ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΚΑΤΑΝΕΜΗΜΕΝΗ ΕΠΕΞΕΡΓΑΣΙΑ ΜΕΣΩ NODE.JS ΚΑΙ WEB WORKERS

Διπλωματική Εργασία

του

Παύλου Χρυσοχοΐδη

Θεσσαλονίκη, Αύγουστος 2018

ΚΑΤΑΝΕΜΗΜΕΝΗ ΕΠΕΞΕΡΓΑΣΙΑ ΜΕΣΩ NODE.JS ΚΑΙ WEB WORKERS

Παύλος Χρυσοχοΐδης

ΠΤΥΧΙΟ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ, ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ, 2013

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κασκάλης Θεόδωρος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την / /

Κασκάλης Θεόδωρος	Μαργαρίτης Κωνσταντίνος	Χατζηγεωργίου Αλέξανδρος
.....	.....	.....

Παύλος Χρυσοχοΐδης
.....

Περίληψη

Η επεξεργαστική ισχύς είναι πια άφθονη στην εποχή μας, λόγω της ραγδαίας εξέλιξης του τομέα της πληροφορικής, καθώς και της ευρείας εξάπλωσης των υπολογιστών, με την μορφή διαφόρων συσκευών όπως σταθεροί και φορητοί υπολογιστές, έξυπνα κινητά τηλέφωνα, tablets αλλά και παιχνιδομηχανές. Ταυτόχρονα με την ανάπτυξη του τομέα της πληροφορικής, αξιοσημείωτη εξέλιξη έχουν παρουσιάσει και οι τεχνολογίες ανάπτυξης διαδικτυακών εφαρμογών. Πράγμα που έχει ως αποτέλεσμα οι εφαρμογές τέτοιου είδους να κατέχουν την μερίδα του λέοντος παγκοσμίως και να παρουσιάζουν αυξανόμενη τάση.

Στην παρούσα εργασία με βάση τις παραπάνω παρατηρήσεις γίνεται η ανάπτυξη μιας εφαρμογής που επιτρέπει την κατανεμημένη επεξεργασία χρησιμοποιώντας τεχνολογίες διαδικτύου. Η χρήση αυτών των τεχνολογιών επιλέγεται ώστε να διευκολυνθούν οι προγραμματιστές στην επίλυση των προβλημάτων που τους ενδιαφέρουν γράφοντας κώδικα στην γλώσσα προγραμματισμού JavaScript. Επίσης αυτή η επιλογή έγινε και για τον λόγο ότι αυτές οι εφαρμογές μπορούν να εκτελεστούν σε όλες σχεδόν τις συσκευές που υποστηρίζουν περιηγητές ιστού.

Μετά την ανάπτυξη και παρουσίαση της εφαρμογής γίνεται η δοκιμαστική εκτέλεση δύο προβλημάτων. Του υπολογισμού του π με χρήση της μεθόδου Monte Carlo και την μέτρηση λέξεων σε αρχεία μεγάλου όγκου Word Count. Οι δοκιμές αυτές ήταν αρκετά ικανοποιητικές και έδειξαν ότι η χρήση αυτών το τεχνολογιών, συμπεριλαμβανομένου των Node.js και Web Workers μπορεί να προσφέρει λύσεις στον τομέα της κατανεμημένης επεξεργασίας αλλά παρουσιάζουν όμως πολλές και διάφορες προκλήσεις.

Λέξεις Κλειδιά: Κατανεμημένη Επεξεργασία, Node.js, Web Workers, JavaScript, Εθελοντική Προσφορά Επεξεργαστικής Ισχύς

Abstract

Nowadays the computing power is plentiful, due to the rapid development of the IT sector and the widespread usage of computers in various types of devices such as desktops, laptops, smartphones, tablets, and gaming machines. At the same time, the development of the technologies used to create Internet applications has also been remarkable. As a result, such applications have a big part of the global application development and tend to rise.

For the present thesis, based on the above observations, an application has been developed that allows distributed processing using web technologies. The use of previously mentioned technologies is chosen in order to help developers solve a collection of problems that they are interested in by writing their source code in JavaScript. This choice was also made because these apps can run on almost all the devices that support web browsers.

After the development and presentation of the application, the application was tested on two problems. The first one, is the calculation of π using the Monte Carlo method and the second one is the Word Count of a large file. These tests were quite satisfactory and showed that the use of these technologies, such as Node.js and Web Workers, can offer solutions in the field of distributed processing, but they also present many different challenges.

Keywords: Distributed Computing, Node.js, Web Workers, JavaScript, Volunteer Computing

Ευχαριστίες

Με την ολοκλήρωση της παρούσας διπλωματικής εργασίας θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Κασκάλη Θεόδωρο, αναπληρωτή καθηγητή του τμήματος Εφαρμοσμένης Πληροφορικής του πανεπιστημίου Μακεδονίας, για την υπομονή του, την καθοδήγηση και την βοήθεια που μου προσέφερε κατά την διάρκεια εκπόνησης της εργασίας.

Θα ήθελα επίσης να ευχαριστήσω την οικογένεια και τους φίλους μου για την στήριξη, την ενθάρρυνση και την βοήθεια τους.

Περιεχόμενα

Κατάλογος Εικόνων

1 Εισαγωγή

Η ραγδαία εξέλιξη της τεχνολογίας ιδιαίτερα στον τομέα των υπολογιστών έχει καταστήσει την επεξεργαστική ισχύ σχετικά φθηνή και εύκολα προσβάσιμη στην καθημερινή ζωή. Αυτό έχει βοηθήσει πολύ στην εξάπλωση της με αποτέλεσμα να υπάρχει πια άφθονη επεξεργαστική δύναμη. Για παράδειγμα ένα κινητό/ταμπλέτα νέας γενιάς έχει πολύ περισσότερη επεξεργαστική ισχύ από τους πρώτους προσωπικού υπολογιστές.

Αυτή την ισχύ άρχισε να χρησιμοποιεί η επιστημονική κοινότητα, ήδη από το 1996 με την εμφάνιση του The Great Internet Mersenne Prime Search (GIMPS) μέσα από την ιδέα της δωρεάς επεξεργαστικής ισχύς, ώστε να κάνει εφικτή την λύση διαφόρων επιστημονικών προβλημάτων [6]. Η ιδέα αυτή βασίζεται στην εθελοντική προσφορά των υπολογιστών των χρηστών αλλά και στο ότι το πρόβλημα το οποίο λύνουν είναι δυνατό να παραλληλοποιηθεί, ώστε να λυθεί σε συστήματα κατανεμημένης επεξεργασίας.

Ένα τέτοιο σύστημα είναι το Berkeley Open Infrastructure for Network Computing (BOINC) [10] το οποίο προσφέρει την δυνατότητα στους χρήστες του είτε να συμμετέχουν στην επίλυση κάποιου προβλήματος, προσφέροντας την επεξεργαστική ισχύ του προσωπικού τους υπολογιστή (πλέον το σύστημα υποστηρίζει και έξυπνες συσκευές), είτε να καταχωρήσουν το δικό τους πρόβλημα και να επωφεληθούν από την συνεισφορά των άλλων. Το BOINC παρέχει ένα επίπεδο αφαίρεσης στον χρήστη/προγραμματιστή ώστε να τον διευκολύνει και να μπορεί να επικεντρωθεί στην λύση του προβλήματος χωρίς να χρειάζεται να λάβει υπόψη του την διαχείριση του κατανεμημένου συστήματος.

1.1 Σκοπός – Στόχοι

Τα παραπάνω συστήματα είναι αρκετά διαδεδομένα και έχουν δώσει λύσεις σε διάφορα προβλήματα τα οποία παρουσιάζονται στο επόμενο κεφάλαιο. Η παρούσα εργασία προσπαθεί να εμπνευστεί από αυτά και να μεταφέρει κάποια από την λειτουργικότητα τους σε ένα σύστημα το οποίο έχει υλοποιηθεί σε τεχνολογίες διαδικτύου (web), Node.js για τον διακομιστή (server), προγράμματα περιήγησης (browsers) για την διεπαφή του χρήστη όπου εκεί θα γίνονται και οι σχετικοί υπολογισμοί χρησιμοποιώντας Web Workers.

Στόχος της εργασίας είναι η ανάπτυξη μιας βιβλιοθήκης/framework η οποία θα επιτρέπει σε οποιονδήποτε θέλει να λύσει κάποιο πρόβλημα που απαιτεί μεγάλη υπολογιστική ισχύ, εύκολα να προγραμματίσει την λύση (σε JavaScript) αλλά και ακόμη πιο εύκολα οι χρήστες να μπορούν να συμμετέχουν στους υπολογισμούς της λύσης χωρίς να απαιτείται η εγκατάσταση κάποιου επιπλέον προγράμματος στον προσωπικό τους υπολογιστή (η σε από οποιαδήποτε άλλη συσκευή υποστηρίζει περιηγητή ιστού). Το μόνο που θα πρέπει να κάνουν είναι να επισκεφθούν χρησιμοποιώντας ένα σύνδεσμο την ιστοσελίδα του προβλήματος.

1.2 Συνεισφορά

Για την ανάπτυξη της εφαρμογής έγινε αρχικά βιβλιογραφική έρευνα σχετικά με θέματα κατανεμημένης επεξεργασίας, το είδος και την κατηγορία των προβλημάτων που μπορούν να κατανεμηθούν. Τέλος, πραγματοποιήθηκε μελέτη παρόμοιων εφαρμογών και τεχνολογιών οι οποίες μπορούν να χρησιμοποιηθούν.

Χρησιμοποιώντας το framework που αναπτύχθηκε, υλοποιήθηκαν διάφοροι αλγόριθμοι οι οποίοι λύνουν γνωστά προβλήματα και τα αποτελέσματα παρατίθενται σε επόμενο κεφάλαιο, όπως και η βιβλιογραφική έρευνα, ώστε να μπορεί κάποιος μελλοντικός αναγνώστης να συγκρίνει τα αποτελέσματα αυτών των αλγορίθμων με υλοποιήσεις σε άλλα συστήματα.

1.3 Διάρθρωση της μελέτης

Η σχετική έρευνα με το αντικείμενο της διπλωματικής παρουσιάζεται στο Κεφάλαιο 2.

Το Κεφάλαιο 3 περιγράφει τις προδιαγραφές, την αρχιτεκτονική και γενικά πως λειτουργεί η εφαρμογή.

Στο Κεφάλαιο 4 υλοποιούνται διάφοροι αλγόριθμοι που λύνουν γνωστά προβλήματα και παρατίθενται τα αποτελέσματα τους.

Τέλος, στο 5^{ο} και τελευταίο κεφάλαιο γίνεται η σύνοψη των συμπερασμάτων της έρευνας καθώς και αναφέρονται προτάσεις για μελλοντικές επεκτάσεις της.

2 Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο γίνεται η καταγραφή των εννοιών, των τεχνολογιών και άλλων παρόμοιων εφαρμογών στα οποία στηρίχθηκε η ανάπτυξη της παρούσας διπλωματικής.

2.1 Κατανεμημένα συστήματα

Με τον όρο κατανεμημένα συστήματα περιγράφονται τα συστήματα εκείνα των οποίων τα συστατικά μέρη βρίσκονται σε ένα σύνολο υπολογιστών συνδεδεμένων μεταξύ τους μέσω κάποιου δικτύου επικοινωνίας, οι οποίοι επικοινωνούν και συντονίζονται μόνο με ανταλλαγή μηνυμάτων. Από τον όρο αυτό προκύπτουν τα επακόλουθα για ένα κατανεμημένο σύστημα [1]:

Συγχρονισμός (Concurrency): Η εκτέλεση του προγράμματος σε ένα κατανεμημένο σύστημα γίνεται ταυτόχρονα σε όλους τους υπολογιστές από τους οποίους αποτελείται. Το σύστημα θα πρέπει να έχει την δυνατότητα να διαχειριστεί κοινόχρηστους πόρους και να αυξήσει την δυνατότητα αυτή προσθέτοντας και άλλους υπολογιστικούς πόρους. Όλα αυτά εξασφαλίζοντας τον συντονισμό μεταξύ των μερών του συστήματος.

Έλλειψη κοινού ρολογιού (No global clock): Ο συντονισμός μεταξύ συστημάτων συχνά εξαρτάται από την έννοια του χρόνου και συγκεκριμένα από την ύπαρξη κοινού ρολογιού μεταξύ των μερών. Σε ένα κατανεμημένο σύστημα όμως όπου η επικοινωνία επιτυγχάνεται μόνο από την ανταλλαγή μηνυμάτων στο δίκτυο η χρήση ενός κοινού ρολογιού δεν είναι εφικτή λόγω των περιορισμών του δικτύου στον συντονισμό των χρόνων των επιμέρους συστημάτων.

Ανεξάρτητες αποτυχίες (Independent failures): Στον κόσμο των κατανεμημένων συστημάτων τα οποία αποτελούνται από πολλά μέλη οι αποτυχίες πρέπει να θεωρούνται ως ο κανόνας λειτουργίας τους. Αυτές οι αποτυχίες μπορεί να είναι κάποια βλάβη στο δίκτυο, η οποία απομονώνει ένα κομμάτι του συστήματος ή μία άλλη περίπτωση θα ήταν μία βλάβη σε ένα υπολογιστή ή ακόμη ένας απρόσμενος τερματισμός του προγράμματος. Ιδανικά για όλες αυτές τις

περιπτώσεις θα θέλαμε να μην υπήρχε καμία συνέπεια ώστε να υπάρχει ομαλή λειτουργία του υπόλοιπου συστήματος ως σύνολο.

2.1.1 Χαρακτηριστικά - προκλήσεις κατανεμημένων συστημάτων

Από την χρήση των κατανεμημένων συστημάτων προκύπτουν κάποια κοινά χαρακτηριστικά τα οποία αποτελούν συνήθως προκλήσεις για όλα αυτά τα συστήματα. Αυτά είναι τα εξής [1][3]:

Ετερογένεια: Τα κατανεμημένα συστήματα αποτελούνται από πολλά και διαφορετικά δίκτυα, υπολογιστές, λειτουργικά συστήματα, γλώσσες προγραμματισμού αλλά και υλοποιήσεις από διαφορετικούς προγραμματιστές

Ανοιχτότητα (Openness): Αυτό το χαρακτηριστικό είναι αυτό που καθορίζει το κατά πόσο ένα σύστημα είναι επεκτάσιμο. Ο βαθμός με τον οποίο είναι δυνατόν να προστίθενται και γίνονται διαθέσιμοι νέοι πόροι στο σύστημα είναι αυτός που χαρακτηρίζει την ανοιχτότητα του συστήματος.

Ασφάλεια: Θα πρέπει να εξασφαλίζεται η εμπιστευτικότητα, η πρόσβαση δηλαδή μόνο από εξουσιοδοτημένους χρήστες, η ακεραιότητα των δεδομένων ή/και μηνυμάτων

και να μην είναι δυνατή η μη εξουσιοδοτημένη μεταβολή ή αλλοίωση τους. Τέλος, θα πρέπει να διασφαλίζεται η διαθεσιμότητα των υπηρεσιών του συστήματος.

Επεκτασιμότητα: Ένα άλλο χαρακτηριστικό των κατανεμημένων συστημάτων είναι η αποτελεσματική λειτουργία τους σε οποιαδηποτε κλίμακα και με ένα βέλτιστο τρόπο ως προς το κόστος.

Αντιμετώπιση βλαβών: Καθώς οι βλάβες σε αυτά τα συστήματα είναι πολύ συχνές και είναι κάτι που σίγουρα θα συμβεί, πρέπει τα μέρη τους να είναι έτσι προγραματισμένα ώστε να μπορούν να τις διαγνώσουν και να τις αντιμετωπίσουν.

Συγχρονισμός (Concurrency): Η ταυτόχρονη εκτέλεση ή η πρόσβαση στους κοινόχρηστους πόρους του συστήματος δεν θα πρέπει να επηρεάζεται από τον αριθμό των χρηστών και τα δεδομένα θα πρέπει να είναι πάντα σε συνεπή κατάσταση.

Διαφάνεια: Η διαφάνεια έχει σχέση με το πως οι χρήστες αλλά και οι προγραμματιστές των εφαρμογών βλέπουν το σύστημα (ενιαίο). Αναλαμβάνουν δηλαδή να χειριστούν διάφορες κοινές λειτουργίες επιτρέποντας τους προγραμματιστές να επικεντρωθούν στην ανάπτυξη της εφαρμογής που τους ενδιαφέρει.

Ποιότητα υπηρεσιών: Ένα κατανεμημένο σύστημα όχι μόνο πρέπει να κάνει διαθέσιμες τις υπηρεσίες που παρέχει αλλά πρέπει και να παρέχει εγγυήσεις για την ποιότητα σχετικά με την απόδοση, την ασφάλεια και την αξιοπιστία του.

2.2 Παραδείγματα χρήσης κατανεμημένων συστημάτων

Τα κατανεμημένα συστήματα, σήμερα βρίσκονται σχεδόν σε όλες τις διαδικτυακές υπηρεσίες που χρησιμοποιούμε στην καθημερινότητα μας [1]. Όπως για παράδειγμα το διαδίκτυο, την αναζήτηση στο διαδίκτυο, τα διαδικτυακά παιχνίδια, το email, τα

κοινωνικά δίκτυα, eCommerce, κλπ. Κάποιες από τις εφαρμογές τους περιγράφονται παρακάτω.

2.2.1 Αναζήτηση στο διαδίκτυο

Με την ραγδαία ανάπτυξη του διαδικτύου τα τελευταία χρόνια η αναζήτηση σε αυτό έχει εκτοξευθεί σε τεράστια μεγέθη. Επίσης έχει μεγαλώσει κατά πολύ τις σελίδες από τις οποίες αποτελείται, σε μέγεθος δισεκατομμυρίων και τις μοναδικές διευθύνσεις σε βαθμό τρισεκατομμυρίων. Αυτό έχει δυσκολέψει αρκετά την δουλειά των μηχανών αναζήτησης που πρέπει να κατατάξουν σε ευρετήρια όλο το περιεχόμενο του Παγκόσμιου Ιστού. Δεδομένου ότι οι μηχανές αυτές εκτελούν εξελιγμένη επεξεργασία πάνω σε αυτή την τεράστια βάση δεδομένων έχουν αποτελέσει μεγάλη πρόκληση για τη σχεδίαση των κατανεμημένων συστημάτων.

