείκτες μέχρι να φτάσουμε στο σωστό μπλοκ δεδομένων. Σε κάθε μπλοκ εσωτερικού κόμβου γίνεται αναζήτηση του δείκτη που ταιριάζει στο κλειδί εισαγωγής. Η αναζήτηση γίνεται σειριακά μέχρι να ικανοποιηθούν τα κριτήρια του B^+-Δέντρου όπως περιγράφηκαν. Μόλις βρεθεί ο κατάλληλος δείκτης κλειδώνεται το νέο μπλοκ που δείχνει και καλείται αναδρομικά η συνάρτηση με αυτό ως κύριο μπλοκ.

```
if (header->type == 'i') {
	index_data = (IndexRecord*) skipHeader(header);
	// if key is lesser than leftmost block key insert at the leftmost
pointer
	if (key <= *left_pointer) {
		acquireShareLatch(latch, *left_pointer);
		releaseShareLatch(latch, block_num);
		new_key = insertBTreeRec(file_desc, *left_pointer, key, payload,
latch);
	} else {
		// Find key position
		i = findPosition(header, key);
		// Go down one level
		acquireShareLatch(latch, index_data[i-1].pointer);
		releaseShareLatch(latch, block_num);
		new_key = insertBTreeRec(file_desc, index_data[i-1].pointer, key,
payload, latch);
}
```

Σχήμα 16: Αναζήτηση δείκτη σε μπλοκ εσωτερικού κόμβου.

Όταν φτάσουμε στο κατάλληλο μπλοκ δεδομένων για εισαγωγή τότε επιχειρούμε την εισαγωγή της εγγραφής στο μπλοκ. Αν υπάρχει χώρος τότε την τοποθετούμε στο μπλοκ στην κατάλληλη θέση πάλι με σειριακή αναζήτηση. Αν, όμως, δεν υπάρχει διαθέσιμος χώρος τότε πρέπει να γίνει διαχωρισμός του μπλοκ. Όταν γίνει ο διαχωρισμός του μπλοκ τότε αποφασίζεται εκ νέου σε ποιο μπλοκ θα μπει η εγγραφή, το παλιό ή το καινούριο. Στο τέλος της εισαγωγής επιστρέφουμε αναδρομικά στο προηγούμενο επίπεδο (μπλοκ) με τιμή επιστροφής την τιμή που πρέπει να διαδοθεί προς τα πάνω σε περίπτωση διαχωρισμού ή την τιμή -1 σε περίπτωση ομαλής εισαγωγής.

```
else if (header->type == 'd') {
	int pos = findPosition(header, key);
	releaseShareLatch(latch, block_num);
	acquireExclusiveLatch(latch, block_num, EXCLUSIVE);
	if ((header->records+1) <= ((BLOCK_SIZE - sizeof(int) -
	(sizeof(BlockHeader)))/sizeof(DataRecord))) {
		/* If record fits in block */
		dataInsertNonfull(file_desc,
		                  header,
		                  block_num,
		                  pos,
		                  key,
		                  payload);
		releaseExclusiveLatch(latch, block_num);
		// Inform the upper level that no split happened
		return -1;
	} else {
		/* If record doesn't fit in block */
		// Split block
		int return_key = splitBlock(file_desc,
		                            header,
		                            &new_header,
		                            new_key,
		                            'd');
		// According to original position decide where to insert record
		if (pos <= (header->records)) {
			dataInsertNonfull(file_desc,
			                  header,
			                  block_num,
			                  pos,
			                  key,
			                  payload);
		} else {
			dataInsertNonfull(file_desc,
			                  new_header,
			                  BF_GetBlockCounter(file_desc) - 1,
			                  pos-header->records,
			                  key,
			                  payload);
		}
		releaseExclusiveLatch(latch, block_num);
		/* Post split term */
		resizeLatch(latch);
		// Propagate split key to the upper level
		return return_key;

	}
```

Σχήμα 17: Εισαγωγή εγγραφής σε μπλοκ δεδομένων.

Ο διαχωρισμός μπλοκ γίνεται σε τρία βήματα. Πρώτα δημιουργείται ένα καινούριο άδειο μπλοκ είτε δεδομένων είτε ευρετηρίου ανάλογα με το μπλοκ που είναι υπό διαχωρισμό. Στη συνέχεια, χωρίζονται στη μέση οι εγγραφές του παλιού μπλοκ και το άνω μισό μεταφέρεται στο νέο μπλοκ. Κάθε μπλοκ έχει ίσο αριθμό εγγραφών μετά το διαχωρισμό αν ο αρχικός αριθμός εγγραφών ήταν ζυγός ή ένα μπλοκ έχει μία παραπάνω εγγραφή, συνήθως το καινούριο, αν ήταν μονός. Τελικά ενημερώνονται οι κεφαλίδες των δύο μπλοκ κατάλληλα και ένα κλειδί επιλέγεται κατάλληλα έτσι ώστε να διαδοθεί στο ανώτερο επίπεδο. Το κλειδί αυτό φυσικά δεν επιλέγεται τυχαία αλλά επιλέγεται μεταξύ των υπαρχόντων κλειδιών. Για μπλοκ δεδομένων συνήθως επιλέγεται το κλειδί της τελευταίας εγγραφής στο παλιό μπλοκ, ενώ για μπλοκ ευρετηρίου επιλέγεται το κλειδί που είχε ο αριστερός δείκτης του νέου μπλοκ πριν το διαχωρισμό.

```
if (type == 'i') {
        IndexRecord *new_sibling, *old_sibling;
        old_sibling = (IndexRecord*) skipHeader(header);
        new_sibling = (IndexRecord*) skipHeader(*new_header);
        block = (int*) new_sibling + sizeof(BlockHeader) / sizeof(int) + 1;
        // Split old block records in half
        (*new_header)->records = header->records - treshold;
        // Move second half records to new block
        for (i = 0; i < (*new_header)->records; ++i) {
                new_sibling[i].key = old_sibling[treshold+i].key;
                new_sibling[i].pointer = old_sibling[treshold+i].pointer;
        }
        // Make dangling pointer of the upgoing key the leftmost pointer of the
new block
        *block = old_sibling[treshold - 1].pointer;
        // Update sibling pointers
        (*new_header)->next = header->next;
        header->records = header->records - (*new_header)->records;
        header->next = BF_GetBlockCounter(file_desc) - 1;
        // Return upgoing key
        return old_sibling[treshold - 1].key;
}
```

Σχήμα 18: Διαχωρισμός μπλοκ εσωτερικού κόμβου.

```
else {
        DataRecord *new_sibling, *old_sibling;
        old_sibling = (DataRecord*) skipHeader(header);
        new_sibling = (DataRecord*) skipHeader(*new_header);
        block = (int*) new_sibling + sizeof(BlockHeader) / sizeof(int) + 1;
        // Split old block records in half
        (*new_header)->records = header->records - treshold;
        // Move second half records to new block
        for (i = 0; i < (*new_header)->records; ++i) {
                int boundary = (strlen(old_sibling[treshold+i].payload) < 64 ?
                               strlen(old_sibling[treshold+i].payload) + 1 : 64);
                new_sibling[i].key = old_sibling[treshold+i].key;
                strncpy(new_sibling[i].payload,
                        old_sibling[treshold+i].payload,
                        boundary);
        }
        // Update sibling pointers
        (*new_header)->next = header->next;
        header->records = header->records - (*new_header)->records;
        header->next = BF_GetBlockCounter(file_desc) - 1;
        // Return upgoing key
        return old_sibling[treshold - 1].key;
}
```

Σχήμα 19: Διαχωρισμός μπλοκ δεδομένων.