Η Google είναι μία από τις πρωτοπόρες εταιρείες στον τομέα αυτό και έχει κάνει μεγάλη προσπάθεια στην σχεδίαση και υλοποίηση ενός κατανεμημένου συστήματος για την υποστήριξη της αναζήτησης, το οποίο είναι ένα από τα μεγαλύτερα και πιο περίπλοκα στην ιστορία. Τα βασικότερα συστατικά μιας τέτοιας υποδομής είναι τα εξής [1]:

- μεγάλος αριθμός συνδεδεμένων υπολογιστών οι οποίοι βρίσκονται σε κέντρα δεδομένων (data centers) σε όλο τον κόσμο
- κατανεμημένο σύστημα διαχείρισης αρχείων ικανό να αντέξει τα μεγάλα μεγέθη και την ανάγκη για συνεχόμενη ανάγνωση
- ένα προγραμματιστικό μοντέλο το οποίο μπορεί να υποστηρίξει την διαχείριση πολύ μεγάλων παράλληλων και κατανεμημένων υπολογισμών (MapReduce το οποίο αναλύεται παρακάτω)

2.2.2 Εθελοντική προσφορά υπολογιστικής ισχύος

Η εθελοντική προσφορά υπολογιστικής ισχύος (Volunteer computing) είναι μια ιδέα με βάση την οποία χρησιμοποιείται ένα μεγάλο δίκτυο συνδεδεμένων υπολογιστών ως ένα μεγάλο παράλληλο σύστημα επεξεργασίας [5]. Χρήστες οι οποίοι έχουν διαθέσιμη επεξεργαστική ισχύ την προσφέρουν σε αυτό το μεγάλο δίκτυο προσθέτοντας τον υπολογιστή τους σε αυτό. Συνήθως αυτή η διαδικασία περιλαμβάνει την εγκατάσταση

κάποιας εφαρμογής στον υπολογιστή τους, η οποία αναλαμβάνει την επικοινωνία με το υπόλοιπο δίκτυο καθώς και την εκτέλεση των απαραίτητων υπολογισμών.

Τα προβλήματα τα οποία μπορούν να λυθούν σε τέτοιου είδους συστήματα θα πρέπει να είναι δυνατόν να παραλληλοποιηθούν. Με άλλα λόγια, θα πρέπει να μπορούν να χωριστούν σε πολλά μικρά κομμάτια εργασιών τα οποία όμως είναι όσο το δυνατόν πιο ανεξάρτητα μεταξύ τους. Κάποια από αυτά μπορεί να είναι η εύρεση του επόμενου μεγάλου πρώτου αριθμού, η αποκρυπτογράφηση μηνυμάτων ακόμη και η εύρεση ζωής στο διάστημα! Βέβαια λόγω της φύσης της εθελοντικής προσφοράς τα προβλήματα αυτά θα πρέπει να είναι και τέτοια ώστε να παροτρύνουν κάποιον χρήστη να προσφέρει την υπολογιστική ισχύ της συσκευής του.

Υπάρχουν διάφορα τέτοια συστήματα τα οποία έχουν αναπτυχθεί από την επιστημονική κοινότητα, τα οποία είτε λειτουργούν για την λύση κάποιου συγκεκριμένου προβλήματος είτε προσφέρουν ένα περιβάλλον ώστε να κάνουν πιο εύκολη την προσθήκη και επίλυση διαφόρων προβλημάτων. Αυτά τα συστήματα είναι αρκετά επιτυχημένα κάτι που δείχνει ότι ο κόσμος ενδιαφέρεται και συμβάλλει στην επιστημονική πρόοδο. Κάποια από τα πιο ευρέως διαδεδομένα συστήματα περιγράφονται παρακάτω.

2.2.2.1 Great Internet Mersenne Prime Search

Το GIMPS εμφανίστηκε το 1996 με σκοπό την εύρεση μεγάλων πρώτων αριθμών [6]. Στην αρχική του έκδοση χρειαζόταν αρκετή προσπάθεια από τους χρήστες που συμμετείχαν καθώς η κατανομή των εργασιών για κάθε υπολογιστή αλλά και η συγκέντρωση των αποτελεσμάτων γινόταν μέσω ηλεκτρονικού ταχυδρομείου. Με την πάροδο του χρόνου η διαδικασία βελτιώθηκε και πλέον είναι αρκετή μόνο η εγκατάσταση ενός προγράμματος πελάτη στον υπολογιστή το οποίο αναλαμβάνει να κάνει τα πάντα αυτό. Το τελευταίο επίτευγμα του δικτύου του GIMPS ήταν το Δεκέμβριο του 2017 στις 26 του μήνα όπου βρήκε τον 50ο γνωστό μεγάλο πρώτο αριθμό.

2.2.2.2 Distebuted.net

Το distributred.net είναι κι αυτό ένα παρόμοιο σύστημα με το GIMPS το οποίο εμφανίστηκε το 1997 με σκοπό την επίλυση της πρόκλησης του RSA RC-56, να “σπάσει”

δηλαδή το μυστικό κλειδί των 56-bit [7][8]. Το δίκτυο συνεχίζει ακόμη και σήμερα προσπαθώντας να λύσει και άλλα τέτοιου είδους κρυπογραφικά προβλήματα.

Για τη συμμετοχή και σε αυτό το σύστημα ο χρήστης πρέπει να εγκαταστήσει στον υπολογιστή του ένα πρόγραμμα πελάτη, το οποίο χρησιμοποιεί την CPU όταν αυτή βρίσκεται σε κατάσταση αδράνειας.

2.2.2.3 Berkeley Open Infrastructure for Network Computing

Το BOINC είναι και αυτό ένα σύστημα εθελοντικής προσφοράς υπολογιστικής ισχύος σαν τα παραπανω με τη διαφορά ότι αυτό είναι μία πλατφόρμα η οποία επιτρέπει την προσθήκη και επίλυση διαφορετικών προβλημάτων τα οποία χρησιμοποιούν το δίκτυο του [9].

Αρχικά, δημιουργήθηκε για να υποστηρίξει το έργο SETI@home, το οποίο ψάχνει ζωή στο διάστημα αναλύοντας σήματα που μας έρχονται από αυτό. Αργότερα, τον Απρίλιο του 2002 εξελίχθηκε σε μία πλατφόρμα που μπορεί να υποστηρίξει και άλλα τέτοιου είδους προβλήματα. Είναι μία πλατφόρμα ανοιχτού κώδικα και έχει προσφέρει πολύ στην εξέλιξη της εθελοντικής προσφοράς CPU, λύνοντας και θέματα όπως την εσκεμένης παραπληροφόρησης του συστήματος με λανθασμένα αποτελέσματα. Αυτό το έκανε εφικτό με την εισαγωγή της βαθμολόγησης του κάθε υπολογιστή με βάση την αξιολόγηση και επαλήθευση των αποτελεσμάτων που στέλνει στο δίκτυο [10].

Το BOINC και αυτό στηρίζεται στην εγκατάσταση ενός προγράμματος στον υπολογιστή του χρήστη που θέλει να συμμετέχει στο δίκτυο, επιτρέποντας το να επικοινωνεί με το υπόλοιπο δίκτυο αλλά και να εκτελεί τους απαραίτητους υπολογισμούς.

2.2.2.4 Το Bayanihan σύστημα εθελοντικής προσφοράς υπολογιστικής ισχύος

Το Bayanihan [5] είναι και αυτό μία πλατφόρμα όπως το BOINC με την διαφορά όμως ότι ο χρήστης ο οποίος θέλει να συμμετάσχει στο δίκτυο και να προσφέρει την υπολογιστική δύναμη του υπολογιστή του δεν χρειάζεται πια να εγκαταστήσει κάποιο πρόγραμμα αλλά το μόνο που πρέπει να κάνει είναι να επισκεφτεί μία ιστοσελίδα. Αυτό γίνεται δυνατό γιατί χρησιμοποιούνται τεχνολογίες web και συγκεκριμένα Java Applets.

Το μοντέλο που χρησιμοποιείται από το bayanihan είναι το master-worker, δηλαδή η ιδέα ότι στο δίκτυο υπάρχουν οι συνδεδεμένοι υπολογιστές των χρηστών οι οποίοι είναι οι workers που αναλαμβάνουν την επίλυση των εργασιών και των διακομιστών που αναλαμβάνουν τον συντονισμό των workers αλλά και τη διαχείριση των εργασιών και τα αποτελέσματα τους.

Βέβαια η χρήση των java applets δεν είναι πλέον τόσο διαδεδομένη αλλά το σύστημα αυτό έχει πολλά χαρακτηριστικά στα οποία στηρίχθηκε και η ανάπτυξη της εφαρμογής της παρούσας εργασίας.

2.3 Αρχιτεκτονικές συστημάτων κατανεμημένης επεξεργασίας

Τα κατανεμημένα συστήματα με βάση την υποδομή την οποία χρησιμοποιούν και το τρόπο με τον οποίο οι συνιστώσες τους επικοινωνούν μεταξύ τους μπορούν να κατηγοριοποιηθούν στις παρακάτω αρχιτεκτονικές [2][3].

Πελάτης-Διακομιστής (Client-Server): Το μοντέλο σχεδίασης πελάτη-διακομιστή χωρίζει την επεξεργασία μεταξύ δύο μερών τον πελάτη και τον διακομιστή. Ο διακομιστής έχει στη διάθεση του πόρους, τους οποίους κάνει διαθέσιμους στους πελάτες μέσω των υπηρεσιών. Ο πελάτης στέλνει μήνυμα/αίτηση στον διακομιστή και περιμένει την απάντηση του με τους πόρους/δεδομένα τα οποία ζήτησε ο πελάτης.

Πολυεπίπεδη (n-tier) αρχιτεκτονική: Αυτή η αρχιτεκτονική είναι παραπλήσια με την παραπάνω πελάτη-διακομιστή, αλλά εισάγει επιπλέον επίπεδα μεταξύ του πελάτη και του εξυπηρετητή με σκοπό να κάνει την ανάπτυξη εφαρμογών λιγότερο πολύπλοκη. Η πιο απλή μορφή της είναι η μεταφορά λειτουργικότητας σε ένα ενδιάμεσο ‘stateless’ συστατικο μέρος/υπηρεσία.

Αρχιτεκτονική Ομότιμων (Peer to Peer): Όπου όλα τα μέρη του συστήματος έχουν την ίδια λειτουργικότητα και όλα είναι υπεύθυνα για την ομαλή και σωστή λειτουργία των

υπηρεσιών που προσφέρει. Δεν υπάρχει δηλαδή κάποια κεντρική διαχείριση των πόρων του συστήματος.

2.4 Τεχνολογίες που χρησιμοποιούνται

Σε αυτή την ενότητα γίνεται μία επισκόπηση των τεχνολογιών που χρησιμοποιήθηκαν για την ανάπτυξη της εφαρμογής, ώστε να εξοικειωθεί ο αναγνώστης με αυτές.

2.4.1 Node.js

Εικόνα 1: Λογότυπο Node.Js

Το Node.Js δημιουργήθηκε απο τον Ryan Dahl το 2009 σχεδόν δεκατρία (13) χρόνια μετά την εισαγωγή του πρώτου περιβάλλοντος JavaScript στην πλευρά του διακομιστή (Netscape Enterprise Server). Η αρχική έκδοση υποστηρίζει μονο το Linux και το Mac OS X [12].

Αφορμή για τη δημιουργία του Node.Js στάθηκε μια μπάρα προόδου αποστολής αρχείων στο Flickr. Το πρόγραμμα περιήγησης δε γνώριζε πόσο από το αρχείο έχει μεταμορφωθεί, άρα έπρεπε να απευθυνθεί στο διακομιστή Web. Αυτό δεν άρεσε στο δημιουργό, προτιμούσε ένα πιο εύκολο τρόπο. Έτσι, εξέφρασε αρνητική άποψη για τις περιορισμένες δυνατότητες του Apache HTTP Server (ο πιο δημοφιλής διακομιστής ιστού το 2009) για να χειριστεί πολλές ταυτόχρονες συνδέσεις [12].

Το Node.Js [11] είναι ένα περιβάλλον διακομιστών ανοιχτού κώδικα. Κατέχει μια αρχιτεκτονική βασισμένη σε συμβάντα, η οποία επιτρέπει ασύγχρονα I/O. Οι

συγκεκριμένες επιλογές σχεδιασμού στοχεύουν στη βελτιστοποίηση της απόδοσης και της κλιμάκωσης σε εφαρμογές ιστού με πολλές λειτουργίες εισόδου - εξόδου (I/O) καθώς επίσης και για εφαρμογές Web σε πραγματικό χρόνο, όπως για παράδειγμα προγράμματα επικοινωνιας σε πραγματικό χρόνο και παιχνίδια.

Επιπλέον, το Node.Js μπορει να εκτελείται σε διάφορες πλατφόρμες όπως Windows, Linux, Unix, Mac OS X, είναι δωρεάν και χρησιμοποιεί την JavaScript ως γλώσσα προγραμματισμού. Η JavaScript χρησιμοποιήθηκε κυρίως στη πλευρά του πελάτη (client) με σκοπό να μετατρέψει την στατική HTML σε μια δυναμική και πιο διαδραστική εμπειρία για το χρήστη.

Εκτός από όσα αναφέρθηκαν παραπάνω, το Node.Js μπορεί να δημιουργήσει δυναμικό περιεχόμενο σελίδας, να δημιουργήσει, να ανοίξει, να διαβάσει, να γράψει, να διαγράψει και να κλείσει αρχεία. Μπορεί επίσης να προσθέσει, να διαγράψει, να τροποποιήσει δεδομένα στη βάση δεδομένων σας. Περιλαμβάνει όλα όσα χρειάζεστε για να εκτελέσετε ένα πρόγραμμα γραμμένο σε JavaScript.

Συνοψίζοντας, μπορούμε να πούμε ότι το Node.Js αντιπροσωπεύει ένα παράδειγμα περιβάλλοντος ανάπτυξης εφαρμογών σε JavaScript που επέτρεψε κατά κάποιον τρόπο τη δημιουργία εφαρμογών ιστού γύρω από μια κοινή γλώσσα προγραμματισμού αντί για πολλές διαφορετικές γλώσσες μεταξύ του πελάτη και του διακομιστή. Το API του Node.Js είναι έτσι σχεδιασμένο ώστε να ελαττώνει την πολυπλοκότητα της γραφής των εφαρμογών διακομιστή. Τέλος η δημιουργία του Node.Js αποδεικνύεται πολύ χρήσιμη και για ανάπτυξη desktop εφαρμογών.

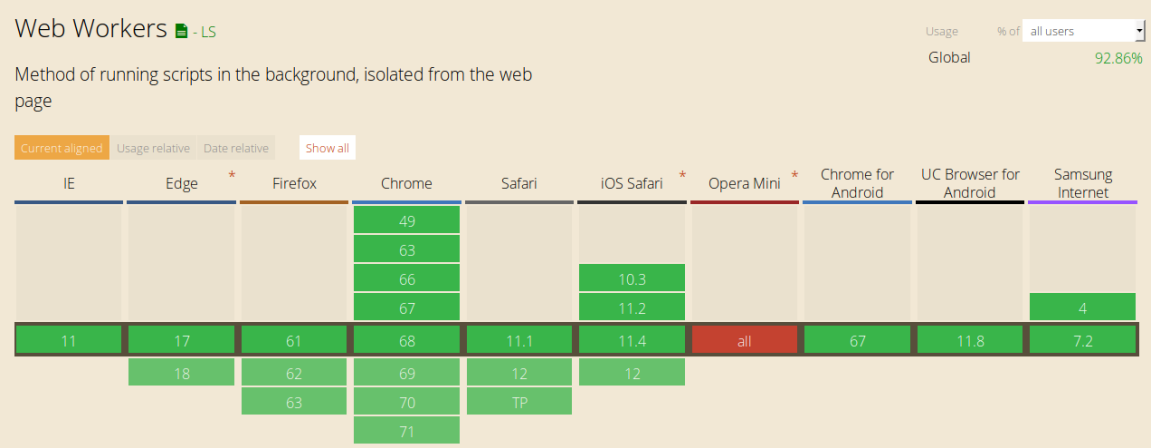
2.4.2 Web Workers

Ένας Web Worker, όπως ορίζεται από το World Wide Web Consortium (W3C) [13] είναι ένα σύνολο JavaScript εντολών που εκτελούνται από μια σελίδα HTML στο παρασκήνιο, χωρίς να επηρεάζει την αλληλεπίδραση του χρήστη με το γραφικό περιβάλλον της εφαρμογής και τις υπόλοιπες ενέργειες που αυτή σχετίζεται. Οι Web Workers είναι σε θέση να χρησιμοποιούν αποτελεσματικα πολυπύρηνα CPU.

Οι Web Workers δίνουν τη δυνατότητα στις ιστοσελίδες να μπορούν να ανταποκρίνονται σε οποιαδήποτε ενέργεια του χρήστη αλλά και ταυτόχρονα να εκτελούν εργασίες που διαρκούν για μεγάλο χρονικό διάστημα στο παρασκήνιο. Η πιο απλή χρήση

των Web Workers είναι η εκτέλεση υπολογισμών οι οποίοι είναι εξαιρετικά δαπανηροί ως προς τη χρήση της CPU χωρίς να εμποδίζουν την αλληλεπίδραση του χρήστη.

Οι Web Workers υποστηρίζονται από όλους σχεδόν τους πιο διαδεδομένους φυλλομετρητές ιστού όπως Chrome, Firefox, Safari και Internet Explorer [14].



Εικόνα 2: Υποστήριξη των Web Workers από διάφορους περιηγητές ιστού

2.4.3 Socket.IO

Εικόνα 3: Λογότυπο Socket.IO

Το Socket.IO [15] είναι μια βιβλιοθήκη JavaScript για διαδικτυακές εφαρμογές πραγματικού χρόνου. Επιτρέπει την επικοινωνία σε πραγματικό χρόνο, αμφίδρομη μεταξύ των Web clients και των διακομιστών. Έχει δύο μέρη: μια βιβλιοθήκη πελάτη που τρέχει στο πρόγραμμα περιήγησης και μια βιβλιοθήκη διακομιστή για Node.Js. Τα δύο συστατικά έχουν σχεδόν ταυτόσημο API. Το Socket.IO όπως και το Node.Js οδηγείται από συμβάντα (event-driven).

Το Socket.IO χρησιμοποιεί το Web Socket ως κύριο πρωτόκολλο και σε περίπτωση που δεν είναι διαθέσιμο χρησιμοποιεί μια άλλη τεχνική, τη γνωστή ως polling,

διατηρώντας το ίδιο API. Μπορεί να χρησιμοποιηθεί ως ένα απλό περιτύλιγμα γύρω από το Web Socket παρέχοντας και άλλες πολλές λειτουργίες.