Τέλος, όταν ολοκληρωθεί η αναδρομική αυτή διαδικασία επιστρέφουμε στην αρχική κλήση με το μπλοκ ρίζα. Αν υπάρχει μέχρι εδώ κλειδί που έχει διαδοθεί τότε εκτός από το διαχωρισμό της ρίζας θα πρέπει να δημιουργηθεί και ένα νέο μπλοκ το οποίο και θα είναι η νέα ρίζα του δέντρου.

Διαγραφή εγγραφής

Η διαγραφή μιας εγγραφής δεν γίνεται αναδρομικά. Αντίθετα, βρίσκουμε το μπλοκ δεδομένων που είναι η εγγραφή με το συγκεκριμένο κλειδί προς διαγραφή. Βρίσκουμε με σειριακή αναζήτηση τη θέση της πρώτης εγγραφής με το κλειδί αναζήτησης και το διαγράφουμε μετακινώντας όλες τις εγγραφές που προηγούνται κατά μία θέση αριστερά.

```
acquireExclusiveLatch(latch, selected_block, EXCLUSIVE);
header = (BlockHeader*) block;
pos = findPosition(header, key);
data = (DataRecord*) skipHeader(header);
if (pos != header->records) {
        // Deletion in middle
        // Shift records one position left
        for (i = header->records; i != pos; --i) {
                strncpy(data[i-1].payload,
                        data[i].payload,
                        strlen(data[i].payload));
                data[i-1].key = data[i].key;
        }
}
header->records--;
BF_WriteBlock(tree->file_desc, selected_block);
releaseExclusiveLatch(latch, selected_block);
```

Σχήμα 20: Διαγραφή εγγραφής.

Αναζήτηση εγγραφών

Το γεγονός ότι η δομή ευρετηρίου που χρησιμοποιούμε είναι ουσιαστικά ένα B^+-Δέντρο μας επιτρέπει την διεξαγωγή και ερωτημάτων εύρους τιμών. Με αυτό υπ’ όψιν ένας χρήστης έχει 6 διαφορετικές μεθόδους αναζήτησης εγγραφών με βάση μια τιμή κλειδιού: ίσο ($=$), διάφορο ($\neq$), μικρότερο ($<$), μικρότερο-ίσο ($\leq$), μεγαλύτερο ($>$), μεγαλύτερο-ίσο ($\geq$).

Αναζήτηση ίσου. Η αναζήτηση ίσου είναι ίσως η πιο απλή μέθοδος αναζήτησης. Το μόνο που χρειάζεται είναι μία κατά βάθος διάσχιση του δέντρου μέχρι το κατάλληλο μπλοκ δεδομένων και αναζήτηση στο μπλοκ αυτό για εγγραφές με τιμή κλειδιού ίση με τον όρο αναζήτησης.

```
case EQUAL: // ==
        current_block = findBlock(tree, key, latch);
        acquireShareLatch(latch, current_block);
        if (BF_ReadBlock(tree->file_desc, current_block, (void**) &block) < 0) {
                BF_PrintError("Error reading block");
                return NULL;
        }
        header = (BlockHeader*) block;
        data = (DataRecord*) skipHeader(header);

        for (i = 0; i < header->records; i++) {
                if (data[i].key == key) {
                        insertPayload(result, data[i].payload);
                }
        }
        releaseShareLatch(latch, current_block);
        break;
```

Σχήμα 21: Αναζήτηση εγγραφής με τελεστή ίσον ($=$).

Αναζήτηση διάφορου. Η αναζήτηση εγγραφών που έχουν κλειδί διάφορο του όρου αναζήτησης είναι λίγο πιο περίπλοκο ζήτημα. Κυρίως λόγω της δομής του ευρετηρίου που όπως γνωρίζουμε κάθε μπλοκ έχει έναν δείκτη μόνο στο δεξιό αδελφό μπλοκ του και όχι ανάποδα. Επομένως, δεν μπορεί να ακολουθηθεί η τακτική του ίσου τελεστή να βρεθεί η

εγγραφή με κλειδί τον όρο αναζήτησης και απλά να αποκλειστεί από το αποτέλεσμα διασχίζοντας όλες τις άλλες εγγραφές. Φυσικά αυτό θα μπορούσε να γίνει είτε απλά προσθέτοντας δείκτες και προς την αντίθετη κατεύθυνση, είτε κάνοντας παραπάνω προσπελάσεις σε μπλοκ για να προσπελάσουμε και τις μικρότερες εγγραφές. Στην παρούσα υλοποίηση θα χρησιμοποιήσουμε μια πιο ”lazy” τεχνική, αλλά αρκετά αποδοτική, αποθηκεύοντας στα μεταδεδομένα του αρχείου το αριστερότερο μπλοκ δεδομένων. Έτσι, προσπελαύνουμε αυτό το μπλοκ και με σειριακή αναζήτηση με τους πλάγιους δείκτες διαλέγουμε τις κατάλληλες εγγραφές.

```
case NOT_EQUAL: // !=
	current_block = tree->first_data_block;
	while (current_block != -1) {
		acquireShareLatch(latch, current_block);
		if (BF_ReadBlock(tree->file_desc, current_block, (void**) &block)
< 0) {
			BF_PrintError("Error reading block");
			return NULL;
		}
		header = (BlockHeader*) block;
		data = (DataRecord*) skipHeader(header);
		for (i = 0; i < header->records; i++) {
			if (data[i].key != key) {
				insertPayload(result, data[i].payload);
			}
		}
		releaseShareLatch(latch, current_block);
		current_block = header->next;
	}
	break;
```

Σχήμα 22: Αναζήτηση εγγραφής με τελεστή διάφορο ($\neq$).

```
case LESS: // <
case LESS_EQUAL: // <=
	// Start from the leftmost data block
	current_block = tree->first_data_block;
	while (current_block != -1) {
		acquireShareLatch(latch, current_block);
		if (BF_ReadBlock(tree->file_desc, current_block, (void**) &block)
< 0) {
			BF_PrintError("Error reading block");
			return NULL;
		}
		header = (BlockHeader*) block;
		data = (DataRecord*) skipHeader(header);
		for (i = 0; i < header->records; i++) {
			if (operator == LESS) {
				if (data[i].key < key) {
					insertPayload(result, data[i].payload);
				}
			}
			else if (operator == LESS_EQUAL) {
				if (data[i].key <= key) {
					insertPayload(result, data[i].payload);
				}
			}
		}
		releaseShareLatch(latch, current_block);
		current_block = header->next;
	}
	break;
```

Σχήμα 23: Αναζήτηση εγγραφής με τελεστές μικρότερο ($<$) και μικρότερο-ίσο ($\leq$).

```
case GREATER: // >
case GREATER_EQUAL: // >=
        current_block = findBlock(tree, key, latch);
        while (current_block != -1) {
                acquireShareLatch(latch, current_block);
                if (BF_ReadBlock(tree->file_desc, current_block, (void**) &block)
< 0) {
                        BF_PrintError("Error reading block");
                        return NULL;
                }
                header = (BlockHeader*) block;
                data = (DataRecord*) skipHeader(header);

                for (i = 0; i < header->records; i++) {
                        if (operator == GREATER) {
                                if (data[i].key > key) {
                                        insertPayload(result, data[i].payload);
                                }
                        }
                        else if (operator == GREATER_EQUAL) {
                                if (data[i].key >= key) {
                                        insertPayload(result, data[i].payload);
                                }
                        }
                }
                releaseShareLatch(latch, current_block);
                current_block = header->next;
        }
        break;
```

Σχήμα 24: Αναζήτηση εγγραφής με τελεστές μεγαλύτερο ($>$) και μεγαλύτερο-ίσο ($\geq$).