2.4.4 Angular

Εικόνα 4: Λογότυπο Angular

Η Angular [16] είναι μια πλατφόρμα ανοιχτού κώδικα που επιτρέπει την υλοποίηση εφαρμογών οι οποίες προορίζονται και εκτελούνται στους Web browsers. Αποτελεί την εξέλιξη του πολύ διαδεδομένου AngularJS.

Τη συντήρηση και εξέλιξη της Angular την έχει αναλάβει η Google. Στην ομάδα συμμετέχουν προγραμματιστές οι οποίο συνέβαλαν στην ανάπτυξη της AngularJS. Εκτός από αυτήν, η Angular διαθέτει μια μεγάλη κοινότητα προγραμματιστών οι οποίοι προσφέρουν αφιλοκερδώς το χρόνο και τις γνώσεις τους [17].

Η Angular επιτρέπει στους προγραμματιστές πολύ εύκολα να δημιουργήσουν εφαρμογές μια σελίδας (single page applications), οι οποίες είναι πλούσιες σε περιεχόμενο, animation αλλά επιτρέπουν και τη μεταφορά του routing στη μεριά του client κάνοντας έτσι πιο απλή την ανάπτυξη του server ο οποίος απλά εκθέτει συνήθως μόνο ένα API για την ανάκτηση δεδομένων και εκτέλεση ενεργειών.

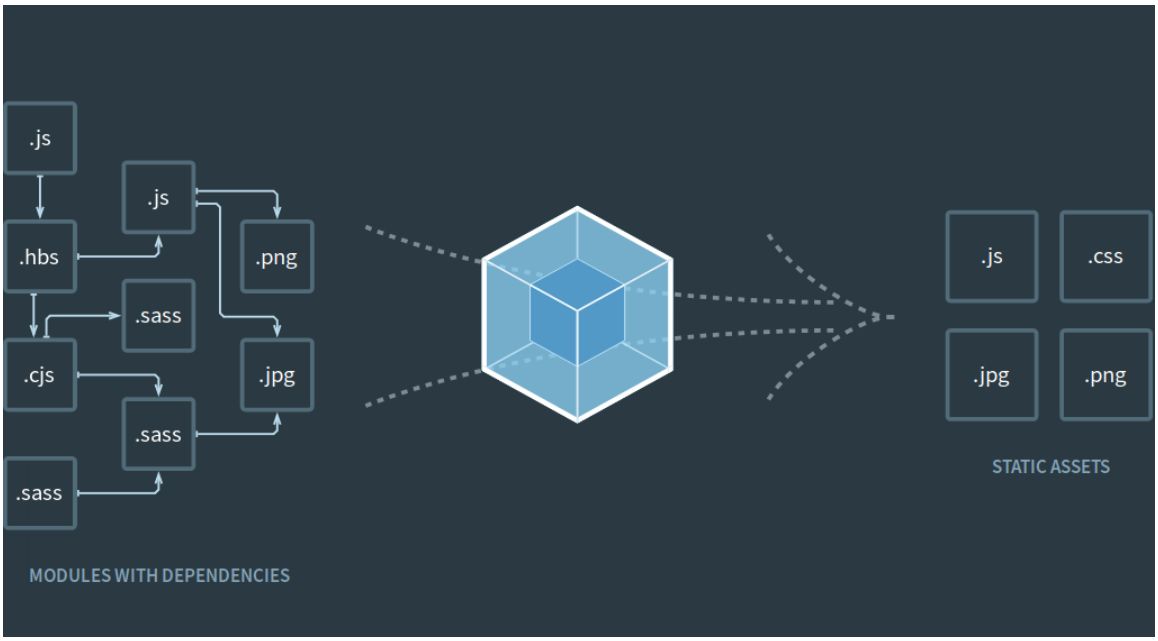
2.4.5 Webpack

Εικόνα 5: Λογότυπο Webpack

Με την ανάπτυξη των τεχνολογιών JavaScript όλο και περισσότερα μεγάλα έργα άρχισαν να υλοποιούνται στη γλώσσα αυτή. Αυτό οδήγησε στην ανάγκη να βρεθεί μία λύση στο τρόπο με τον οποίο δομούνται τα αρχεία του κώδικα του προγράμματος. Δεν μπορούσε να συντηρηθεί ένα μεγάλο έργο γραμμένο μόνο σε μερικά αρχεία αλλά ούτε και

σε πάρα πολλά γιατί έπρεπε αυτά στο τέλος, όταν έρθει η ώρα της διανομής της εφαρμογής να ενωθούν. Από αυτή την ανάγκη δημιουργηθηκε το Webpack [18].

Είναι μία εφαρμογή ανοικτού κώδικα, που έχει μία μεγάλη κοινότητα που το υποστηρίζει και που απλά μαζεύει όλα τα αρχεία που είναι απαραίτητα για την εφαρμογή μας και τα φτιάχνει ένα πακέτο, ανάλογα με τις ρυθμίσεις του βέβαια. Αυτό που το κάνει σημαντικό είναι ο τρόπος που λειτουργεί, που μπορεί να διαβάσει μέσα από κάθε αρχείο ποια είναι τα άλλα αρχεία που αυτό εξαρτάται και να συμπεριλάβει και αυτά το τελικό πακέτο φτιάχνοντας ένα γράφο εξαρτήσεων (dependencies). Τέλος και ίσως και πιο σημαντικό είναι ότι έχει τη δυνατότητα να διαβάσει και να "καταλάβει" και αρχεία διαφορετικά από αρχεία JavaScript, όπως html, css αλλά και αρχεία εικόνων και να βρει όλες τις εξαρτήσεις τους και να τις συμπεριλάβει μαζί με αυτά στο τελικό πακέτο. Αυτό το καταφέρνει με την βοήθεια των Loaders.



Εικόνα 6: Γραφική απεικόνιση λειτουργίας Webpack

2.4.6 TypeScript

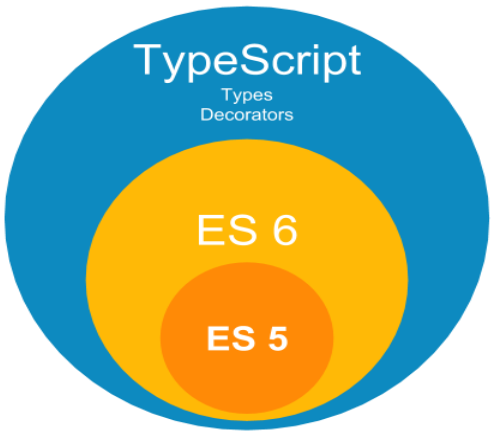
Εικόνα 7: Λογότυπο Typescript

Η ανάπτυξη αλλά κυρίως η συντήρηση μεγάλων έργων λογισμικού γραμμένων σε JavaScript είναι πάρα πολύ δύσκολη και χειροτερεύει όσο αυτά μεγαλώνουν. Αυτό γίνεται για το λόγο ότι η JavaScript είναι "χαλαρή" γλώσσα ως προς τη σύνταξη και δεν

υποστηρίζει τύπους. Και καθώς δε γίνεται compile οι προγραμματιστές είναι συχνά εύκολο να κάνουν λάθος και αυτό να το δουν μόνο κατά την εκτέλεση της εφαρμογής.

Στο πρόβλημα αυτό η Microsoft έδωσε τη λύση, ή έστω μία λύση την TypeScript. Αφού την ανέπτυξε και τη χρησιμοποίησε για περίπου δύο χρόνια για εσωτερικά έργα της εταιρείας, την έκανε διαθέσιμη σε όλους δημοσιεύοντας την και μετατρέποντας την σε λογισμικό ανοιχτού κώδικα, τον Οκτώβριο του 2012 [20].

Η TypeScript [19], λοιπόν, είναι ένα υπερσύνολο της JavaScript προσφέροντας όλα τα χαρακτηριστικά της ECMAScript 6 και επιπλέον τύπους, annotations, interfaces κτλ. Αυτό γίνεται δυνατό καθώς προσθέτει ένα επιπλέον βήμα για την εκτέλεση του κώδικα, το transpile. Το οποίο αυτό που κάνει είναι να μετατρέψει τον κώδικα που είναι γραμμένος σε TypeScript σε απλή JavaScript την οποία καταλαβαίνουν οι JavaScript μηχανές. Κατά την διάρκεια του Transpile γίνεται και ο έλεγχος των τύπων το οποίο σταματά την διαδικασία αν υπάρχει κάποιο λάθος.



Εικόνα 8: Typescript υπερσύνολο της ES 6 & ES 5

Τέλος, λόγω των τύπων γίνεται εύκολο για τους επεξεργαστές κειμένων (IDE) να έχουν καλύτερα συστήματα υποβοήθησης του προγραμματιστή, όπως εξυπνότερη

αυτόματη συμπλήρωση, καλύτερη αναφορά σφαλμάτων αλλά και καλύτερα εργαλεία για refactoring του κώδικα.

2.4.7 Npm

Εικόνα 9: Λογότυπο Npm

Το npm δημιουργήθηκε το 2009 ως ένα έργο ανοιχτού λογισμικού, για να βοηθήσει τους JavaScript προγραμματιστές να μπορούν εύκολα να διανείμουν τον κώδικα τους [22].

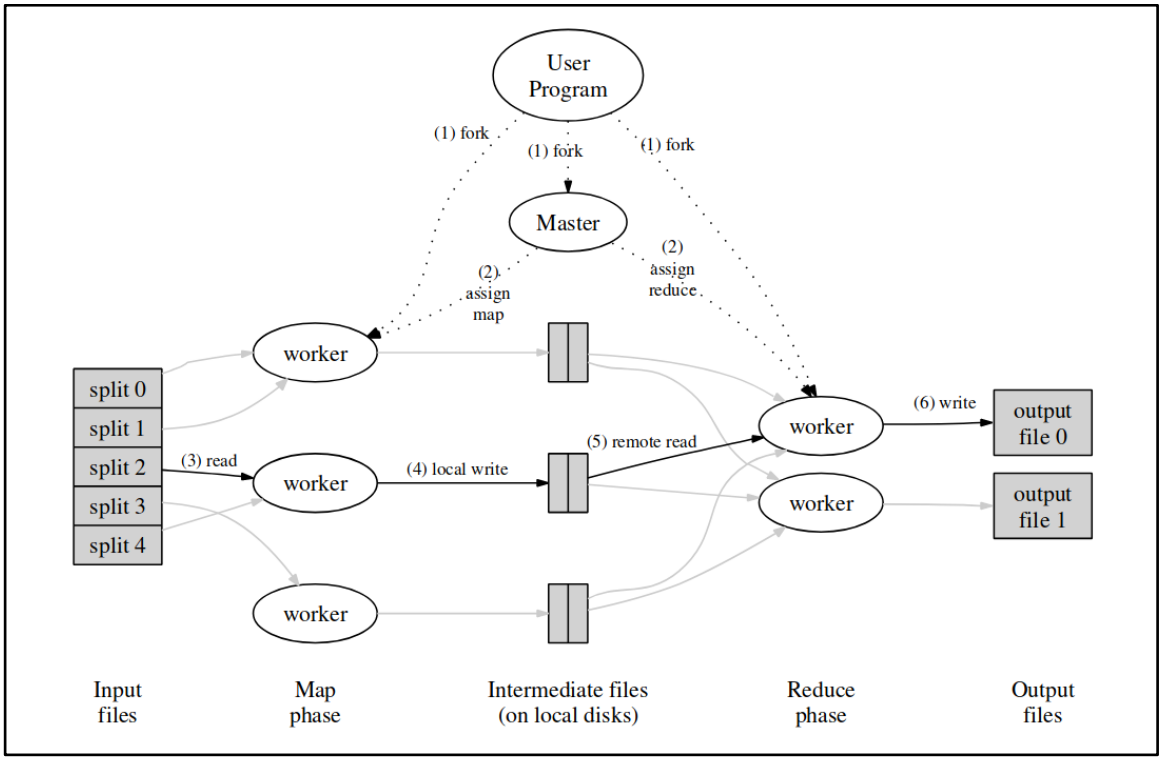
Το npm [21] είναι ένας διαχειριστής βιβλιοθηκών που κυρίως χρησιμοποιείται για τη διαχείριση των πακέτων βιβλιοθηκών στο περιβάλλον εκτέλεσης JavaScript Node.Js. Όλα τα πακέτα βιβλιοθηκών αποθηκεύονται στην ηλεκτρονική βάση του npm (npm registry) και είναι προσβάσιμα από όλους τους προγραμματιστές με τη χρήση προγράμματος πελάτη που είναι διαθέσιμο. Το πρόγραμμα αυτό ονομάζεται και απλά npm.

Το μητρώο πακέτων (registry) μπορεί να είναι είτε δημόσιο, όπου τα πακέτα που περιέχονται εκεί είναι διαθέσιμα σε όλους, είτε ιδιωτικό, κάτι το οποίο παρέχεται ως έξτρα υπηρεσία. Η φιλοξενία και η συντήρηση του npm μητρώου γίνεται από την εταιρία Npm Inc.

2.4.8 MapReduce

Το MapReduce [23] δεν είναι μόνο μία τεχνολογία αλλά ένα προγραμματιστικό μοντέλο το οποίο στοχεύει στην επεξεργασία μεγάλων συνόλων δεδομένων σε κατανεμημένα συστήματα. Οι χρήστες ορίζουν μία συνάρτηση map, η οποία δέχεται ως είσοδο ένα συνδυασμό κλειδιού/τιμής και παράγει σαν αποτέλεσμα ένα άλλο συνδυασμό κλειδιού/τιμής. Το τελευταίο γίνεται η είσοδος της συνάρτησης reduce, την οποία ορίζει και αυτή ο προγραμματιστής και παράγει το τελικό αποτέλεσμα. Η μορφή αυτή του

μοντέλου επιτρέπει την ευκολότερη και αυτοματοποιημένη επεξεργασία σε μεγάλα συστήματα κοινών υπολογιστών.



Εικόνα 10: Απεικόνιση τρόπου λειτουργίας MapReduce

Η ανάπτυξη του MapReduce έγινε από την Google θέλωντας να αντιμετωπίσουν προβλήματα τα οποία είχαν σχέση με την εκτέλεση σχετικά απλών ενεργειών σε πολύ μεγάλα σύνολα δεδομένων. Η εκτέλεση και η υλοποίηση των υπολογισμών που θα έπρεπε να εκτελεστούν θα ήταν εύκολη για μικρά σύνολα δεδομένων. Αλλά, τα μεγέθη δεδομένων τα οποία αντιμετωπίζουν είναι τεράστια και έτσι δημιουργούν προβλήματα που έχουν σχέση με το πως θα γίνει η παραλληλοποίηση αυτών των υπολογισμών, πως θα μοιραστούν τα δεδομένα σε όλο το δίκτυο αλλά και πως θα γίνει ο χειρισμός των σφαλμάτων των υποδομών/μερών στα οποία γίνεται η επεξεργασία.

Το μοντέλο αυτό στηρίζεται και σε ένα κατανεμημένο σύστημα διαχείρισης αρχείων το GFS, το οποίο αναλαμβάνει να κατανείμει στην αρχή της εκτέλεσης, τα αρχεία που χρησιμοποιούνται ως είσοδος για την συνάρτηση map. Εξασφαλίζοντας ότι τα αρχεία αυτά θα είναι πάντα διαθέσιμα στο δίκτυο επεξεργασίας του προβλήματος. Επίσης

χειρίζεται και είναι ανθεκτικό σε τυχόν βλάβες που μπορεί να προκύψουν στα μέρη του συστήματος.

Μία υλοποίηση ανοιχτού κώδικα αυτού του μοντέλου είναι το Apache Hadoop [24], το οποίο χρησιμοποιεί το δικό του κατανεμημένο σύστημα διαχείρισης αρχείων, το Hadoop Distributed File System (HDFS) [25].

3 Σχεδίαση, υλοποίηση και περιγραφή της εφαρμογής

Σε αυτό το κεφάλαιο γίνεται η περιγραφή της εφαρμογής, η οποία αναπτύχθηκε στα πλαίσια της εκπόνησης της παρούσας εργασίας, ώστε να εξετάσουμε μέσα από τη χρήση της αν είναι δυνατή και σε ποιό βαθμό η κατανεμημένη επεξεργασία με χρήση τεχνολογιών διαδικτύου. Έχοντας ως στόχο:

1. την απλοποίηση της διαδικασίας εγγραφής/προσθήκης νέων υπολογιστικών μονάδων στο σύστημα. Να μπορεί εύκολα κάποιος χρήστης δηλαδή να προσφέρει την υπολογιστική δύναμη του υπολογιστή του χωρίς να απαιτείται η εγκατάσταση κάποιας επιπλέον εφαρμογής, εκτός από ένα περιηγητή ιστού που υπάρχει σε όλα τα γνωστά λειτουργικά συστήματα ήδη προεγκατεστημένος.
2. Τη δυνατότητα ο προγραμματιστής να γράψει τον κώδικα για τη λύση του προβλήματος του σε JavaScript, παρέχοντας του ένα απλό προγραμματιστικό μοντέλο.
3. Και τέλος, η απόδοση του συστήματος, να μην είναι πολύ μακριά από άλλες τεχνολογίες που δε χρησιμοποιούν τεχνολογίες ιστού.