Αναζήτηση μικρότερου ή μικρότερου-ίσου. Η αναζήτηση εγγραφών που είναι μικρότερες (ή/και ίσες) από τον όρο αναζήτησης είναι ουσιαστικά η μισή διαδικασία της αναζήτησης διάφορου. Ξεκινάμε από το αριστερότερο μπλοκ δεδομένων και προχωράμε σειριακά μέχρι να βρούμε εγγραφή με κλειδί τον όρο αναζήτησης.

Αναζήτηση μεγαλύτερου ή μεγαλύτερου-ίσου. Σε αυτή την αναζήτηση κάνουμε ουσιαστικά μια αναζήτηση ίσου και προχωράμε σειριακά προς τα δεξιά μέχρι και την τελευταία εγγραφή.

3.2 Πρόγραμμα Εξυπηρετητή

Το πρόγραμμα του εξυπηρετητή είναι ο ενδιάμεσος μεταξύ της δομής ευρετηρίου και του χρήστη. Είναι υπεύθυνο τόσο για τα αιτήματα που δέχεται και την εξυπηρέτηση των χρηστών, όσο και για τον συγχρονισμό και την ομαλή λειτουργία της δομής του ευρετηρίου. Αποτελεί το εσωτερικό επίπεδο αφαίρεσης της δομής ευρετηρίου από το εξωτερικό περιβάλλον και ρυθμιστή της κυκλοφορίας αιτημάτων.

Ο εξυπηρετητής δέχεται αιτήματα σύνδεσης από χρήστες, τα αποθηκεύει σε μία ουρά αναμονής από όπου ο εσωτερικός μηχανισμός του τα περιλαμβάνει και αρχίζει να διαχειρίζεται τα ερωτήματα των χρηστών προς το ευρετήριο. Αυτός ο εσωτερικός μηχανισμός είναι ένα πλήθος από ξεχωριστά νήματα εκτέλεσης τα οποία είναι χτισμένα με το πρότυπο της δεξαμενής νημάτων (thread pool). Ουσιαστικά, ο εξυπηρετητής δημιουργεί κατά την εκκίνηση του ένα πλήθος από νήματα (συνήθως δύναμη του 2), τα οποία περιμένουν πάνω από την ουρά αναμονής αιτημάτων για κάποιο αίτημα χρήστη. Μόλις έρθει ένα τέτοιο αίτημα

ένα από τα νήματα αυτά συνδέεται αποκλειστικά με τον χρήστη που το έστειλε και τον εξυπηρετεί έως ότου αυτός κλείσει την σύνδεση και επιστρέφει πίσω στην ουρά για νέες αιτήσεις.

3.2.1 Δομικά στοιχεία Εξυπηρετητή

Τα δομικά στοιχεία του εξυπηρετητή είναι τρία. Πρώτον, ένα νήμα το οποίο είναι υπεύθυνο για τις εντολές προς τον εξυπηρετητή από την κονσόλα, π.χ. τερματισμός. Δεύτερον, ένα νήμα το οποίο είναι υπεύθυνο για την υποδοχή των συνδέσεων των χρηστών. Τέλος, έναν αριθμό από νήματα, ο οποίος καθορίζεται κατά το ξεκίνημα του εξυπηρετητή, που είναι υπεύθυνα για την εξυπηρέτηση των αιτημάτων των χρηστών προς το ευρετήριο. Όλα αυτά τα δομικά χαρακτηριστικά συνθέτουν το παζλ του προγράμματος του εξυπηρετητή και υλοποιούν την βασική ιδέα που περιγράφηκε παραπάνω.

Η λειτουργία του εξυπηρετητή ξεκινάει με το άνοιγμα μιας υποδοχής παρακολούθησης (listening socket) η οποία δέχεται τα αιτήματα των χρηστών για σύνδεση σε μία συγκεκριμένη θύρα (port). Όταν προκύψει ένα τέτοιο αίτημα τότε ανοίγει μία νέα υποδοχή η οποία και προστίθεται σε μία ουρά αναμονής. Στη συνέχεια ένα από τα αδρανή νήματα της δεξαμενής αποσύρει από την ουρά την υποδοχή αυτή και ξεκινάει την αποκλειστική εξυπηρέτηση των αιτημάτων του χρήστη μέχρι αυτός να κλείσει τη σύνδεση. Η διαδικασία αυτή συνεχίζεται κανονικά μέχρι τον τερματισμό του εξυπηρετητή.

Το πρώτο κομμάτι του εξυπηρετητή αφορά στην τοπική χρήση του, δηλαδή σε τοπικές εντολές κονσόλας. Μια τέτοια εντολή είναι και ο ομαλός τερματισμός του. Όταν δοθεί εντολή για τερματισμό του εξυπηρετητή τότε αυτός ενημερώνει το νήμα παρακολούθησης του δικτύου να σταματήσει να δέχεται αιτήματα μέσω μιας μεταβλητής και προετοιμάζει το έδαφος για τον ομαλό τερματισμό των νημάτων εξυπηρέτησης.

Το δεύτερο κομμάτι του εξυπηρετητή αποτελεί το νήμα παρακολούθησης του δικτύου. Το νήμα αυτό παρακολουθεί μέσω της συνάρτησης accept το δίκτυο για αιτήματα και μόλις έρθει αίτημα σύνδεσης από χρήστη τοποθετεί την υποδοχή που επιστρέφει η accept στην ουρά αναμονής.

```c
void *accept_connections(void *args) {
    int newsock;
    struct sockaddr_in client;
    struct sockaddr *clientptr = (struct sockaddr*) &client;
    socklen_t client_addr_len = sizeof(client);
    static struct sigaction act;

    act.sa_handler = exitSignal;
    sigfillset(&(act.sa_mask));
    sigaction(SIGUSR1, &act, NULL);

    // Accept incoming connections
    while((newsock = accept(sock, clientptr, &client_addr_len)) > 0) {
       // Put new connection to the waiting queue
       if (server_active) {
              logMessage(logfile, "Accepted connection from:",
                          inet_ntoa(client.sin_addr), 0);
              submitJob(scheduler, newsock);
       } else {
              close(newsock);
       }
    }

    close(sock);
    return NULL;
}
```

Σχήμα 25: Συνάρτηση παρακολούθησης δικτύου για αιτήματα σύνδεσης.

Το τρίτο και τελευταίο κομμάτι του εξυπηρετητή και ίσως το πιο βασικό είναι τα νήματα εξυπηρέτησης του χρήστη. Τα νήματα αυτά είναι πολλά και το πλήθος τους καθορίζεται κατά την εκκίνηση του εξυπηρετητή μέσω παραμέτρου. Ο απόλυτος υπεύθυνος αυτών των νημάτων είναι μία δομή που ονομάζεται job scheduler. Είναι υπεύθυνη για την δημιουργία, την κατάσταση και την καταστροφή αυτών των νημάτων και περιέχει όλα τα απαραίτητα στοιχεία για την λειτουργία και τον συγχρονισμό τους, όπως η ουρά αναμονής.

```c
typedef struct JobScheduler {
    BTreePtr tree;        //pointer to existing btree
    LatchPtr latch;

    int threads_count;          //number of threads
    Queue jobs_queue;          // Waiting Queue
    pthread_t *tids;            //array of threads

    pthread_mutex_t queue_mutex;    //mutex for queue synchronization
    pthread_cond_t cond_execute;    //worker threads wait for jobs execution
    pthread_cond_t cond_master;     //wait for worker threads to finish
} JobScheduler;
```

Σχήμα 26: Δομή job scheduler.

Κάθε νήμα εκτελεί την ίδια συνάρτηση. Για την εξυπηρέτηση των αιτημάτων του χρήστη, σε οποιαδήποτε εντολή, ακολουθούνται τρία βασικά στάδια:

1. **Προετοιμασία ή διαλογή.** Σε αυτό το βήμα γίνεται η αποδόμηση του αιτήματος του χρήστη. Αποσαφηνίζεται ποια εντολή έχει δώσει ο χρήστης (εισαγωγή, διαγραφή, ερώτημα) και στη συνέχεια γίνεται ο διαχωρισμός των πεδίων του αιτήματος ανάλογα

με την εντολή. Ο διαχωρισμός αυτός επιτυγχάνεται με την χρήση της συνάρτησης strtok. Αν κάποιο από τα πεδία λείπει ή δεν είναι σωστό τότε πάμε κατευθείαν στο βήμα 3 με απόκριση μηνύματος λάθους.

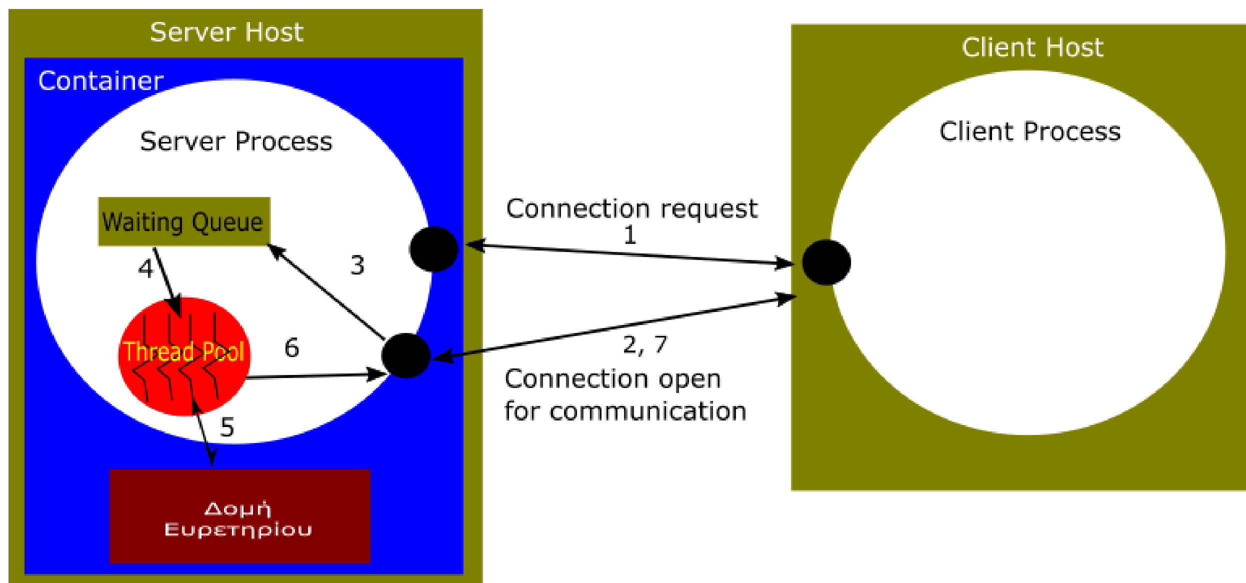
2. **Εκτέλεση ερωτήματος.** Σε αυτό το βήμα καλείται η κατάλληλη συνάρτηση του επιπέδου της δομής του ευρετηρίου σύμφωνα με τις παραμέτρους που εξήχθησαν από το 1ο βήμα.
3. **Απάντηση.** Στο τελικό αυτό βήμα ετοιμάζεται το μήνυμα απόκρισης προς το χρήστη είτε αυτό περιλαμβάνει επιβεβαίωση ή αποτυχία, είτε περιλαμβάνει αριθμό εγγραφών μετά από ερώτημα.

```
// Key
token = strtok_r(NULL, " \n", &saveptr);
if (token != NULL) {
	if ((key = atoi(token)) <= 0) {
		sprintf(response, "Error. Invalid Key inserted.\n");
		respondToClient(client_socket, response);
		continue;
	}
}
else {
	sprintf(response, "Error. No Key inserted.\n");
	respondToClient(client_socket, response);
	continue;
}
// Payload
token = strtok_r(NULL, "\n", &saveptr);
if (token == NULL) {
	sprintf(response, "Error. No payload inserted.\n");
	respondToClient(client_socket, response);
	continue;
}
```

Σχήμα 27: Φάση προετοιμασίας. Εδώ γίνεται η διαλογή του πεδίου κλειδιού και του πεδίου φορτίου.

Για τον τερματισμό των νημάτων ο job scheduler τοποθετεί στην ουρά αναμονής τόσες ειδικές υποδοχές TERMINATE_JOB όσες και το πλήθος των νημάτων και περιμένει. Μόλις ένα νήμα εξάγει από την ουρά την ειδική αυτή υποδοχή τερματίζει αμέσως και απελευθερώνονται οι πόροι του.