3.1 Περιγραφή απαιτήσεων της εφαρμογής

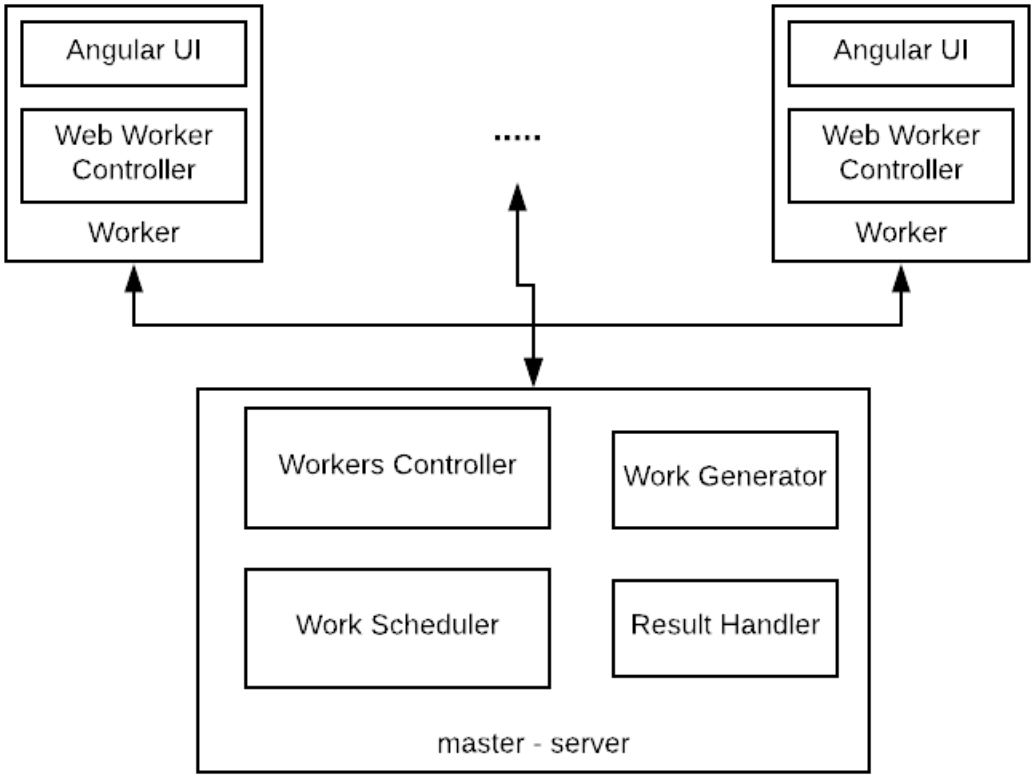
Με βάση τους στόχους που τέθηκαν παραπάνω, σε αυτή την ενότητα περιγράφονται οι απαιτήσεις που προκύπτουν και πρέπει να υλοποιηθούν. Αυτές οι απαιτήσεις είναι οι εξής:

- Δυνατότητα υλοποίησης πολλών διαφορετικών αλγορίθμων με κάποια σχετική ευκολία από τον προγραμματιστή
- Η εφαρμογή θα πρέπει να μπορεί να παραλληλοποιεί την επεξεργασία. Δηλαδή θα πρέπει να μπορεί να χωρίσει την δουλειά σε κομμάτια προκειμένου να τα μοιράσει στο δίκτυο των συνδεδεμένων υπολογιστών
- Θα πρέπει να είναι δυνατή η εκτέλεση της επεξεργασίας σε κάθε ένα από τα κομμάτια δουλειάς στο σύστημα
- Στο τέλος της επεξεργασίας θα πρέπει να συγκεντρώνονται τα αποτελέσματα και να γίνονται διαθέσιμα στον προγραμματιστή ώστε να μπορεί να τα χρησιμοποιήσει όπως αυτός θέλει

- Η αποτυχία επεξεργασίας κάποιου κομματιού εργασίας θα πρέπει να γίνεται αντιληπτή και να μην εμποδίζει την ολοκλήρωση της επεξεργασίας.
- Η παρουσίαση διαφόρων στατιστικών στοιχείων για την κατάσταση του δικτύο είναι απαραίτητη για τη διευκόλυνση των χρηστών.
- Θα πρέπει να υπάρχει επικοινωνία μεταξύ των μερών του συστήματος.

3.2 Αρχιτεκτονική εφαρμογής

Η ανάπτυξη της εφαρμογής στηρίχθηκε σε τρία βασικά χαρακτηριστικά για να μπορέσει να εκπληρώσει τις παραπάνω απαιτήσεις. Στο μοντέλο master-worker, στο MapReduce και στην ιδέα ότι η εφαρμογή μπορεί να λειτουργήσει ως ένα επαναχρησιμοποιήσιμο πακέτο λογισμικού. Στο παρακάτω σχήμα παρουσιάζεται μια επισκόπηση της εφαρμογής.



Εικόνα 11: Απεικόνιση αρχιτεκτονικής της εφαρμογής

Όπως γίνεται φανερό και από το σχήμα, η εφαρμογή αποτελείται από δύο μέρη, τον Worker και τον Master (Server). Από τις γραμμές στο σχήμα επίσης, φαίνεται ότι η επικοινωνία μεταξύ των μερών του συστήματος είναι μόνο μεταξύ του Master με τους Worker και όχι μεταξύ των Worker. Αυτό γίνεται πρώτον, διότι με βάση το μοντέλο που ακολουθήθηκε δεν χρειάζεται η επικοινωνία μεταξύ των Workers αλλά και γιατί αυτή η επικοινωνία θα ήταν σχετικά δύσκολη λόγω της φύσης των περιηγητών ιστού. Υπάρχει η δυνατότητα βέβαια η χρήση της τεχνολογίας WebRTC για αμφίδρομη επικοινωνία μεταξύ τους αν αυτό χρειαστεί.

3.2.1 Worker

Ο Worker υλοποιείται από μια Angular εφαρμογή και εκτελείται από τους browsers των χρηστών που προσφέρουν τους υπολογιστές τους στο δίκτυο. Αναλαμβάνει την εκτέλεση της επεξεργασίας με βάση το κομμάτι δουλείας που του αποστέλλεται από τον Master αλλά και την παρουσίαση διαφόρων στατιστικών για την κατάσταση του συστήματος.

Η επεξεργασία γίνεται με την χρήση των Web Workers ώστε να μην επηρεάζεται η πλοήγηση του χρήστη στο γραφικο περιβαλλον της σελίδας. Ο κώδικας ο οποίος θα εκτελεστεί από την εφαρμογή για την επεξεργασία των κομματιών εργασίας αποστέλλεται από τον Master κατά την διάρκεια αρχικοποίησης των Web Workers. Ο αριθμός τους εξαρτάται από των αριθμών των πυρήνων της CPU και από προεπιλογή είναι τουλάχιστον ένας ή ο μισός αριθμός τους, όποιο είναι μεγαλύτερο. Μετά την ολοκλήρωση της επεξεργασίας κάθε εργασίας, το αποτέλεσμα αποστέλλεται στον Master και ο Worker περιμένει την επόμενη εργασία η οποία του ορίζεται από τον Master.

Το γραφικό μέρος του Worker αποτελείται από δύο οθόνες. Η μία παρουσιάζει, με την βοήθεια γραφημάτων, την κατάσταση του υπολογιστή του χρήστη με στοιχεία όπως αριθμός πυρήνων CPU, αριθμός πυρήνων που χρησιμοποιούνται, πρόοδο μιας εργασίας κλπ. Τα στοιχεία αυτά υπολογίζονται από τον Worker και τα γραφήματα ανανεώνονται όποτε αυτός το επιλέξει. Η άλλη οθόνη δείχνει την κατάσταση όλου του δικτύου με παρόμοια στατιστικά στοιχεία. Τα στοιχεία αυτά υπολογίζονται από τον Master, ο οποίος τα στέλνει στους Workers που έχουν κάνει εγγραφή (subscribe) για να τα λαμβάνουν.

Ο Worker, δεν προορίζεται να είναι διαθέσιμος στον προγραμματιστή για αλλαγές κατά την υλοποίηση διαφορετικών προβλημάτων. Τον κώδικα που μπορεί να αλλάξει ο

προγραμματιστής είναι αυτός του Web Worker και αυτό γίνεται κατά την αρχικοποίηση του Master.

3.2.2 Master

Ο Master είναι πιο περίπλοκος διότι αναλαμβάνει να διαχειριστεί τους συνδεδεμένους Workers, να δημιουργήσει τα κομμάτια εργασιών, να τα προγραμματίσει ώστε να γίνει η εκτέλεση τους στο δίκτυο, να συγκεντρώσει τα αποτελέσματα τους και να παράξει το τελικό αποτέλεσμα.

Κατά την εκτέλεση, ο Master περιμένει συνδέσεις στην θύρα 8080 από προεπιλογή τύπου http που μπορούν να αναβαθμιστούν και σε ws (Web Socket). Από εκεί σερβίρει τα στατικά αρχεία για την εφαρμογή του Worker αλλά υλοποιεί και ένα API. Χρησιμοποιώντας αυτό το API οι workers έχουν πρόσβαση στον κώδικα του Web Worker. Αυτή η λειτουργικότητα θα μπορούσε να μεταφερθεί σε άλλη υπηρεσία άλλά επιλέγεται να μείνει εδώ για μεγαλύτερη ευκολία στην εκτέλεση (deploy) της εφαρμογής.

Η επικοινωνία των μερών του συστήματος, γίνεται με ανταλλαγή μηνυμάτων μέσω της σύνδεσης ws. Αυτό επιλέχθηκε θέλοντας να αποφύγουμε τεχνικές polling από τους workers γιατί υπάρχουν μηνύματα τα οποία στέλνονται από τον master μόλις για παράδειγμα κάποια κομμάτια εργασίας είναι διαθέσιμα. Η δομή των μηνυμάτων είναι τύπου JSON, τα οποία σειριοποιούνται και αποσειριοποιούνται κάνοντας χρήση της “έμφυτης” υποστήριξης της JavaScript.

3.2.2.1 Μοντέλο λειτουργίας

Για την εκτέλεση της επεξεργασίας δημιουργούνται τα κομμάτια εργασίας, τα οποία θα πρέπει να περιέχουν οποιαδήποτε πληροφορία είναι απαραίτητη για τον Worker αλλά και ένα μοναδικό ID σε σχέση με όλες τις άλλες εργασίες. Η επεξεργασία του προβλήματος χωρίζεται σε κύκλους. Κάθε ένας κύκλος αποτελείται από τη δημιουργία των εργασιών, την επεξεργασία τους και τέλος την επεξεργασία των αποτελεσμάτων του κύκλου. Αυτό μπορεί να έχει σαν αποτέλεσμα τη δημιουργία νέου κύκλου εργασιών ή την ολοκλήρωση του προβλήματος εφόσον δεν υπάρχει άλλος κύκλος επεξεργασίας. Σημαντικό είναι να τονιστεί ότι σε κάθε κύκλο θα πρέπει να εκτελείται μόνο επεξεργασία σε εργασίες ίδιου τύπου.

Ο προγραμματιστής που θέλει να χρησιμοποιήσει την εφαρμογή για να λύσει κάποιο πρόβλημα, πρέπει να γράψει τον κώδικα του χρησιμοποιώντας τον Master. Εκεί πρέπει να υλοποιήσει τρία βασικά μερη.

Την γεννήτρια κομματιών εργασίας (Work Generator): Εδώ γίνεται η δημιουργία των κομματιών εργασίας. Έχοντας ο προγραμματιστής πρόσβαση στα αποτελέσματα των εργασιών, μπορεί να δημιουργήσει και άλλες εργασίες, οι οποίες θα εκτελεστούν στον επόμενο κύκλο. Όταν πια, δεν είναι απαραίτητη η δημιουργία νέων εργασιών από τα αποτελέσματα των προηγουμένων, τότε ο προγραμματιστής ενημερώνει το σύστημα. Και έτσι μόλις ολοκληρωθεί η επεξεργασία όλων των εργασιών ο προγραμματιστής έχει πρόσβαση στα αποτελέσματα. Αυτά μπορεί να τα επεξεργαστεί ξανά αλλά σε αυτό το σημείο η επεξεργασία γίνεται στο master οπότε θα πρέπει να είναι κάτι εύκολο υπολογιστικά.

Τον κώδικα του Web Worker: Αυτός είναι ο κώδικας που θα χρησιμοποιηθεί κατά την αρχικοποίηση των Workers. Θα εκτελεστεί μόλις λάβει ο Worker κάποιο κομμάτι εργασίας. Το API είναι αυτό του Web Worker, οπότε η επικοινωνία με την κύρια εφαρμογή θα πρέπει να γίνει με ανταλλαγή μηνυμάτων.

Και τέλος, τον *Χειριστή των αποτελεσμάτων*: Που είναι αυτός που στο τέλος θα έχει πρόσβαση σε όλα τα αποτελέσματα και σε αυτό το σημείο ο προγραμματιστής μπορεί να εξάγει τη τελική λύση ή και να κάνει ό,τι άλλο χρειαστεί.

3.2.3 Επικοινωνία Master με Workers

Θέλωντας να επιτύχουμε αμφίδρομη επικοινωνία μεταξύ master και workers, γίνεται χρήση της τεχνολογίας WebSockets των περιηγητών ιστού. Η τεχνολογία αυτή μας δίνει τη δυνατότητα να μπορούν και τα δύο μέρη να επικοινωνήσουν με το άλλο, χωρίς να πρέπει πάντα ο Worker να ξεκινά τη επικοινωνία. Έτσι ο Master μπορεί να στέλνει κομμάτια εργασίας στους Workers, όποτε αυτά είναι διαθέσιμα, χωρίς οι τελευταίοι να πρέπει να ρωτάνε κάθε ένα συγκεκριμένο χρονικό διάστημα. Τα WebSockets δεν χρησιμοποιούνται αυτούσια, αλλά υπάρχει μία βιβλιοθήκη γύρω από αυτά, το Socket.IO. Αυτό διευκολύνει αρκετά την χρήση τους, αλλά παρέχει και δυνατότητες, όπως το ήδη υλοποιημένο heartbeat που μας επιτρέπει να καταλάβουμε αν έχουμε χάσει την επικοινωνία με τον Master.

3.3 Υλοποίηση - περιγραφή εφαρμογής

Σε αυτή την ενότητα γίνεται η παρουσίαση της υλοποίησης αλλά και του περιβάλλοντος διεπαφής της εφαρμογής.

3.3.1 Worker

Όπως έγινε αναφορά σε προηγούμενο κεφάλαιο, η εφαρμογή του Worker είναι μια εφαρμογή Angular. Η εφαρμογή είναι σχετικά απλή και χρησιμοποιεί βασικές λειτουργίες της Angular. Ο κώδικας έχει γραφτεί σε TypeScript. Αυτό που θα ήταν ενδιαφέρον να περιγράψουμε από άποψη υλοποίησης, είναι το service WebWorkersService.ts, γιατί είναι αυτό που αναλαμβάνει να διαχειριστεί τους Web Workers, να τους τροφοδοτήσει με κομμάτια εργασιών αλλά και να συγκεντρώσει τα αποτελέσματα και να τα στείλει στο Master.

Μόλις η εφαρμογή είναι έτοιμη να δεχτεί εργασίες, καλεί τη συνάρτηση init() του WebWorkersService. Αυτό γίνεται στο επίπεδο του AppComponent έτσι ώστε η επεξεργασία να γίνεται συνέχεια σε οποιοδήποτε επίπεδο κι αν βρίσκεται ο χρήστης.

```
public init(): void {
    if (this.mIsInitialized) {
        throw new Error('Service is already initialized.');
    }
    this.handleSocketConnectionStatus();
    this.handleSocketMessages();
    this.mIsInitialized = true;
}

private handleSocketConnectionStatus() {
    this.mSocketService.connectionStatus.subscribe(
        (connected) => {
            if (connected) {
                this.initializeWebWorkers();
                this.mSocketService
                   .sendRegisterMessage(this.mWebWorkersToInititalize);
            } else {
                this.mWebWorkers.forEach((aWorker) => {
                    if (this.mWorkerToWorkIdMap.has(aWorker)) {
                        this.updateWorkStatus(
                            this.mWorkerToWorkIdMap.get(aWorker),
                            WorkStatuses.failed
                        );
                        this.mWorkerToWorkIdMap.delete(aWorker);
                    }
```

```
                    aWorker.terminate();
                });
                this.mWebWorkers = [];
                this.mAvailableWorkers = [];
                this.notifyForWorkersAvailabilityNumbers();
            }
        });
}

private handleSocketMessages() {
    this.mSocketService.messages
        .pipe(filter((message) => {
            return message.type === SocketMessageTypes.startWork;
        }))
            .subscribe((message) => {
                this.updateWorkStatus(
                    this.getWorkIdsFromWorkMessage(message),
                    WorkStatuses.received
                );
                this.sendWorkToAvailableWorker(message);
            });
}
```

Βλέπουμε από την αρχικοποίηση ότι χειρίζεται συμβάντα SocketConnectionStatus και SocketMessages. Για τα μεν πρώτα, αυτό γίνεται για να μπορέσει να γίνει η αρχικοποίηση ή η καταστροφή των Web Workers ανάλογα αν υπάρχει σύνδεση με τον master ή όχι και να αποσταλεί το μήνυμα register ώστε να ενημερωθεί ο master για τους νέους διαθέσιμους πόρους. Ο χειρισμός των συμβάντων λήψης μηνύματος όπως βλέπουμε είναι απαραίτητος για να τροφοδοτήσουμε με δουλειά τους Web Workers.

```
private sendWorkToAvailableWorker(workMessage: any) {
    const aWorker = this.mAvailableWorkers.shift();
    const workIds = this.getWorkIdsFromWorkMessage(workMessage);
    const work = workMessage.work;
    if (aWorker) {
        this.notifyForWorkersAvailabilityNumbers();
        this.updateWorkStatus(workIds, WorkStatuses.submited);
        this.mWorkerToWorkIdMap.set(aWorker, workIds);
        aWorker.postMessage(work);
    }
}
```

Η συνάρτηση sendWorkToAvailableWorker είναι αυτή που πρέπει να θυμόμαστε και κυρίως το γεγονός ότι στέλνουμε στον Web Worker (aWorker.postMessage) την εργασία work όπως ακριβώς τη λάβαμε από τον master και για την ακρίβεια το πεδίο work από το μήνυμα που λήφθηκε.