3.2.2 Κύκλος ζωής μιας σύνδεσης χρήστη στον εξυπηρετητή



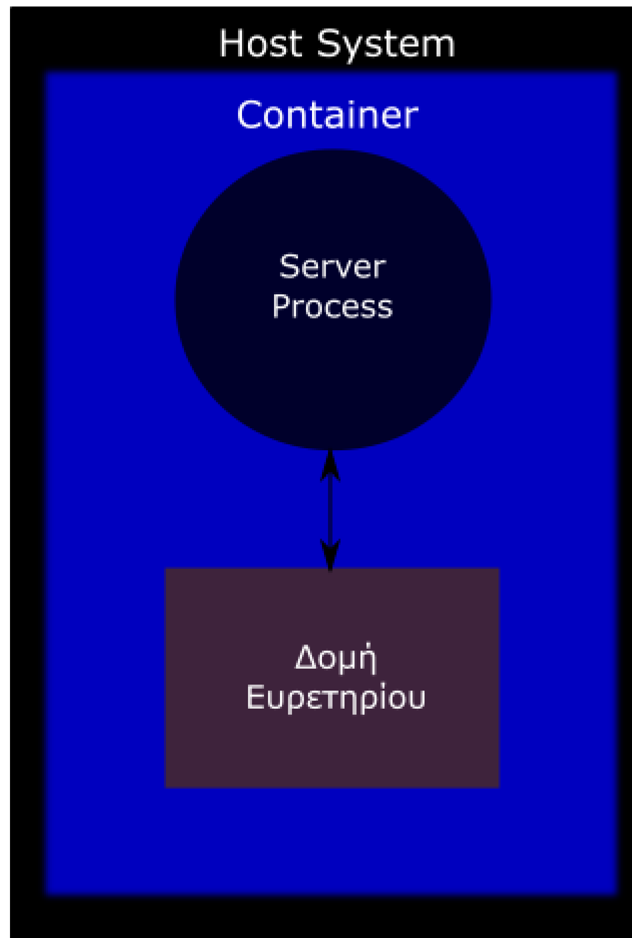
Σχήμα 28: Το μοντέλο του εξυπηρετητή. Ο πελάτης επικοινωνεί με την υποδοχή παρακολούθησης, το νήμα συνδέσεων δημιουργεί μία νέα υποδοχή την οποία τοποθετεί στην ουρά αναμονής και τα νήματα εξυπηρέτησης λαμβάνουν αυτές τις υποδοχές από την ουρά για την εξυπηρέτηση του πελάτη.

1. Ο χρήστης στέλνει ένα αίτημα σύνδεσης στην θύρα παρακολούθησης του εξυπηρετητή.
2. Ο εξυπηρετητής λαμβάνει το αίτημα και ανοίγει μία νέα υποδοχή (socket) αποκλειστική για τον νέο αυτόν πελάτη.
3. Η νέα αυτή υποδοχή τοποθετείται στην ουρά αναμονής και δίνεται σήμα σε τυχόν αδρανή νήματα να ελέγξουν εκ νέου την ουρά.
4. Αν υπάρχει διαθέσιμο νήμα τότε εξάγει την υποδοχή από την ουρά και ανοίγει δίαυλο επικοινωνίας αποκλειστικά με τον συγκεκριμένο πελάτη.
5. Όταν ο πελάτης στείλει κάποιο ερώτημα το νήμα που είναι υπεύθυνο για αυτόν επικοινωνεί με το ευρετήριο για να απαντήσει με τις κατάλληλες εγγραφές.
6. Μόλις ολοκληρωθεί η αναζήτηση στο ευρετήριο το νήμα προωθεί την απάντηση του στην υποδοχή του πελάτη.
7. Μέσω του δικτύου η απάντηση στέλνεται στον πελάτη ο οποίος την λαμβάνει και την χρησιμοποιεί καταλλήλως.
8. Η διαδικασία συνεχίζεται από το βήμα 5 έως ότου ο πελάτης κλείσει την σύνδεση.

3.3 Container Υπηρεσίας

Η υπηρεσία ευρετηρίου στο σύνολό της (πρόγραμμα εξυπηρετητή και δομή ευρετηρίου) περικλείεται μέσα σε ένα Linux container. Το επίπεδο του container αποτελεί το εξωτερικό επίπεδο αφαίρεσης της δομής ευρετηρίου και του εξυπηρετητή από το εξωτερικό

περιβάλλον. Προσφέρει απομόνωση, εύκολη μεταφορά και ανάπτυξη της εφαρμογής σε διαφορετικά περιβάλλοντα εκτέλεσης. Η πρακτική χρησιμότητα του container έχει τρεις άξονες.



Σχήμα 29: Σχηματική αναπαράσταση της θέσης του container σχετικά με την υπηρεσία και το σύστημα οικοδεσπότη.

Πρώτον, ένα container προσφέρει απομόνωση από το υπόλοιπο σύστημα. Όπως περιγράψαμε στο Κεφάλαιο 2 ένα container είναι σαν μία πιο ελαφριά εικονική μηχανή η οποία χρησιμοποιεί τον πυρήνα του ΛΣ του οικοδεσπότη για την λειτουργία της και περιέχει όλα τα εργαλεία που χρειάζεται για να εκτελεστεί αυτόνομα από το υπόλοιπο σύστημα μέσω των cgroups και του namespace isolation. Αυτό προσφέρει ένα έξτρα επίπεδο ασφάλειας στις εφαρμογές που τρέχουν στο container καθώς δεν επηρεάζονται τόσο από σφάλματα του οικοδεσπότη όσο και από κακόβουλες επιθέσεις και ευάλωττα σημεία του.

Δεύτερον, προσφέρει υψηλή φορητότητα από σύστημα σε σύστημα καθώς επιτρέπει στους προγραμματιστές να δουλεύουν σε προτυποποιημένα περιβάλλοντα χρησιμοποιώντας τοπικά containers. Σκεφτείτε το ακόλουθο σενάριο:

- Οι προγραμματιστές μιας εταιρείας ανάπτυξης λογισμικού αναπτύσουν τον κώδικά τους τοπικά και μοιράζονται την δουλειά τους με τους συναδέλφους τους μέσω των containers.

- Χρησιμοποιούν τα containers για να προωθήσουν τις εφαρμογές τους σε ένα περβάλλον ελέγχου της εφαρμογής όπου μπορούν να διεξάγουν αυτοματοποιημένους ή χειροκίνητους ελέγχους πάνω τους.
- Όταν οι προγραμματιστές βρουν bugs, μπορούν να τα διορθώσουν στο περιβάλλον ανάπτυξης και να τις επαναναπτύξουν στο περιβάλλον ελέγχου για έλεγχο και αξιολόγηση.
- Όταν ο έλεγχος ολοκληρωθεί η διανομή της επιδιόρθωσης είναι τόσο εύκολη όσο η προώθηση της ενημερωμένης εικόνας του container της εφαρμογής στο περιβάλλον παραγωγής.

Τρίτον, προσφέρει υψηλή κλιμάκωση σε ένα σύστημα καθώς είναι αρκετά ελαφρύ και γρήγορο, σε αντίθεση με τις εικονικές μηχανές, με αποτέλεσμα να μπορεί να χρησιμοποιηθεί περισσότερη υπολογιστική ισχύς από τα μηχανήματα μιας επιχείρησης.

Είδαμε, λοιπόν τη γενικότερη χρησιμότητα ενός container. Στην παρούσα εργασία θα επικεντρωθούμε περισσότερο στο πρώτο κομμάτι της απομόνωσης. Για την υλοποίηση του επιπέδου του container θα χρησιμοποιηθεί η πλατφόρμα του Docker. Όπως είδαμε στο προηγούμενο κεφάλαιο κάθε Docker container δημιουργείται από μία εικόνα. Κάθε εικόνα αποτελείται από πολλά επίπεδα συστημάτων αρχείων. Ευτυχώς, το Docker μας προσφέρει μία ειδική ψευδογλώσσα με την οποία μπορούμε αρκετά εύκολα να περιγράψουμε τη δομή του container μας με ένα γενικό τρόπο που το καθιστά ανεξάρτητο του συστήματος οικοδεσπότη. Το πρότυπο που ακολουθείται είναι παρόμοιο με αυτό του makefile για την μεταγλώττιση μιας εφαρμογής. Σε αυτή την περίπτωση έχουμε ένα αρχείο με όνομα Dockerfile το οποίο περιέχει, αντί για εντολές bash και gcc, εντολές για το Docker.