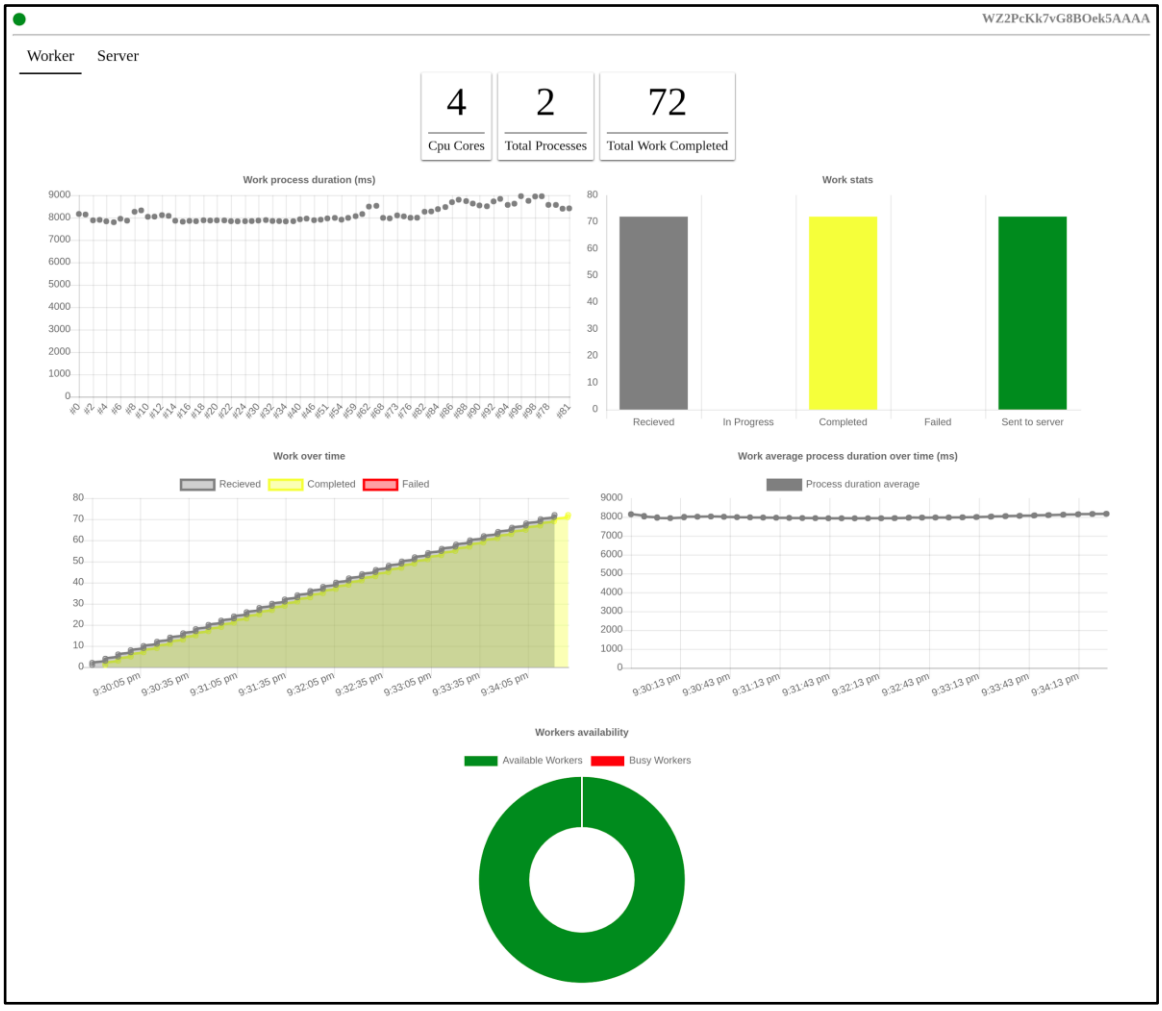
```
private initializeWebWorkers() {
    this.mWebWorkers = [];
    this.mAvailableWorkers = [];
    for (let i = 0; i < this.mWebWorkersToInititalize; i++) {
        const aWorker = new Worker('/api/webworker.js');
        this.mWebWorkers.push(aWorker);
        this.handleMessagesFromWebWorker(aWorker, i);
        this.mAvailableWorkers.push(aWorker);
    }
    this.notifyForWorkersAvailabilityNumbers();
}

private handleMessagesFromWebWorker(aWorker: Worker, index: number) {
    aWorker.onmessage = (workerMessage) => {
        const workResult = workerMessage.data;
        const workIds = this.mWorkerToWorkIdMap.get(aWorker);
        this.mWorkerToWorkIdMap.delete(aWorker);
        this.mAvailableWorkers.push(aWorker);
        this.notifyForWorkersAvailabilityNumbers();
        this.updateWorkStatus(workIds, WorkStatuses.completed);
        this.mSocketService.sendWorkCompleted(workIds.id, workResult.data);
        this.updateWorkStatus(workIds, WorkStatuses.sentToServer);
    };
}
```

Τέλος, για το service WebWorkersService, όπως φαίνεται στις παραπάνω συναρτήσεις, όταν γίνεται η αρχικοποίηση των Web Workers χρησιμοποιούμε το API του Master ώστε να εκτελεστεί ο κώδικας που έχει γράψει ο εκάστοτε προγραμματιστής για το εκάστοτε πρόβλημα. Σε αυτό το σημείο, βλέπουμε επίσης ότι χειριζόμαστε τα μηνύματα από τους Web Workers (handleMessagesFromWebWorker) και για την ακρίβεια περιμένουμε πάντα μόνο ένα είδος μηνύματος, το αποτέλεσμα της εργασίας. Αυτό το στέλνουμε κατευθείαν στον master, όπως το έχει φτιάξει ο Web Worker (this.mSocketService.sendWorkCompleted(workIds.id, workResult.data);) και κάνουμε το Web Worker διαθέσιμο ώστε να λάβουμε νέα κομμάτια εργασιών.

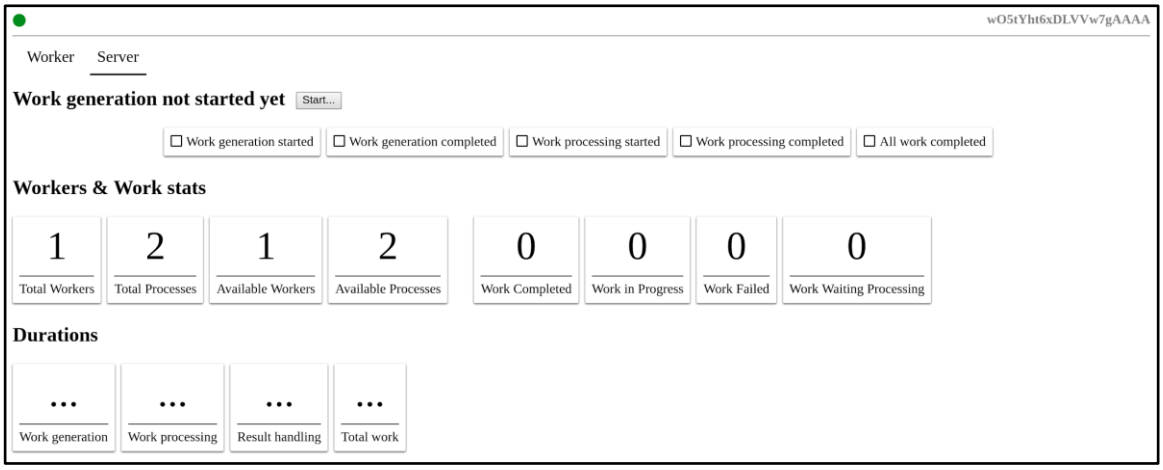
Σχετικά με το γραφικό περιβάλλον, όπως θα δούμε παρακάτω χωρίζεται σε δύο οθόνες που περιέχουν στατιστικά στοιχεία για τον υπολογιστή αλλά και όλο το δίκτυο. Αξίζει να σημειωθεί ότι για τα γραφήματα χρησιμοποιήθηκε η βιβλιοθήκη Chart.js. [26]

Στην επόμενη εικόνα φαίνεται η αρχική σελίδα της εφαρμογής που είναι τα στατικά στοιχεία για τον συγκεκριμένο Worker που έχουμε συνδεθεί. Εκτός από αυτά όμως υπάρχουν οι ενδείξεις για την κατάσταση της σύνδεσης με τον Master και το μενού από το οποίο μπορούμε να πλοηγηθούμε στα στατιστικά στοιχεία του δικτύου (Server).



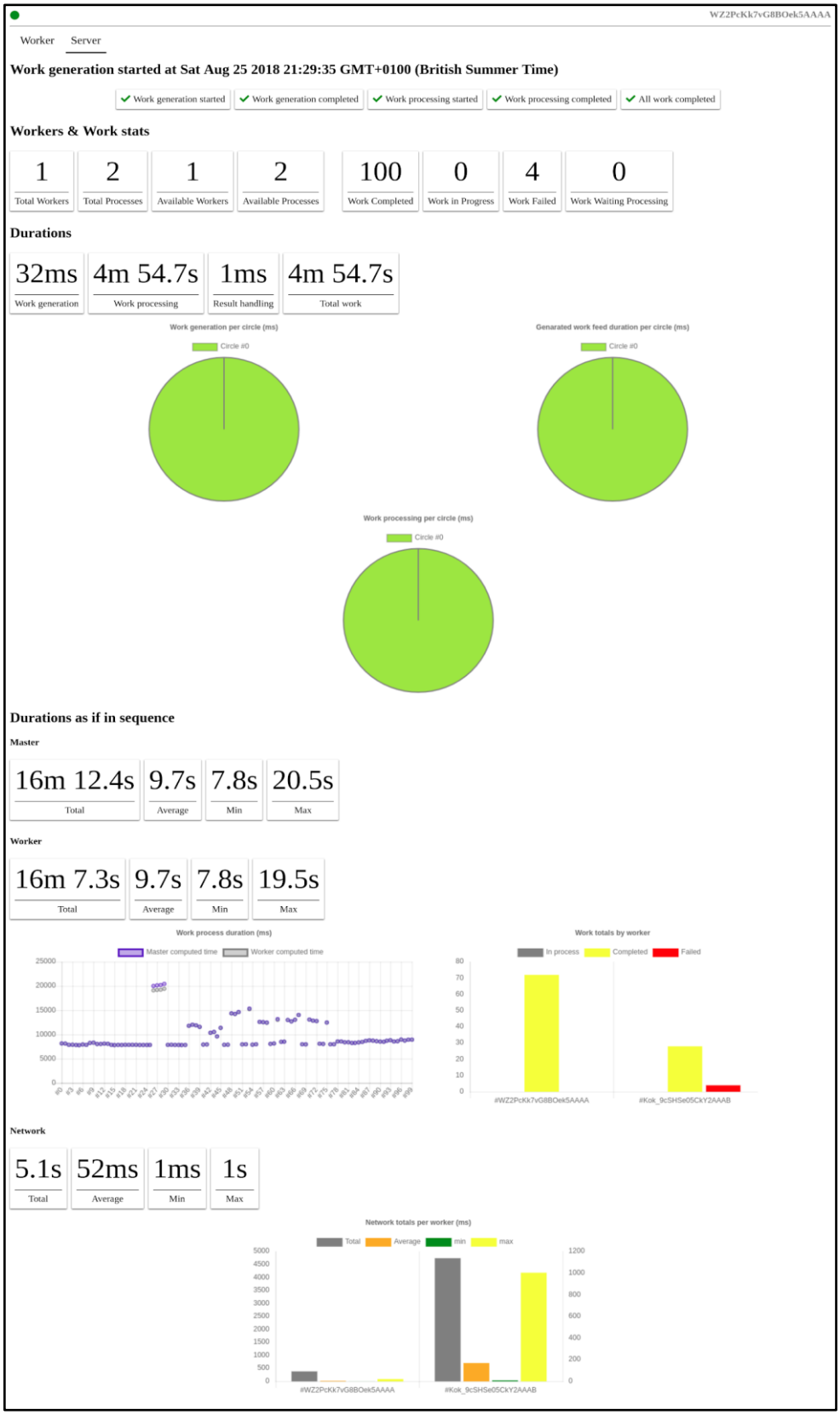
Εικόνα 12: Οθόνη με στατιστικά στοιχεία Worker

Πηγαίνοντας στην σελίδα Server υπάρχουν δύο περιπτώσεις. Να μην έχει ξεκινήσει ακόμη η δημιουργία εργασιών στο Master και να υπάρχει ένα κουμπί start με ένα σχετικό μήνυμα, όπως φαίνεται στην επόμενη εικόνα.



Εικόνα 13: Οθόνη με στατιστικά στοιχεία Master πριν αρχίσει η επεξεργασία

Αυτό το κουμπί υπάρχει εκεί για να διευκολύνει κατά την διάρκεια της εξέτασης και ανάλυσης της εφαρμογής και δεν είναι απαραίτητο. Η οθόνη που θα είναι συνήθως ορατή στο χρήστη θα είναι αυτή με τα στατιστικά στοιχεία του δικτύου, όπως φαίνεται στην επόμενη εικόνα.



Εικόνα 14: Οθόνη με στατιστικά στοιχεία Master αφού ολοκληρωθεί η επεξεργασία

3.3.2 Master

Η υλοποίηση της εφαρμογής του Master έχει γίνει με χρήση της TypeScript, με στόχο τη διευκόλυνση της συγγραφής του κώδικα αλλά και μετά την συγγραφή να είναι πιο εύκολη η κατανόηση του. Το περιβάλλον εκτέλεσης είναι το Node.js το οποίο μπορεί να εκτελέσει JavaScript και γι αυτό το λόγο υπάρχει ένα επιπλέον βήμα της μετατροπής του κώδικα σε απλή JavaScript, το λεγόμενο Transpiling.

Η εφαρμογή του Master δεν έχει κάποιο γραφικό περιβάλλον οπότε σε αυτή την ενότητα γίνεται η περιγραφή ορισμένων κλάσεων που μπορούν να θεωρηθούν σημαντικές για την εφαρμογή.

Η κλάση WebBasedParallelizer είναι αυτή που χρησιμοποιεί ο προγραμματιστής ώστε να υλοποιήσει τον κώδικα για την επίλυση του προβλήματος του. Όπως φαίνεται από τον κατασκευαστή της κλάσης παρακάτω, δέχεται ως παραμέτρους τον κώδικα του Web Worker στην πρώτη παράμετρο webWorkerCode, την κλάση του κατασκευαστή εργασιών που πρέπει να υλοποιεί την διεπαφή (interface) workGenerationContract και την κλάση που διαχειρίζεται τα τελικά αποτελέσματα workResultHandler, στις επόμενες δύο αντίστοιχα παραμέτρους. Η διεπαφή WorkGenerationContract και η κλάση WorkResultHandler παρουσιάζονται αργότερα σε αυτό το κεφάλαιο.

```
constructor(
    webWorkerCode: string,
    workGenerationContract: WorkGenerationContract,
    workResultHandler: WorkResultHandler,
    host: string = '0.0.0.0',
    port: number = 8080
) {
    this.server = new Server(this.staticFilesLocation, webWorkerCode, host,
port);
    this.workersController = new WorkersController(this.server);
    this.mWorkStorage = new WorkStorage();
    this.workScheduler = new WorkScheduler(this.workersController,
this.mWorkStorage);
    workResultHandler.setWorkScheduler(this.workScheduler);
    this.workGenerator = new WorkGenerator(this.workScheduler,
workGenerationContract, this.mWorkStorage, workResultHandler);
    this.mMessagesExecutor = new MessagesExecutor(this.workersController,
this.workGenerator, this.workScheduler, workResultHandler,
this.mWorkStorage, this);
    this.updateStats();
}
```

Επίσης στον κατασκευαστή βλέπουμε ότι γίνεται η αρχικοποίηση της κλάσης Server, η οποία αναλαμβάνει την επικοινωνία με τους Workers μέσω Web Socket, την διανομή των αρχείων της εφαρμογής των Worker και τον κώδικα των Web Workers τον οποίο έχει δώσει ο χρήστης ως παράμετρο.

```
export class Server {
    ...
    constructor(staticFilesLocation: string, webWorkerCode: string, host:
string, port: number) {
        this.staticFilesLocation = staticFilesLocation;
        this.mWebWorkerCode = webWorkerCode;
        this.host = host;
        this.port = port;
        this.expressApp = express();
        this.httpServer = new http.Server(this.expressApp);
        this.initializeStaticFilesServer();
        this.initializeWebSocketServer();
    }

    public start() {
        this.httpServer.listen(this.port, this.host, () => {
            this.logger.info(`Serving static files from
${this.staticFilesLocation} to http://${this.host}:${this.port}/`);
        });
    }

    private initializeStaticFilesServer() {
        this.expressApp.use("/", express.static(this.staticFilesLocation))
            .use("/api/webworker.js", (req: Request, res: Response) => {
                res.send(this.mWebWorkerCode);
            })
            .use('*', (req, res) => {
                res.sendFile(path.join(this.staticFilesLocation,
'index.html'));
            });
    }

    private initializeWebSocketServer() {
        this.webSocketServer = socketio(this.httpServer, {
            path: "/connect",
            serveClient: false,
            pingInterval: 15000,
            pingTimeout: 20000
        });
        this.connectionsStream = new Observable<ClientConnection>(
            (observer) => {
```

```
                this.webSocketServer.on("connection", (socket) => {
                    this.logger.info('New connection', socket.id);
                    observer.next(new ClientConnection(socket));
                });
            }).pipe(share());
    }
    ....
}
```

Στην κλάση του Server παραπάνω βλέπουμε ότι γίνεται χρήση της βιβλιοθήκης rxjs [27], η οποία μας επιτρέπει να απλοποιήσουμε τη διαχείριση των μηνυμάτων της σύνδεσης Web Socket με την χρήση της έννοιας των ροών (streams) και την υλοποίηση τους από την βιβλιοθήκη, τα Observables.

Κάτι άλλο που είναι σημαντικό να αναφέρουμε είναι η κλάση MessagesExecutor η οποία έχει πρόσβαση σε σχεδόν όλα τα στιγμιότυπα (instances) των κλάσεων που δημιουργούνται στον κατασκευαστή του WebBasedParallelizer. Αυτό γίνεται έτσι ώστε να έχουμε την δυνατότητα να χειριστούμε τα εισερχόμενα μηνύματα από του Workers σε ένα κεντρικό σημείο. Όπως βλέπουμε από τον κώδικα του MessagesExecutor αλλά και ενός από τα μηνύματα που υπάρχουν στον φάκελο /src/communications/messages/incomming, το RegisterMessage.

```
export class MessagesExecutor {
    ...
    private executeIncomingMessages(): void {
        this.mIncomingMessagesSubscription = this.mWorkersController
            .workersMessages.subscribe((workerMessage) => {
                workerMessage.message.execute(
                    workerMessage.worker,
                    this.mWorkersController,
                    this.mWorkGenerator,
                    this.mWorkResultHandler,
                    this.mWorkStorage,
                    this.mWebBasedParallelizer
                );
            });
    }
}

export class RegisterMessage extends IncommingMessage {
```

```
    public static TYPE: string = 'register';
    private numOfThreads: number;

    constructor(jsonMessage: any) {
        super(jsonMessage);
        this.numOfThreads = jsonMessage.numOfThreads;
    }

    public execute(worker: Worker, workersController:
WorkersController, workGenerator: WorkGenerator, workResultHandler:
WorkResultHandler, workStorage: WorkStorage, webBasedParallelizer:
WebBasedParallelizer) {
        this.logger.debug(`[Incoming message] Executing register.
Total threads: ${this.numOfThreads}`);
        worker.register(this.numOfThreads);
    }
}
```

Τον έλεγχο και το συντονισμό των Workers αναλαμβάνει ο WorkersController. Η κλάση αυτή ενημερώνει για την ύπαρξη διαθέσιμων Worker αλλά και για οποιαδήποτε αποτυχία κάποιας εργασίας σε κάποιον Worker, ώστε αυτή να προγραμματιστεί ξανά για επεξεργασία.

```
export class WorkersController {
....
    constructor(server: Server) {
        this.server = server;
        this.subscribeForNewConnections();
    }

    private subscribeForNewConnections() {
        this.server.connections.subscribe((connection) => {
            let worker = new Worker(connection, this.mFailedWorkStream);
            this.handleWorkerMessages(worker);
            this.handleWorkerState(worker);
        });
    }
....

 private handleWorkerState(worker: Worker): void {
        worker.state.subscribe(
            (state) => {
                this.logger.debug(`Worker ${worker.id} new state:
${WorkerStatus[state]}`);
```

```
                switch (state) {
                    case WorkerStatus.connected: {
                        this.allWorkers.push(worker);
                        break;
                    }
                    case WorkerStatus.ready:
                    case WorkerStatus.workingAvailableThreads: {
                        this.workerAvailable(true, worker);
                        break;
                    }
                    case WorkerStatus.working: {
                        this.workerAvailable(false, worker);
                        break;
                    }
                    case WorkerStatus.disconnected: {
                        this.workerAvailable(false, worker);
                        let workerIndex = this.allWorkers.indexOf(worker);
                        this.logger.debug(`Worker index: ${workerIndex}`);
                        if (workerIndex > -1) {
                            this.allWorkers.splice(workerIndex, 1);
                        }
                    }
                }
            },
            () => {},
            () => {
                this.logger.debug(`Worker ${worker.id} state completed`);
            }
        )
    }
....
}
```

Τον προγραμματισμό και τη διανομή των εργασιών, όπως και την ενημέρωση για την ολοκλήρωση τους αναλαμβάνει ο WorkScheduler. Αυτός έχει τη λίστα των διαθέσιμων εργασιών και κάθε φορά που υπάρχει διαθέσιμος Worker του αποστέλλει ένα κομμάτι εργασίας. Επίσης, επειδή είναι αυτός που γνωρίζει πότε έχουν τελειώσει όλες οι εργασίες ενημερώνει για τα αποτελέσματα τον ResultHandler.

```
export class WorkScheduler {
...
    constructor(workersController: WorkersController, workStorage:
WorkStorage) {
        this.workersController = workersController;
```

```
        this.mWorkStorage = workStorage;
        this.workersController.workersAvailable
            .pipe(
                delay(0),
                combineLatest(
                    this.availableWork
                        .pipe(delay(0), debounceTime(200))
                )
            )
            .subscribe((data) => this.giveWorkToAvailableWorkers(data[0],
data[1]));
        this.handleFailedWork();
    }
...
    public workCompleted(work: Work, worker: Worker): void {
        this.logger.info(`[WorkScheduler] Work #${work.id} completed at
worker: #${worker.id}`);
        let assignedWorker = this.mCurrentRunningWork.get(work.id);
        if(!assignedWorker) {
            this.logger.debug(`[WorkScheduler] Work not found in current
working list`);
            throw new Error(`Work #${work.id} not found in current work
list.`);
        }
        if(assignedWorker.id !== worker.id) {
            this.logger.debug(`[WorkerScheduler] Assigned worker missmatch
#${assignedWorker.id} != #${worker.id}`);
            throw new Error(`Assigned worker missmatch #${assignedWorker.id}
!= #${worker.id}`);
        }
        this.mServerStatsController.processingCompletedForWork(work.id,
worker.id);
        this.mCurrentRunningWork.delete(work.id);
        this.emmitIfAllWorksCompleted();
    }

    public workGenerationAndSupplyCompleted(): void {
        this.logger.debug(`[WorkScheduler] Work generation completed.`);
        this.mWorkGenerationCompleted = true;
        this.emmitIfAllWorksCompleted();
    }

    private emmitIfAllWorksCompleted(): void {
        // TODO: maybe could find a better solution for detecting if current
circle completed
        let currentWorkCircleCompleted = !this.availableWork.value.length &&
!this.mCurrentRunningWork.size;
        let areAllCiclesCompleted = this.mWorkGenerationCompleted &&
currentWorkCircleCompleted;
```

```
        this.logger.debug(`Checking if all works completed:
${areAllCiclesCompleted}, current circle: ${currentWorkCircleCompleted}`);
        if (areAllCiclesCompleted) {
            this.mServerStatsController.worksProcessingCompleted();
            this.mServerStatsController.allWorkProcessingCompleted();
            this.mAllWorksCompleted.value !== areAllCiclesCompleted &&
this.mAllWorksCompleted.next(areAllCiclesCompleted);
        } else if (currentWorkCircleCompleted){
            this.mServerStatsController.worksProcessingCompleted();
            this.mAllWorksForCurrentCircleCompleted.value !==
currentWorkCircleCompleted &&
this.mAllWorksForCurrentCircleCompleted.next(currentWorkCircleCompleted);
        }
    }

    private giveWorkToAvailableWorkers(workers: Worker[], workIds:
string[]): void {
        this.logger.debug(`Giving work to workers. Available workers:
${workers.length}, work: ${workIds.length}`);
        let updateWorkList: boolean = false;
        if (workers.length && workIds.length) {
            let worker = workers[0];
            let workId = workIds[0];
            if (workId) {
                this.logger.log(`Sending work ${workId} to worker:
${worker.id}`);
                this.mServerStatsController.processingStartedForWork(workId,
worker.id);
                //TODO: same work to many workers?
                this.mCurrentRunningWork.set(workId, worker);
                updateWorkList = true;
                worker.workOn(this.mWorkStorage.getWork(workId));
            }
        }
        if (updateWorkList) {
            this.logger.debug(`Updating available work list`);
            workIds.shift();
            this.availableWork.next(workIds);
        }
    }

    private handleFailedWork() {
        this.workersController.failedWorkStream
            .pipe(delay(0))
            .subscribe((failedWork) => {
                this.logger.trace(`Work #${failedWork.id} failed.`);
                let workerOfWork =
this.mCurrentRunningWork.get(failedWork.id);
                if (!workerOfWork) {
```

```
                    this.logger.trace(`Work #${failedWork.id} not found in
current work list. Skip...`);
                    return;
                }
                this.mServerStatsController
                    .processingCompletedForWork(failedWork.id,
workerOfWork.id);
                this.mCurrentRunningWork.delete(failedWork.id);
                this.availableWork.value.push(failedWork.id);
                this.availableWork.next(this.availableWork.value);
            });
    }
}
```

Ο ResultHandler έχει γραφτεί ως αφηρημένη (abstract) κλάση, και αυτή χρησιμοποιεί ο προγραμματιστής για να έχει πρόσβαση στα τελικά αποτελέσματα. Υπάρχει η συνάρτηση workCompleted(workResult: WorkResult) η οποία καλείται κάθε φορά που ολοκληρώνεται η επεξεργασία κάποιας εργασίας. Με την βοήθεια του WorkScheduler περιμένει το σήμα του ότι όλες οι εργασίες ολοκληρώθηκαν και με την σειρά του καλεί τη συνάρτηση allCompletedHandleResults(allResults: WorkResult[]) για να ενημερώσει τον προγραμματιστή.

```
export abstract class WorkResultHandler {
    protected logger: Logger = Logger.gi();
    private mWorkResults: WorkResult[] = [];
    private mWorkScheduler: WorkScheduler;
    private mWorksSchedulerSub: Subscription | null;
    private mServerStatsController: ServerStatsController =
ServerStatsController.gi();

    public clearWorkResults() {
        this.mWorkResults = [];
    }

    public get workResults (): WorkResult[] {
        return this.mWorkResults;
    }

    public setWorkScheduler(workScheduler: WorkScheduler): void {
        this.mWorkScheduler = workScheduler;
        if (this.mWorksSchedulerSub) {
            this.mWorksSchedulerSub.unsubscribe();
            this.mWorksSchedulerSub = null;
```

```
        }
        this.mWorksSchedulerSub = this.mWorkScheduler
            .areAllWorksCompleted.pipe(
                filter((completed) => completed),
                take(1),
                delay(0)
            )
            .subscribe((completed) => {
                this.logger.info(`All works completed: ${completed}`);
                this.mServerStatsController.resultHandlingStarted();
                this.allCompletedHandleResults(this.mWorkResults);
                this.mServerStatsController.resultHandlingCompleted();
            });
    }

    public workCompleted(workResult: WorkResult) {
        this.logger.debug(`Handling result for:
${JSON.stringify(workResult)}`);
        this.mWorkResults.push(workResult);
    }

    public abstract allCompletedHandleResults(allResults: WorkResult[]):
void;

    public abstract desirializeWorkResult(workResultJson: any, work: Work):
WorkResult;
}
```

Τέλος, παρουσιάζονται η κλάση WorkGenerator και η διεπαφή WorkGenerationContract. Ο WorkGenerator είναι αυτός που χωρίζει την εργασία σε κομμάτια, για την ακρίβεια χρησιμοποιεί το WorkGenerationContract για να το κάνει. Μόλις ο Master είναι έτοιμος και ξεκινήσει την παραγωγή εργασιών registerWork() ο WorkGenerator καλεί τη μέθοδο WorkGenerationContract.generateWork(). Με αυτή την μέθοδο ο προγραμματιστής πρέπει να παράξει όλα τα κομμάτια εργασιών του πρώτου κύκλου, όλα όσα είναι γνωστά πριν την ολοκλήρωση της επεξεργασίας. Επίσης για κάθε μία εργασία θα πρέπει να καλέσει την μέθοδο WorkGenerator.submitWork(work: Work) ώστε να ενημερώσει και τον WorkGenerator. Αυτός με την σειρά του στέλνει τις εργασίες αυτές στον WorkScheduler σε ομάδες. Αν υπάρχουν και άλλες εργασίες να παραχθούν, δηλαδή δεν έχει καλέσει ο προγραμματιστής την συνάρτηση WorkGenerator.allWorkGenerationCompleted() τότε καλείται η συνάρτηση WorkGenerationContract.handleCompletedWorkResults(workResults:

WorkResultHandler). Από εδώ μπορούν να παραχθούν οι εργασίες για τον επόμενο κύκλο επεξεργασίας. Αυτό συνεχίζεται έως ότου δεν υπάρχουν άλλοι κύκλοι επεξεργασίας και υπολογιστεί το τελικό αποτέλεσμα.

```
export class WorkGenerator {
    ...
    constructor(workScheduler: WorkScheduler, workGenerationImpl:
WorkGenerationContract, workStorage: WorkStorage, workResultHandler:
WorkResultHandler) {
        this.mWorkScheduler = workScheduler;
        this.mWorkGenerationImpl = workGenerationImpl;
        this.mWorkStorage = workStorage;
        this.mWorkResultHandler = workResultHandler;
        this.mWorkGenerationImpl.attachWorkGenerator(this);
        this.signalWorkGenerationCompletionWhenDone();
        this.detectWhenWorkCirleIsCompletedAndNotifyForMoreWork();
    }

    public allWorkGenerationCompleted() {
        this.mAllWorkGenerationCompleted.next(true);
    }

    public registerWork(): void {
        if (this.mIsWorkGenerationStarted) {
            this.mLogger.info(`[WorkGenerator] Work already started.`);
            return;
        }
        this.mIsWorkGenerationStarted = true;
        this.mServerStatsController.workGenerationStarted();
        this.mWorkGenerationImpl.generateWork();
        this.mServerStatsController.workGenerationCompleted();
        this.sentWorkToSchedulerInBatches();
    }

    public submitWork(work: Work): void {
        this.mWorkStorage.addWork(work);
        this.mAllCurrentWorkCircleIdsToBeSentToWorkScheduler.push(work.id);
    }

    public workCompleted(work: Work, worker: Worker): void {
        this.mWorkScheduler.workCompleted(work, worker);
    }

    private detectWhenWorkCirleIsCompletedAndNotifyForMoreWork() {
        this.mWorkScheduler.areAllWorksCompletedForCurrentCirlce
            .pipe(
                filter((isCompleted) => isCompleted),
```

```
                takeUntil(this.mAllWorkGenerationCompleted.pipe(delay(0)))
            )
            .subscribe(() => {
                this.mLogger.debug(`[WorkGenerator] Current work circle
completed notifying for handling work results`);
                this.mServerStatsController.workGenerationStarted();
                let moreWorkAdded = this.mWorkGenerationImpl
                    .handleCompletedWorkResults(this.mWorkResultHandler);
                this.mServerStatsController.workGenerationCompleted();
                if (moreWorkAdded) {
                    this.mLogger.debug(`[WorkGenerator] More work added
sentWorkToSchedulerInBatches`);
                    this.sentWorkToSchedulerInBatches();
                }
            });
    }

    private sentWorkToSchedulerInBatches() {
        this.mServerStatsController.generatedWorkFeedStarted();
        this.mCurrentCircleWorkSupplyCompleted.next(false);
        this.mWorkScheduler.totalAvailableWork
            .pipe(
                debounceTime(0),
                takeUntil(this.mCurrentCircleWorkSupplyCompleted
                    .pipe(filter((isCompleted) => isCompleted)))
            )
            .subscribe((availableWork) => {
                this.mLogger.debug(`[WorkGenerator] Total generated work
available: ${availableWork}`);
                if (availableWork < this.GEN_THRESHOLD) {
                    this.mServerStatsController
                        .generatedWorkFeedBatchStarted();
                    this.mLogger.debug(`[WorkGenerator] Total generated work
available under threshold: ${this.GEN_THRESHOLD}`);
                    let totalSent = 0;
                    while
(this.mAllCurrentWorkCircleIdsToBeSentToWorkScheduler.length && totalSent++
< this.GEN_THRESHOLD) {
                        let aWorkId =
this.mAllCurrentWorkCircleIdsToBeSentToWorkScheduler.shift();
                        this.mWorkScheduler.addWork(aWorkId);
                    }

this.mServerStatsController.generatedWorkFeedBatchCompleted();
                    if
(!this.mAllCurrentWorkCircleIdsToBeSentToWorkScheduler.length) {
                        this.mCurrentCircleWorkSupplyCompleted.next(true);
                        this.mServerStatsController
                            .generatedWorkFeedCompleted();
```

```
                        this.mLogger.debug(`[WorkGenerator] No more work to
feed for this circle.`);
                    }
                }
            },
                () => { },
                () => {
                    this.mLogger.debug(`[WorkGenerator]
sentWorkToSchedulerInBatches completed`);
                });
    }

    private signalWorkGenerationCompletionWhenDone() {
        combineLatest(
            this.mAllWorkGenerationCompleted.pipe(
                delay(0),
                filter((isCompleted) => isCompleted),
                take(1)
            ),
            this.mCurrentCircleWorkSupplyCompleted
        ).pipe(
            filter((data) => data[1]),
            take(1)
        ).subscribe((data) => {
            this.mLogger.debug(`All work generation completed and sent to
work scheduler for all circles`, data);
            this.mServerStatsController.allWorkGenerationCompleted();
            this.mWorkScheduler.workGenerationAndSupplyCompleted();
        });
    }

export interface WorkGenerationContract {
    attachWorkGenerator(workGenerator: WorkGenerator): void;
    generateWork(): void;
    handleCompletedWorkResults(workResults: WorkResultHandler): boolean;
}
```

3.4 Διανομή εφαρμογής

Για τη διανομή της εφαρμογής έχει επιλεγεί ο διαχειριστής πακέτων Npm, με στόχο την απλοποίηση της δημιουργίας έργων (projects) για την επίλυση προβλημάτων, όπως περιγράφεται στο επόμενο κεφάλαιο. Με την χρήση του Webpack και του build:prod npm script παράγεται ο τελικός κώδικας της εφαρμογής και δημοσιεύεται στο μητρώο του Npm.

4 Επίλυση προβλημάτων με χρήση της εφαρμογής

Σε αυτό το κεφάλαιο χρησιμοποιούμε την εφαρμογή για να υλοποιήσουμε διάφορους αλγόριθμους που λύνουν κάποια γνωστά προβλήματα και είναι κατάλληλα για επεξεργασία σε κατανεμημένα συστήματα. Επιλέχθηκαν προβλήματα τα οποία χρησιμοποιούν έντονα την CPU αλλά και προβλήματα τα οποία έχουν και απαιτήσεις σε I/O. Μετά την υλοποίηση κάθε αλγορίθμου παραθέτουμε τα αποτελέσματα και τους χρόνους ειδικότερα για διάφορες περιπτώσεις.

4.1 Υπολογισμός του π με την μέθοδο Monte Carlo

Η μέθοδος αυτή προσομοιώνει την επιλογή τυχαίων σημείων μέσα σε ένα τετράγωνο με πλευρά 2r όπου μέσα σε αυτό βρίσκεται ένας κύκλος με ακτίνα r. Επιλέγοντας N τέτοια σημεία και ξέροντας ότι η αναλογία του εμβαδού του κύκλου ως προς το τετράγωνο είναι π/4 ($\pi r^2/4r^2$) μπορούμε να υπολογίσουμε τα M σημεία που θα πέσουν μέσα στον κύκλο (M=Nπ/4). Και τελικά να βρούμε το π=4M/N. Όπως είναι φανερό όσο μεγαλύτερο είναι το δείγμα N των τυχαίων σημείων τόσο αυξάνεται η ακρίβεια.

Η παραλληλοποίηση του αλγορίθμου δεν έχει ιδιαίτερες δυσκολίες. Μπορούμε να χωρίσουμε το σύνολο του δείγματος σε υποσύνολα, τα οποία θα εκτελεστούν παράλληλα. Δεν υπάρχουν εξαρτήσεις δεδομένων κατά την εκτέλεση και μόνο στο τέλος θα πρέπει να γίνει η συγκέντρωση των αποτελεσμάτων. Όπου και θα υπολογιστεί το π από την λύση του 4M/N.

4.1.1 Περιγραφή - Υλοποίηση

Όπως παρουσιάστηκε στο προηγούμενο κεφάλαιο για τη λύση οποιουδήποτε προβληματος, θα πρέπει να υλοποιηθούν τα εξής: η διεπαφή WorkGenerationContract, η κλάση WorkResultHandler και ο κώδικας που θα εκτελεστεί στους Web Workers.