Ας δούμε λοιπόν και το Dockerfile της υπηρεσίας μας:

```
# Container image file for db service
FROM ubuntu:latest
RUN mkdir /db_service
EXPOSE 80
ENTRYPOINT ["/db_service/bin/db_server", "-p", "80", "-f", "/db_service/new.db"]
CMD ["-s", "4"]
```

Σχήμα 30: Dockerfile υπηρεσίας.

Όπως βλέπουμε στην πρώτη γραμμή, το container μας θα έχει ως βάση μία εικόνα του λειτουργικού ubuntu. Η πρώτη αυτή εντολή (FROM) είναι υποχρεωτική για κάθε container καθώς δηλώνει την βασική εικόνα (base image) πάνω στην οποία θα στηθεί το container. Στη συνέχεια έχουμε μία εντολή RUN. Η εντολή RUN ουσιαστικά τρέχει την εντολή που της δίνουμε σαν να ήταν σε ένα bash shell προτάσσοντας την εντολή ”bin/bash -c”. Αυτή η εντολή RUN δημιουργεί έναν φάκελο, τον db_service, ο οποίος και θα φιλοξενεί τα αρχεία της βάσης και της υπηρεσίας γενικότερα. Η επόμενη εντολή, κάνει expose την θύρα 80 του container έτσι ώστε να είναι ορατή τόσο από το σύστημα οικοδεσπότη όσο και από

το δίκτυο. Τέλος, οι εντολές ENTRYPOINT και CMD αποτελούν το σημείο εκκίνησης της υπηρεσίας του εξυπηρετητή με κατάλληλες default παραμέτρους όταν ξεκινάει η λειτουργία του container.

Το Docker κατά την δημιουργία μιας εικόνας από ένα Dockerfile, για κάθε εντολή του αρχείου δημιουργεί και μία ενδιάμεση προσωρινή εικόνα στην μορφή μιας cache. Αυτό έχει δύο πλεονεκτήματα. Το πρώτο είναι ότι ακόμα και αν κάποια εντολή δεν εκτελέστηκε σωστά, η εικόνα έχει δημιουργηθεί μέχρι και την τελευταία επιτυχή εντολή και μπορεί να τρέξει ως αυτόνομο container. Με αυτόν τον τρόπο μπορούμε να τρέξουμε το container και να δούμε τι πήγε στραβά με την εντολή. Το δεύτερο είναι ότι, με αυτή την τεχνική του caching, σε περίπτωση που ξανακατασκευάσουμε μια εικόνα από την αρχή δεν θα εκτελεστούν όλες οι εντολές από την αρχή αλλά μόνο αυτές που δεν είχαν εκτελεστεί και δεν βρίσκονται στην cache [4].

Ας δούμε τώρα και την δημιουργία του container. Αρχικά πρέπει να δημιουργήσουμε την εικόνα μέσω του αρχείου που περιγράψαμε παραπάνω. Αυτό γίνεται με την εντολή: sudo docker build -t="db_docker" .

Η εντολή αυτή λέει στη δαίμονα διεργασία του Docker να δημιουργήσει μία νέα εικόνα από ένα αρχείο Dockerfile που βρίσκεται στον τρέχων φάκελο "." με όνομα "db_docker" (ετικέτα -t). Το αποτέλεσμα είναι το παρακάτω:

```
Sending build context to Docker daemon  631.3kB
Step 1/5 : FROM ubuntu:latest
 ---> f975c5035748
Step 2/5 : RUN mkdir /db_service
 ---> Using cache
 ---> f412fc744300
Step 3/5 : EXPOSE 80
 ---> Using cache
 ---> 65d272c02b5a
Step 4/5 : ENTRYPOINT /db_service/bin/db_server -p 80 -f /db_service/new.db
 ---> Using cache
 ---> da7c61df0691
Step 5/5 : CMD -s 4
 ---> Using cache
 ---> ba711c576022
Successfully built ba711c576022
Successfully tagged db_docker:latest
```

Σχήμα 31: Αποτέλεσμα κατασκευής εικόνας του container.

Στη συνέχεια εκτελούμε το container μέσω της εικόνας που φτιάξαμε πριν με την εντολή: sudo docker run -i -t -p 4444:80 -v $PWD:/db_service db_docker. Η εντολή αυτή λέει στη δαίμονα διεργασία του Docker να εκκινήσει ένα νέο container με βάση την εικόνα db_docker που δημιουργήσαμε προηγουμένως και κάποιες παραμέτρους. Η παράμετρος

-i (–interactive) μαζί με την παράμετρο -t (–tty) επιτρέπει στο χρήστη να αλληλεπιδρά με το container κατά τη λειτουργία του μέσα από την κονσόλα. Συγκεκριμένα, η πρώτη αφήνει ανοικτό το stdin στο container και η δεύτερη δημιουργεί ένα ψευδο-τερματικό για την αποστολή εντολών σε αυτό. Η παράμετρος -p (–publish) δεσμεύει θύρες μεταξύ του οικοδεσπότη και του container. Εδώ, η θύρα 4444 του οικοδεσπότη δεσμεύεται με την θύρα 80 του container. Η παράμετρος -v (–volume) είναι ίσως η πιο κομβική παράμετρος. Όπως έχουμε δει κάθε container αποτελείται από πολλά μόνο για ανάγνωση επίπεδα και ένα επίπεδο το οποίο προσφέρεται για εγγραφές. Αυτό το τελευταίο επίπεδο είναι που κρατάει όλες τις αλλαγές που κάνει ένα container κατά τη διάρκεια της ζωής του και όσο αυτό μεγαλώνει σε μέγεθος τόσο μειώνεται και η απόδοση του container. Εφόσον έχουμε να κάνουμε με δεδομένα μεγάλου μεγέθους και μακράς διαρκείας (persistent) θα πρέπει να αναζητήσουμε χώρο αποθήκευσης εκτός container, δηλαδή στον οικοδεσπότη όπου το container δεν έχει πρόσβαση. Η παράμετρος αυτή λοιπόν μας προσφέρει αυτή την επιλογή δεσμεύοντας έναν χώρο από το σύστημα αρχείων του οικοδεσπότη με έναν χώρο από το σύστημα αρχείων του container. Εδώ ο φάκελος που περιέχει τα αρχεία της βάσης και του εξυπηρετητή (η μεταβλητή $PWD που χρησιμοποιείται υποδηλώνει τον τρέχοντα κατάλογο εργασίας που υπό κανονικές συνθήκες είναι ο αρχικός κατάλογος της υπηρεσίας) δεσμεύεται με το φάκελο /db_service που δημιουργήσαμε εντός του container. Όλα τα αρχεία που περιέχει ο φάκελος του οικοδεσπότη είναι ορατά στο container και όλες οι αλλαγές σε αυτά αποθηκεύονται κατευθείαν στον οικοδεσπότη.