4.1.1.1 WorkGenerationContract

Η διεπαφή (interface) αυτή υλοποιείται από την κλάση PiWorkGenerator και όπως βλέπουμε στον κώδικα παρακάτω, η συνάρτηση generateWork είναι αυτή που θα δημιουργήσει όλα τα κομμάτια εργασιών. Για αυτό το πρόβλημα χρειάζεται μόνο ένας κύκλος επεξεργασίας. Έτσι για κάθε ένα κομμάτι δημιουργούμε ένα αντικείμενο τύπου

PiWork όπου περιέχει τον αριθμό του δείγματος. Όλα τα σύνολα είναι του ίδιου μεγέθους αλλά υπάρχει περίπτωση το τελευταίο να έχει λιγότερα, αν το συνολικό δείγμα δεν διαιρείται ακριβώς από το υποσύνολο που θέλουμε.

```
export class PiWorkGenerator implements WorkGenerationContract {
    ...
    public generateWork(): void {
        this.logger.debug(`[PiWorkGeneratorContract] Generating work for
workers.`);
        let index = 0;
        new Array(this.totalChunksNeeded).fill(0).forEach(() => {
            let isExtraChunk = this.extraWorkerChunkSize
                && index === this.totalChunksNeeded - 1;
            if (isExtraChunk) {
                this.logger.debug(`[PiWorkGeneratorContract] Generating
extra work chunk`);
            }
            this.mWorkGenerator.submitWork(
                new PiWork(
                   `${index++}`,
                   isExtraChunk ? this.extraWorkerChunkSize : this.chunkSize
                )
            );
        });
        this.mWorkGenerator.allWorkGenerationCompleted();
    }
    ....
}
```

4.1.1.2 WorkResultHandler

Για τον υπολογισμό του τελικού αποτελέσματος η κλάση PiWorkResultHandler υλοποιεί τις αφηρημένες μεθόδους της WorkResultHandler και κυρίως αυτή που μας ενδιαφέρει είναι η allCompletedHandleResults. Σαν είσοδο της συνάρτησης έχουμε όλα τα αποτελέσματα των εργασιών και όπως βλέπουμε παρακάτω τα χρησιμοποιούμε για να υπολογίζουμε το π.

```
export class PiWorkResultHandler extends WorkResultHandler {
    public allCompletedHandleResults(workResults: PiWorkResult[]): void {
        this.logger.info(`All work completed. Total results:
${workResults.length}.`);
```

```
        let pi: number = 0;
        let totalRounds: number = 0; // XXX: Number.MAX_SAFE_INTEGER see
BigInt
        let totalScore: number = 0;
        workResults.forEach((result: PiWorkResult) => {
            totalRounds += result.rounds;
            totalScore += result.score;
        });
        pi = this.calculatePi(totalRounds, totalScore);
        this.logger.info(`Computed pi is '${pi}'`);
    }

    public desirializeWorkResult(workResultJson: any, work: Work):
WorkResult {
        return new PiWorkResult(work, workResultJson.score,
workResultJson.rounds);
    }

    private calculatePi(rounds: number, score: number) {
        return 4 * score/rounds;
    }
}
```

Αυτό που πρέπει να προσέξουμε εδώ είναι ότι οι αριθμοί μπορεί να είναι πολύ μεγάλοι και να υπάρξει κάποια υπερχείλιση, με αποτέλεσμα οι υπολογισμοί να είναι λάθος. Στο παράδειγμα που θα δούμε σε αυτό το κεφάλαιο δεν έχουμε τέτοιο πρόβλημα.

4.1.1.3 WebWorkerCode

Τέλος, πρέπει να υλοποιηθεί και ο κώδικας που θα εκτελεστεί στους Workers. Εκεί γίνεται η προσομοίωση της επιλογής των σημείων για το σύνολο του δείγματος της εργασίας. Σημαντικό είναι να τονιστεί ότι επειδή ο κώδικας εκτελείται σε Web Worker, η επικοινωνία με την κύρια εφαρμογή (που εκτελειται στο Main Thread) γίνεται με ανταλλαγή μηνυμάτων κάνοντας χρήση του API. Μόλις λάβει την εργασία ο Web Worker επιλέγει ένα τυχαίο σημείο, εξετάζει αν είναι μέσα στον κύκλο και αν ναι αυξάνει τον μετρητή score. Αυτό το επαναλαμβάνει για τόσες φορές όσο είναι το δείγμα. Μόλις ολοκληρώσει την προσομοίωση ενημερώνει την κύρια εφαρμογή με το αποτέλεσμα και αυτή το στέλνει με την σειρά της στο Master.

4.1.2 Αποτελέσματα

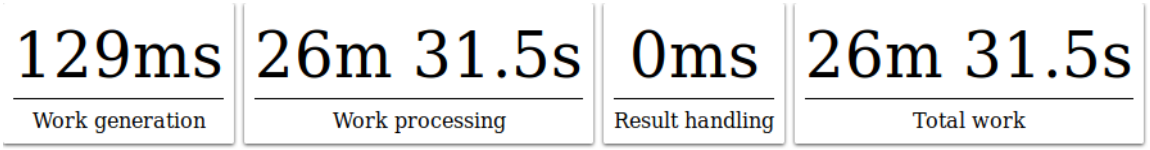
Για την δοκιμή της εφαρμογής χρησιμοποιήθηκε το cloud της Google για την εκτέλεση του Master, σε ένα υπολογιστή με 2 vCPUs και 13 GB μνήμη που βρισκόταν στη ζώνη europe-north1-a, κάπου στην Φινλανδία. Οι Workers ήταν δύο φορητοί υπολογιστές, ένας σταθερός, ένα tablet και ένα (έξυπνο) κινητό τηλέφωνο. Όλοι οι Workers ήταν συνδεδεμένοι στο ίδιο δίκτυο μέσω wifi εκτός του σταθερού που ήταν συνδεδεμένος μέσω ethernet. Η ταχύτητα του δικτύου είναι περίπου 50Mbps κατεβάσματος και 3Mbps ανεβάσματος. Σαν περιηγητής ιστού επιλέχθηκε ο Firefox καθώς σε διάφορες δοκιμές που έγιναν κατά την διάρκεια ανάπτυξης της εφαρμογής, παρατηρήθηκε ότι η διάρκεια επεξεργασίας της ίδιας εργασίας ήταν τουλάχιστον δύο φορές μικρότερη έναντι του Chrome.

Η δοκιμή έγινε χρησιμοποιώντας δείγμα N ίσο με 1000000000000 το οποίο χωρίστηκε σε 1000 κομμάτια εργασιών των 1000000000 δειγμάτων το καθένα. Η επεξεργασία διήρκησε 26 λεπτά 31.5 δευτερόλεπτα με αποτέλεσμα να βρούμε το π ίσο με 3.141593697764 που έχει μια ακρίβεια 5 δεκαδικών στοιχείων αν το συγκρίνουμε με το πραγματικό π = 3.14159265359 [28].

Ακολουθούν διάφορα στατιστικά στοιχεία για τον Master και τους Workers ξεκινώντας από τον Master και συνεχίζοντας με τους Workers με σειρά καλύτερης απόδοσης.

4.1.2.1 Master

Το μεγαλύτερο μέρος της εκτέλεσης όπως ήταν αναμενόμενο αποτέλεσε η επεξεργασία των αποτελεσμάτων.



Εικόνα 15: Υπολογισμός π - Master - διάρκεια κάθε φάσης της εκτέλεσης

Αυτή η συνολική διάρκεια περιλαμβάνει και τον χρόνο της μεταφοράς των δεδομένων κάθε εργασίας και των αποτελεσμάτων της στο δίκτυο και όπως φαίνεται στην παρακάτω εικόνα είναι πολύ μικρή. Ο συνολικός χρόνος του δικτύου είναι το άθροισμα όλων των χρόνων για κάθε εργασία σαν να γινόταν η εκτέλεση σειριακά.

1m 2.6s	63ms	51ms	680ms
Total	Average	Min	Max

Εικόνα 16: Υπολογισμός π - Master - στατιστικά στοιχεία δικτύου

Στην εικόνα που ακολουθεί μπορούμε να δούμε τον χρόνο που θα απαιτούσε η εκτέλεση του προβλήματος αν εκτελούνταν σειριακά. Είναι το άθροισμα της διάρκειας της επεξεργασίας κάθε εργασίας. Αυτός ο χρόνος περιλαμβάνει και τον χρόνο του δικτύου.

6h 39m 43.1s	24s	10.7s	3m 2.3s
Total	Average	Min	Max

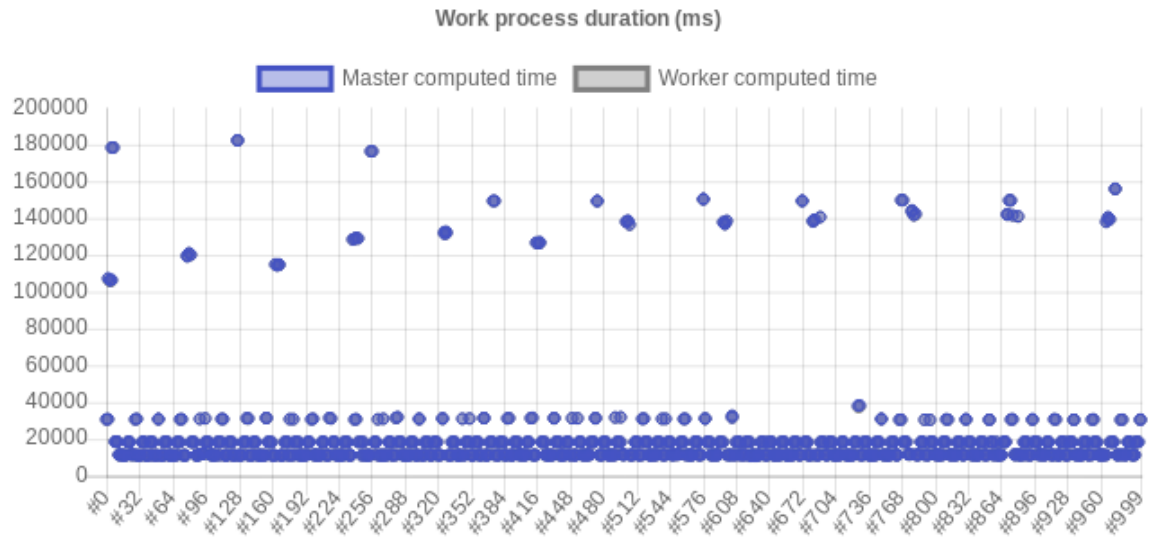
Εικόνα 17: Υπολογισμός π - Συνολική επεξεργασία συμπεριλαμβανομένου του χρόνου του δικτύου

Η επόμενη εικόνα δείχνει πάλι το άθροισμα της διάρκειας κάθε εργασίας αλλά δεν περιλαμβάνει το χρόνο που χρειάστηκε το δίκτυο για τη μεταφορά των δεδομένων.

6h 38m 40.4s	23.9s	10.7s	3m 2.3s
Total	Average	Min	Max

Εικόνα 18: Υπολογισμός π - Συνολική επεξεργασία χωρίς το χρόνο του δικτύου

Τέλος, βλέπουμε τη διάρκεια κάθε εργασίας όπως αυτή υπολογίστηκε από τον Master (περιλαμβάνει το χρόνο του δικτύου) αλλά και από τον κάθε Worker (καθαρός χρόνος επεξεργασίας).



Εικόνα 19: Υπολογισμός π - Διάρκεια επεξεργασίας κάθε εργασίας, από τον Master και τους Workers

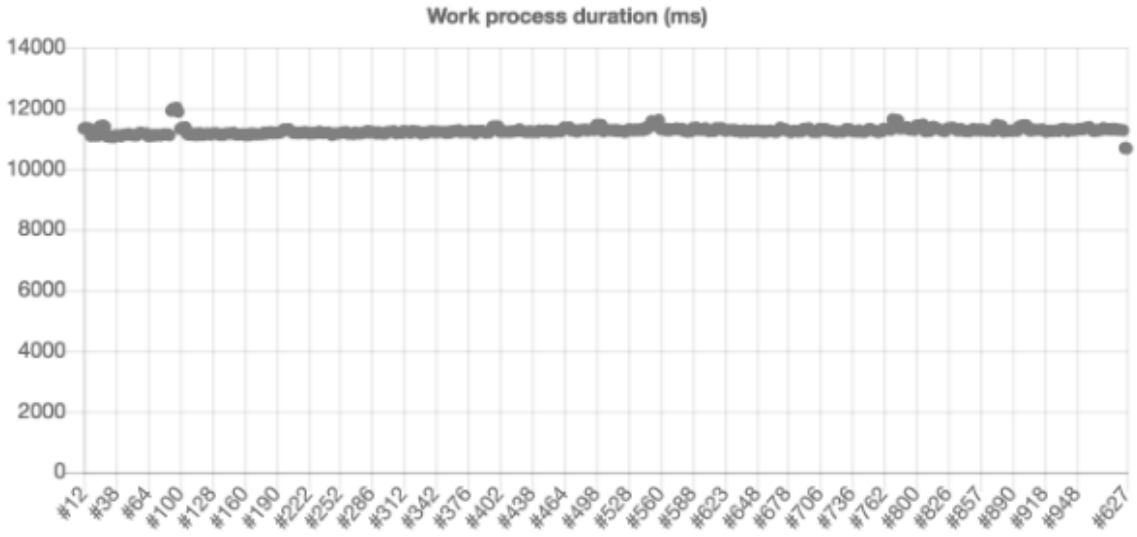
4.1.2.2 Workers

Ακολουθούν στατιστικά στοιχεία για κάθε Worker με σειρά καλύτερης απόδοσης.

4.1.2.2.1 Φορητός υπολογιστής Macbook Pro (Mac Os) 2.8GHz Intel Core i7 Quad Core

Έχοντας την καλύτερη απόδοση έγινε η επεξεργασία 522 εργασιών με μέσο όρο περίπου 11.5 δευτερόλεπτα. Οι 4 WebWorkers που χρησιμοποιήθηκαν είχαν σταθερή απόδοση καθόλη την διάρκεια της επεξεργασίας.

Εικόνα 20: Υπολογισμός π - φορητός υπολογιστής Macbook Pro - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών

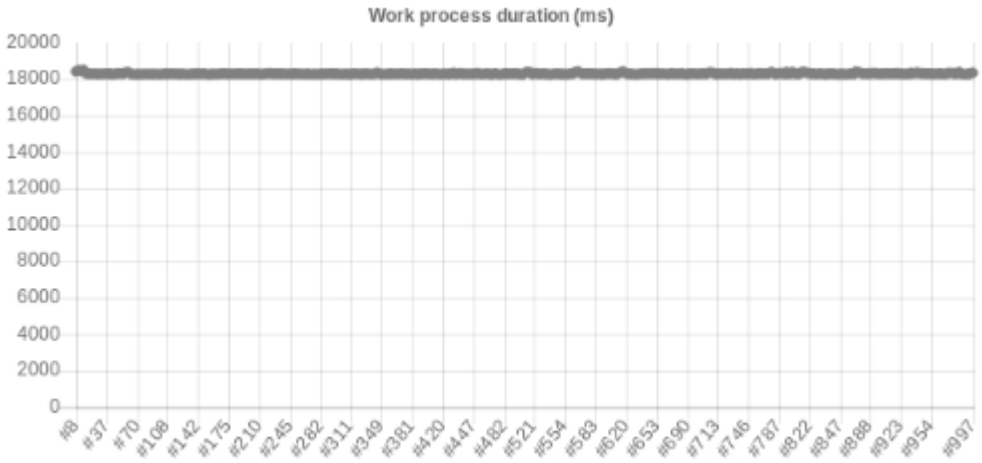


Εικόνα 21: Υπολογισμός π - φορητός υπολογιστής Macbook Pro - διάρκεια κάθε εργασίας

4.1.2.2.2 Σταθερός υπολογιστής (Linux) AMD FX(tm)-8350 Eight-Core Processor

Χρησιμοποιήθηκαν 4 WebWorkers και ολοκληρώθηκαν 324 εργασίες. Όπως φαίνεται και από το παρακάτω γράφημα ο χρόνος εκτέλεσης κάθε εργασίας ήταν σταθερός περίπου 18 δευτερόλεπτα.

Εικόνα 22: Υπολογισμός π – σταθερός υπολογιστής - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών

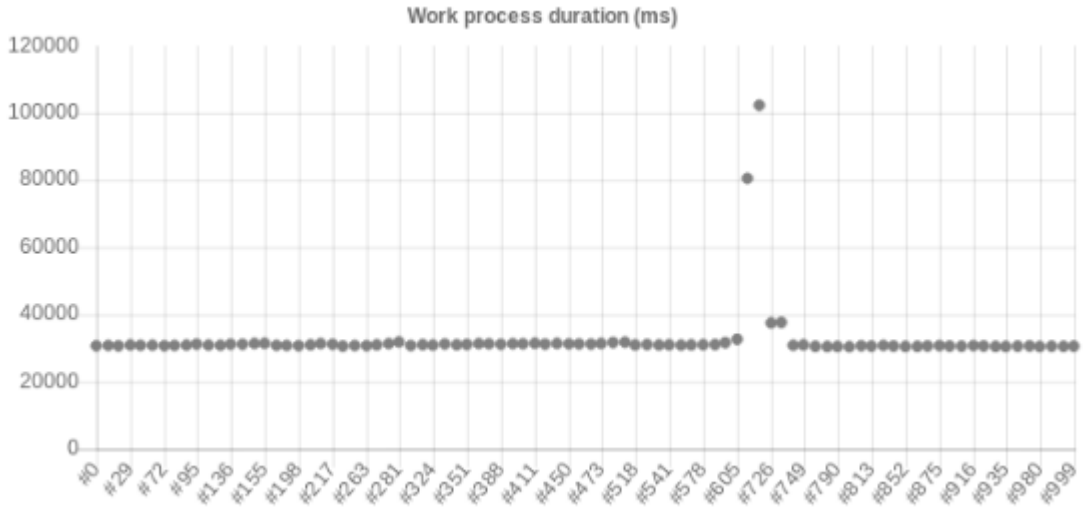


Εικόνα 23: Υπολογισμός π - σταθερός υπολογιστής - διάρκεια κάθε εργασίας

4.1.2.2.3 Φορητός υπολογιστής Asus (Linux) AMD A10-7400P Radeon R6, 10 Compute Cores 4C+6G

Χρησιμοποιήθηκαν 2 WebWorkers και ολοκληρώθηκαν 88 εργασίες. Η διάρκεια κάθε εργασίας ήταν περίπου 30 δευτερόλεπτα και ήταν σταθερή για όλη την διάρκεια της επεξεργασίας εκτός από ένα διάστημα που όπως φαίνεται ο υπολογιστής είχε επιπλέον υπολογιστικό φόρτο.