3.4 Πρόγραμμα Πελάτη

Πρόκειται για το τελικό επίπεδο της υπηρεσίας και αυτό που βλέπει ουσιαστικά ο χρήστης. Χρησιμοποιεί μία διεπαφή γραμμής εντολών (CLI - Command Line Interface) για την επικοινωνία με το χρήστη, ενώ χρησιμοποιεί μία υποδοχή για τη σύνδεση με τον εξυπηρετητή. Οι εντολές που μπορεί να δεχτεί το πρόγραμμα αυτό είναι οι εξής:

- **insert <key> <payload>.** Η εντολή αυτή εισάγει μία εγγραφή στη βάση με κλειδί <key> και φορτίο <payload>.
- **delete <key>.** Η εντολή αυτή διαγράφει την πρώτη εγγραφή με κλειδί <key> από τη βάση.
- **query <key> <operator>.** Η εντολή αυτή στέλνει ένα ερώτημα στη βάση για αναζήτηση εγγραφών με βάση κάποιο τελεστή πάνω σε ένα κλειδί <key>. Ο τελεστής αυτός μπορεί να είναι ένας εκ των LESS, LESS_EQUAL, GREATER, GREATER_EQUAL, EQUAL, NOT_EQUAL. Ως απάντηση λαμβάνεται μια λίστα με τα πεδία φορτία από τις εγγραφές που πληρούν τα κριτήρια του ερωτήματος.
- **exit.** Κλείνει τη σύνδεση με τον εξυπηρετητή και τερματίζει τη λειτουργία του προγράμματος.

4. ΑΞΙΟΛΟΓΗΣΗ ΥΠΗΡΕΣΙΑΣ

Σε αυτό το κεφάλαιο θα παρουσιάσουμε έναν έλεγχο απόδοσης της υπηρεσίας ευρετηρίου με βάση κάποια σενάρια αλληλεπίδρασης του χρήστη με τη βάση. Ο έλεγχος αυτός θα γίνει πάνω σε δύο άξονες: πρώτον σε διαφορετικά μεγέθη δεδομένων της βάσης και δεύτερον στον μέσο χρόνο απόκρισης της υπηρεσίας για διαφορετικό πλήθος συνδέσεων πελατών.

4.1 Δοκιμαστικό σενάριο ερωτημάτων αναζήτησης

Το δοκιμαστικό αυτό σενάριο έχει ως εξής: Αρχεία δεδομένων με 10000, 20000, 50000 και 100000 εγγραφές. Ένα πλήθος από clients στέλνουν ταυτόχρονα ανά δύο δευτερόλεπτα ερωτήματα αναζήτησης -100 ερωτήματα ο καθένας συνολικά- προς την δομή ευρετηρίου. Τα ερωτήματα αυτα αναφέρονται σε αναζήτηση με βάση τελεστές όπως αναφέρθηκε στο προηγούμενο κεφάλαιο. Αρχικά μετράται ο μέσος χρόνος απόκρισης του ευρετηρίου προς ένα πελάτη με τον τύπο $AVR_i = \frac{elapsed_time}{100}$, όπου elapsed_time είναι ο συνολικός χρόνος επεξεργασίας όλων των ερωτημάτων και 100 τα ερωτήματα, και μετά βρίσκεται ο συνολικός μέσος χρόνος απόκρισης με τον τύπο $AVR = \frac{\sum_{i=1}^{N} AVR_i}{N}$ όπου AVR_i είναι ο μέσος χρόνος απόκρισης για κάθε πελάτη όπως υπολογίστικε παραπάνω και N ο συνολικός αριθμός πελατών.

Τα αποτελέσματα είναι τα παρακάτω:

Πίνακας 1: Μέσος Χρόνος Απόκρισης Υπηρεσίας (AVR) σε ερωτήματα αναζήτησης

Record Tuples \ Clients	1	2	4	8	16
10000	0.06 sec	0.065 sec	0.0675 sec	0.0775 sec	0.0895 sec
20000	0.11 sec	0.115 sec	0.1275 sec	0.17125 sec	0.2245 sec
50000	0.15 sec	0.175 sec	0.21 sec	0.2375 sec	0.272 sec
100000	0.165 sec	0.192 sec	0.225 sec	0.25 sec	0.283 sec

4.2 Δοκιμαστικό σενάριο μεικτών ερωτημάτων

Το δοκιμαστικό αυτό σενάριο έχει ως εξής: Αρχεία δεδομένων με 10000, 20000, 50000 και 100000 εγγραφές. Ένα πλήθος από clients στέλνουν ταυτόχρονα ανά δύο δευτερόλεπτα ερωτήματα αναζήτησης ή ερωτήματα ενημέρωσης(εισαγωγή ή διαγραφή) -100 ερωτήματα ο καθένας συνολικά- προς την δομή ευρετηρίου με τυχαία σειρά. Οι μετρήσεις είναι οι ίδιες που χρησιμοποιήθηκαν στη προηγούμενη ενότητα.

Τα αποτελέσματα είναι τα παρακάτω:

Πίνακας 2: Μέσος Χρόνος Απόκρισης Υπηρεσίας (AVR) σε μεικτά φορτία ερωτημάτων

Clients / Record Tuples	1	2	4	8	16
10000	0.08 sec	0.085 sec	0.0875 sec	0.092 sec	0.117 sec
20000	0.12 sec	0.13 sec	0.135 sec	0.1735 sec	0.23 sec
50000	0.14 sec	0.205 sec	0.225 sec	0.243 sec	0.29 sec
100000	0.17 sec	0.264 sec	0.275 sec	0.29 sec	0.34 sec

Βλέποντας τα αποτελέσματα στους δύο παραπάνω πίνακες μπορούμε να εξάγουμε κάποια χρήσιμα συμπεράσματα. Αρχικά, παρατηρούμε ότι οι χρόνοι αυξάνονται αισθητά στο δεύτερο σενάριο σε σχέση με το πρώτο, πράγμα αναμενόμενο καθώς έχουμε και ερωτήματα ενημέρωσης της βάσης που είναι περισσότερο χρονοβόρα από ερωτήματα απλής αναζήτησης δεδομένων. Έπειτα, παρατηρούμε ότι οι χρονικές διαφορές μεταξύ των διαδοχικών αριθμών πελατών είναι πολύ μικρότερες μεταξύ των βάσεων διαδοχικών μεγεθών. Επίσης, παρατηρούμε ότι οι χρονικές τιμές μεταξύ δύο διαδοχικών αριθμών πελατών στην ίδια βάση δεν έχουν ανάλογη αύξηση με τον αριθμό πελατών. Για παράδειγμα, η χρονική τιμή των 8 πελατών για τη βάση των 20000 εγγραφών δεν είναι η διπλάσια της τιμής των 4 εγγραφών αλλά πολύ μικρότερη. Αυτό μας δείχνει ότι η η δομή του B^{link}-Δέντρου προσφέρει υψηλή και αρκετά ικανοποιητική κλιμάκωση με ρυθμό αύξησης αρκετά μικρότερο του ρυθμού αύξησης του αριθμού πελατών. Τέλος, παρατηρούμε ότι οι χρονικές τιμές για ίδιο αριθμό πελατών μεταξύ δύο διαδοχικών βάσεων είναι σε αρκετές περιπτώσεις μικρότερες της άυξησης μεγέθους των βάσεων. Αυτό φανερώνει την υψηλή κλιμάκωση που μπορεί να προσφέρει το B^{link}-Δέντρο και σε σχέση με το πλήθος των δεδομένων που φιλοξενεί.

5. ΣΥΜΠΕΡΑΣΜΑΤΑ

Στην εργασία αυτή δημιουργήσαμε μία υπηρεσία ευρετηρίου ΒΔ ενσωματωμένη μέσα σε ένα container. Δημιουργήσαμε ένα αφαιρετικό σύστημα -ένα μαύρο κουτί στην ανεπίσημη ορολογία της πληροφορικής- το οποίο αποτελείται από διακριτά και ανεξάρτητα κομμάτια. Κάθε κομμάτι είναι ξεχωριστό και προσφέρει κάτι δικό του στην υπηρεσία.

Το πρώτο κομμάτι αφορά στη δομή ευρετηρίου ΒΔ. Είδαμε ότι η δομή του B^{link}-Δέντρου προσφέρει ένα μηχανισμό υψηλής κλιμάκωσης μέσω των κλειδωνιών και κλειδαριών. Επιτρέπει σε πολλούς χρήστες να χρησιμοποιούν τη βάση ταυτόχρονα και όπως είδαμε ο ρυθμός αύξησης του χρόνου εξυπηρέτησης μιας αίτησης είναι μικρότερος από τον ρυθμό αύξησης των χρηστών που επιχειρούν πάνω στη βάση.

Το δεύτερο κομμάτι αφορά στον εξυπηρετητή. Είδαμε ότι ο εξυπηρετητής αποτελείται από νήματα τα οποία εξυπηρετούν έναν χρήστη τη φορά. Αυτό μπορεί να μην εξυπηρετεί τόσο στην κλιμάκωση της υπηρεσίας αλλά δημιουργεί την ψευδαίσθηση στον χρήστη ότι η εξυπηρέτηση του είναι συνεχής και αδιάκοπη και δεν εξαρτάται από το πλήθος ή τη σειρά των ερωτημάτων που στέλνονται από το σύνολο των ενεργών χρηστών της υπηρεσίας.

Το τελευταίο κομμάτι αφορά στο container που περικλείει την υπηρεσία. Αναλύσαμε την τεχνολογία αυτή κυρίως σε θεωρητικό επίπεδο. Είδαμε τη βασική ιδέα των containers που είναι η απομόνωση εφαρμογών μέσω των cgroups και του namespace isolation. Επίσης, είδαμε την δομή και την κατασκεύη ενός container μέσω της πλατφόρμας του Docker όπου κάθε container δημιουργείται από μία εικόνα και κάθε εικόνα αποτελείται από πολλά διαφορετικά συστήματα αρχείων. Τέλος, είδαμε ότι αυτό που ξεχωρίζει τις εικονικές μηχανές από τα containers είναι ότι τα containers χρησιμοποιούν τον πυρήνα του υπάρχοντος λογισμικού και δεν προσομοιώνουν το εγγενές υλικό αλλά το χρησιμοποιούν όπως είναι σε μερίδια πόρων.

Το γεγονός ότι κάθε κομμάτι είναι ανεξάρτητο καθιστά την υπηρεσία εύκολα τροποποιήσιμη ανάλογα με τις ανάγκες και το περιβάλλον. Για παράδειγμα, η δομή ευρετηρίου θα μπορούσε να είναι οποιαδήποτε άλλη γνωστή δομή ευρετηρίου (π.χ. κατακερματισμός) η οποία διαθέτει μια τεχνική ταυτόχρονης εξυπηρέτησης χρηστών. Επίσης, ο εξυπηρετητής θα μπορούσε να φτιαχτεί έτσι ώστε κάθε να εξυπηρετεί καθ’ αίτηση και όχι κατά χρήστη. Τέλος, για το επίπεδο του container θα μπορούσε να χρησιμοποιηθεί κάποια άλλη πλατφόρμα (π.χ. LXC) αντί για το Docker με παρόμοια αποτελέσματα.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος Όρος	**Ελληνικός Όρος**
Virtualization	Εικονικοποίηση
Container	Περιέκτης
Server-Client	Πελάτη-Εξυπηρετητή
Index	Ευρετήριο
Host	Οικοδεσπότης
Guest	Επισκέπτης
Daemon process	Δαίμονας διεργασία
Record	Εγγραφή
Field	Πεδίο
User space	Χώρος χρήστη
File System	Σύστημα αρχείων
Socket	Υποδοχή
Port	Θύρα
Latch	Κλειδωνιά
Lock	Κλειδαριά

ΣΥΝΤΜΗΣΕΙΣ, ΑΡΚΤΙΚΟΛΕΞΑ ΚΑΙ ΑΚΡΩΝΥΜΙΑ

RAM	Random Access Memory
ΒΔ	Βάση(εις) Δεδομένων
VMM	Virtual Machine Manager
ΛΣ	Λειτουργικό Σύστημα
ΚΜΕ	Κεντρική Μονάδα Επεξεργασίας
CLI	Command Line Interface
ΕΚΠΑ	Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΑΝΑΦΟΡΕΣ

[1] R. Elmasri and S.B. Navathe, 2012, «Θεμελιώδεις Αρχές Συστημάτων Βάσεων Δεδομένων», 6, Εκδόσεις Δίαυλος, Αθήνα, pp 520-543.

[2] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, 2014, «Λειτουργικά Συστήματα», 9, Εκδόσεις Γκιούρδας, Αθήνα, pp 711-728.

[3] Marc Rochkind, 2007, «Προγραμματισμός σε UNIX», 2, Εκδόσεις Κλειδάριθμος, Αθήνα.

[4] James Turnbull, 2014, «The Docker Book», Docker Inc.

[5] David Lomet, 2004, «Simple, Robust and Highly Concurrent B-trees with Node Deletion», Έρευνα αναφοράς, Microsoft Research, Microsoft, Redmond WA.

[6] Chenxi Wang, 29 Jun. 2017, «What is Docker? Linux containers explained», https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html. [Προσπελάστηκε 7/12/17]

[7] Wikipedia, 2013, «Operating-system-level virtualization», https://en.wikipedia.org/wiki/Operating-system-level_virtualization. [Προσπελάστηκε 10/12/17]

[8] Wikipedia, 2013, «Virtualization», https://en.wikipedia.org/wiki/Virtualization. [Προσπελάστηκε 10/12/17]

[9] Wikipedia, 2013, «LXC», https://en.wikipedia.org/wiki/LXC. [Προσπελάστηκε 10/12/17]

[10] Wikipedia, 2013, «Linux Namespaces», https://en.wikipedia.org/wiki/Linux_namespaces. [Προσπελάστηκε 10/12/17]

[11] Red Hat Inc., 2018, «What is virtualization», https://www.redhat.com/en/topics/virtualization/what-is-virtualization. [Προσπελάστηκε 12/01/18]

[12] Red Hat Inc., 2018, «What are Linux containers?», https://opensource.com/resources/what-are-linux-containers. [Προσπελάστηκε 19/01/18]

[13] Red Hat Inc., 2018, «What's a Linux container?», https://www.redhat.com/en/topics/containers/whats-a-linux-container#. [Προσπελάστηκε 19/01/18]

[14] Jack Wallen, 23 May 2017, «Containers vs. virtual machines: A simplified answer to a complex question», https://www.techrepublic.com/article/containers-vs-virtual-machines-a-simplified-answer-to-a-complex-question/. [Προσπελάστηκε 08/02/18]

[15] Imesh Guanaratne, 3 Sep 2016, «Evolution of Linux Containers and Future», https://medium.com/containermind/evolution-of-linux-containers-and-future-6f2adc1d5086. [Προσπελάστηκε 05/03/18]