Εικόνα 24: Υπολογισμός π - φορητός υπολογιστής Asus - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 25: Υπολογισμός π - φορητός υπολογιστής Asus - διάρκεια κάθε εργασίας

4.1.2.2.4 Κινητό τηλέφωνο One Plus 2 (Android) Qualcomm MSM8994 Snapdragon 810 Octa-core

Έγινε η επεξεργασία 48 εργασιών από 4 WebWorkers με μέσο χρόνο επεξεργασίας περίπου 2 λεπτά. Βλέπουμε στο επόμενο γράφημα ότι η διάρκεια επεξεργασίας σταδιακά αυξανόταν. Πιθανόν αυτό να οφείλεται σε μείωση της απόδοσης του επεξεργαστή από το λειτουργικό λόγω αύξησης της θερμοκρασίας του κινητού.

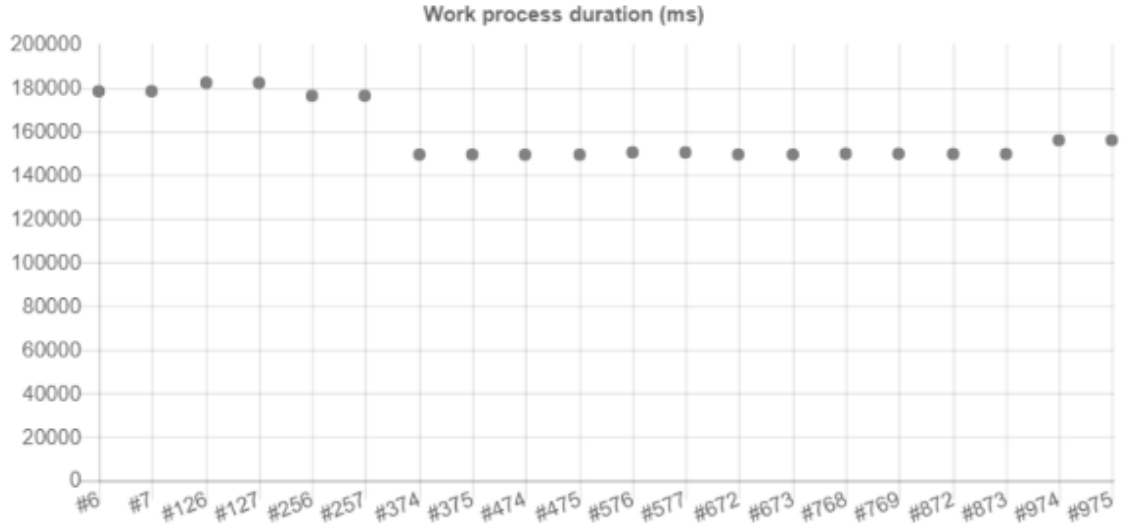
Εικόνα 26: Υπολογισμός π – κινητό τηλέφωνο One Plus 2 - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών

Εικόνα 27: Υπολογισμός π - κινητό τηλέφωνο One Plus 2 - διάρκεια κάθε εργασίας

4.1.2.2.5 Tablet Linx 12x64 (Windows) Intel Atom x5-Z8350 (Cherry Trail), Quad Core

Επεξεργάστηκαν 20 κομμάτια εργασιών με μέση διάρκεια επεξεργασίας περίπου λίγο κάτω από τα 3 λεπτά. Χρησιμοποιήθηκαν 2 WebWorker και είχαν σχετικά σταθερή απόδοση.

Εικόνα 28: Υπολογισμός π – tablet Linx 12x64 - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 29: Υπολογισμός π - tablet Linx 12x64 - διάρκεια κάθε εργασίας

4.2 Μέτρηση αριθμού εμφάνισης λέξεων σε αρχείο (Word Count)

Η μέτρηση της εμφάνισης λέξεων σε ένα αρχείο είναι ένα από τα πρώτα προβλήματα που βλέπει κάποιος, όταν ξεκινά την μελέτη του MapReduce χρησιμοποιώντας το Hadoop. Αυτό το πρόβλημα είναι λίγο διαφορετικό από τον υπολογισμό του π καθώς απαιτεί περισσότερους πόρους I/O, διάβασμα/γράψιμο σε αρχείο και μεταφοράς μεγαλύτερου όγκου δεδομένων. Η επεξεργασία σε κάθε ένα από τα κομμάτια εργασίας δεν έχει εξάρτηση από τα υπόλοιπα δεδομένα.

Για τη λύση του προβλήματος στο σύστημα του Hadoop κατά την φάση του Map εξάγεται από κάθε ένα κομμάτι μία λίστα από κλειδί/τιμή (λέξη/1) και αυτές οι λίστες κατά την διάρκεια του Reduce επεξεργάζονται και γίνεται η άθροιση των εμφανίσεων κάθε μιας λέξης.

4.2.1 Περιγραφή - Υλοποίηση

Η υλοποίηση και η λύση του προβλήματος με την χρήση της εφαρμογής γίνεται με παρόμοιο τρόπο, με την διαφορά ότι το αποτέλεσμα κάθε εργασίας που εκτελείται στους Workers περιέχει το άθροισμα των εμφανίσεων κάθε λέξης. Η λίστα κάθε αποτελέσματος περιέχει κάθε λέξη μόνο μία φορά. Σε αντίθεση με την υλοποίηση του Hadoop όπου κατά την διάρκεια του Map παράγονται λίστες του τύπου λέξη - 1.

Όπως και για τον υπολογισμό του π έτσι και για αυτό το πρόβλημα πρέπει να υλοποιήσουμε το WorkGeneratorContract, WorkResultHandler και τον κώδικα που θα εκτελεστεί στους Web Workers.

4.2.1.1 WorkGenerationContract

Η διεπαφή υλοποιείται από την κλάση WorkCountWorkGenerator όπου γίνεται η παραγωγή όλων των κομματιών εργασίας διαβάζοντας το αρχείο εισόδου και χωρίζοντας το, χρησιμοποιώντας το χαρακτήρα νέας γραμμής, σε λίστες με γραμμές. Οι λίστες αυτές έχουν μέγιστο μέγεθος τον αριθμό που έχουμε δώσει ως παράμετρο.

Η υλοποίηση για χάριν ευκολίας δεν χρησιμοποιεί τις ασύγχρονες μεθόδους πρόσβασης σε αρχεία, το οποίο έχει ως αποτέλεσμα ο Master να μην είναι προσβάσιμος κατά την διάρκεια της παραγωγής των εργασιών. Βέβαια αυτό έχει σχέση και με τον τρόπο που λειτουργεί ο WorkGenerator όπου ορίζεται η συνάρτηση generateWork ως σύγχρονη. Αυτό είναι όμως κάτι που μπορεί να βελτιωθεί στο μέλλον.

```
export class WorkCountWorkGenerator implements WorkGenerationContract {
    ....
    public generateWork(): void {
        this.logger.debug(`[WorkCountWorkGenerator] Generating work for
workers.`);
        try {
            let workIndex = 0;
            let linesChunk: string[] = [];
            let file = readFileSync(this.filePath, {encoding: "UTF-8"});
            file.split("\n").forEach((line) => {
                line = line.trim();
                if (line.length) {
                    linesChunk.push(line);
                }
                if (linesChunk.length === this.linesPerWork) {
                    this.logger.debug(`Work ready. Creating with chunk of:
${linesChunk.length} lines.`);
                    this.mWorkGenerator.submitWork(new
WorkCountWork(`${workIndex++}`, linesChunk));
                    linesChunk = [];
                }
            });
            if (linesChunk.length === this.linesPerWork) {
                this.logger.debug(`Some extra lines left. Creating work with
chunk of: ${linesChunk.length} lines.`);
                this.mWorkGenerator.submitWork(new
WorkCountWork(`${workIndex++}`, linesChunk));
                linesChunk = [];
            }
        } catch (error) {
            this.logger.error(`Read file error: ${error}`);
        }
```

```
        this.mWorkGenerator.allWorkGenerationCompleted();
    }
    ...
}
```

4.2.1.2 WorkResultHandler

Η κλάση WordCountWorkResultHandler υλοποιεί την αφηρημένη κλάση WorkResultHandler και αναλαμβάνει να διαχειριστεί τα αποτελέσματα. Έχοντας την λίστα των αποτελεσμάτων, γίνεται ο υπολογισμός της εμφάνισης της κάθε λέξης. Χρησιμοποιώντας ένα Map αποθηκεύουμε τον αριθμό των τρεχουσών εμφανίσεων ανά λέξη. Μόλις ολοκληρωθεί η επεξεργασία όλων τον αποτελεσμάτων το Map αυτό περιέχει τον τελικό αριθμό ανά λέξη. Το αποτέλεσμα αυτό το γράφουμε στο αρχείο outputFilepath με μορφή λέξη => αριθμός εμφανίσεων.

```
export class WordCountWorkResultHandler extends WorkResultHandler {
    ...
    public allCompletedHandleResults(workResults: WordCountWorkResult[]):
void {
        this.logger.info(`All work completed. Total results:
${workResults.length}.`);
        let wordsOccurancesMap: Map<string, number> = new Map();
        let word: string;
        let occurances: number;
        workResults.forEach((result: WordCountWorkResult) => {
            result.words.forEach(wordOccurances => {
                word = wordOccurances[0];
                occurances = wordsOccurancesMap.get(word) || 0;
                occurances += wordOccurances[1];
                wordsOccurancesMap.set(word, occurances)
            });
        });
        let fd: number = null;
        try {
            fd = openSync(this.outputFilepath, 'w');
            wordsOccurancesMap.forEach((occurances, word) => {
                writeSync(fd, `${word} => ${occurances}\n`);
            });
        } catch (error) {
            this.logger.error(`There was an error writting the output file.
${this.outputFilepath}`);
        } finally {
            if (fd !== null) {
```

```
                closeSync(fd);
            }
        }
        this.logger.info(`Word count completed. Total unique words:
'${wordsOccurancesMap.size}'. Check for output: ${this.outputFilepath}`);
    }
    ...
}
```

4.2.1.3 WebWorkerCode

Ο κώδικας που θα εκτελεστεί στους WebWorkers βρίσκεται στο αρχείο WebWorkerCode.ts. Όπως βλέπουμε παρακάτω για κάθε γραμμή χωρίζεται το κείμενο σε λέξεις με βάση τον κενό χαρακτήρα. Με παρόμοιο τρόπο όπως στην κλάση WordCountWorkResultHandler υπολογίζεται ο αριθμός εμφάνισης κάθε λέξης κάνοντας χρήση του wordOccuranciesMap. Τέλος, γίνεται η μετατροπή των αποτελεσμάτων σε λίστα με στοιχεία της μορφής [λέξη, αριθμός εμφανίσεων].

```
onmessage = function(workEvent) {
    var work = workEvent.data;
    var lines = work.lines;
    var wordOccuranciesMap = {};
    var wordsWithOccurancies = [];
    var counter;

    lines.forEach(function(line) {
        line.trim().split(" ").forEach(function(aWord) {
            aWord = aWord.trim();
            if (aWord.length) {
                counter = wordOccuranciesMap[aWord] || 0;
                counter++;
                wordOccuranciesMap[aWord] = counter;
            }
        });
    });

    Object.keys(wordOccuranciesMap).forEach(function(aWord) {
        wordsWithOccurancies.push([aWord, wordOccuranciesMap[aWord]]);
    });

    postMessage({
        data: {
            wordsOccurances: wordsWithOccurancies
```

```
        },
        work: work
    });
}
```

4.2.2 Αποτελέσματα

Για τη δοκιμή της εφαρμογής χρησιμοποιήθηκε ένα αρχείο το οποίο αποτελούνταν από βιβλία ενωμένα μεταξύ τους σε μορφή απλού κειμένου. Πηγή αυτών των βιβλίων είναι η ιστοσελίδα www.gutenberg.org. Το μέγεθος του αρχείου ήταν περίπου 100 MB. Για να έχουμε όμως ένα αρκετά μεγάλο αρχείο μεγέθους το αρχείο αυτό χρησιμοποιήθηκε για να παραχθεί ένα τελικό μεγέθους 1000 MB που περιέχει 10 φορές το αρχικό.

Από την εκτέλεση της επεξεργασίας στην ίδια υποδομή, χωρίς την χρήση του tablet, έγινε φανερό ότι το πρόβλημα ήταν πολύ απαιτητικό σε πόρους δικτύου. Καθώς όλοι οι workers βρίσκονται στο ίδιο δίκτυο και ο Master σε άλλο απομακρυσμένο αυτό δυσχεραίνει ακόμη περισσότερο τη μεταφορά των δεδομένων. Κάθε ένα από τα κομμάτια εργασίας περιείχαν 200000 γραμμές παράγοντας 73 κομμάτια.

Η εκτέλεση είχε ως αποτέλεσμα την παραγωγή ενός αρχείου μεγέθους 8.8MB που αποτελούνταν από τον αριθμό εμφανίσεων των 566694 μοναδικών λέξεων. Ο συνολικός χρόνος εκτέλεσης ήταν 3 λεπτά και 31.6 δευτερόλεπτα. Ακολουθούν διάφορα στατιστικά στοιχεία.

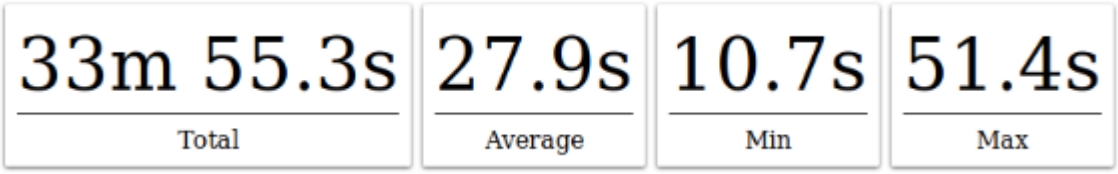
4.2.2.1 Master

Η παραγωγή των κομματιών εργασίας αλλά και η τελική επεξεργασία όλων των αποτελεσμάτων είχαν υπολογίσιμη διάρκεια. Κάτι που δείχνει ότι η επεξεργασία και στο Master μπορεί να είναι απαιτητική ανάλογα με το πρόβλημα.

Εικόνα 30: Word Count - Master - διάρκεια κάθε φάσης της εκτέλεσης

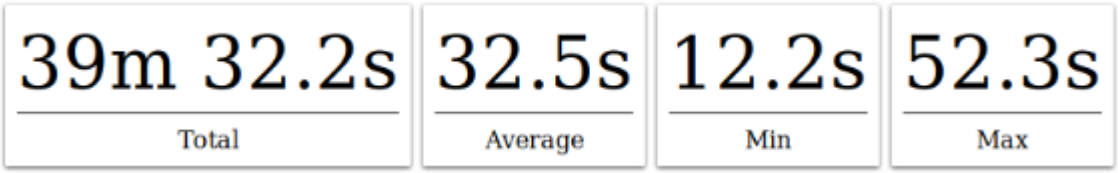
Αυτή η συνολική διάρκεια περιλαμβάνει και τον χρόνο της μεταφοράς των δεδομένων κάθε εργασίας και των αποτελεσμάτων της στο δίκτυο και όπως φαίνεται στην

παρακάτω εικόνα είναι αξιοσημείωτη. Ο συνολικός χρόνος του δικτύου είναι το άθροισμα όλων των χρόνων για κάθε εργασία σαν να γινόταν η εκτέλεση σειριακά.



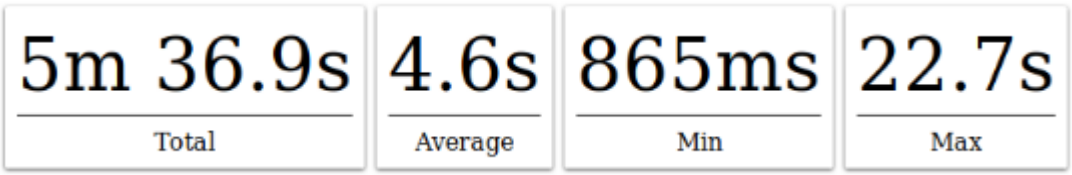
Εικόνα 31: Word Count - Master - στατιστικά στοιχεία δικτύου

Στην εικόνα που ακολουθεί μπορούμε να δούμε τον χρόνο που θα απαιτούσε η εκτέλεση του προβλήματος αν εκτελούνταν σειριακά. Είναι το άθροισμα της διάρκειας της επεξεργασίας κάθε εργασίας. Αυτός ο χρόνος περιλαμβάνει και τον χρόνο του δικτύου.



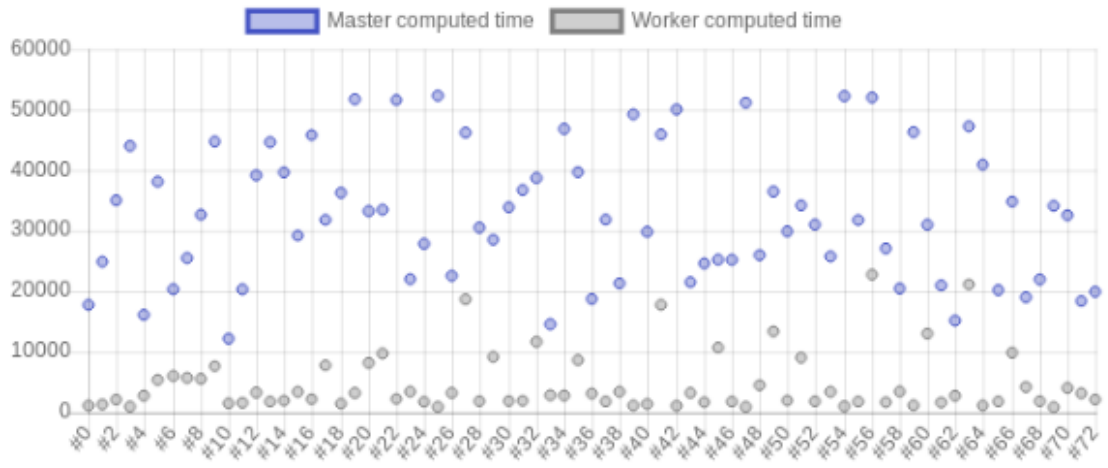
Εικόνα 32: Word Count - Συνολική επεξεργασία συμπεριλαμβανομένου του χρόνου του δικτύου

Η επόμενη εικόνα δείχνει πάλι το άθροισμα της διάρκειας κάθε εργασίας αλλά δεν περιλαμβάνει τον χρόνο που χρειάστηκε το δίκτυο για την μεταφορά των δεδομένων.



Εικόνα 33: Word Count - Συνολική επεξεργασία χωρίς το χρόνο του δικτύου

Τέλος, βλέπουμε τη διάρκεια κάθε εργασίας όπως αυτή υπολογίστηκε από τον Master (περιλαμβάνει το χρόνο του δικτύου) αλλά και από τον κάθε Worker (καθαρός χρόνος επεξεργασίας). Εδώ είναι πολύ φανερή η μεγάλη χρήση των πόρων του δικτύου.



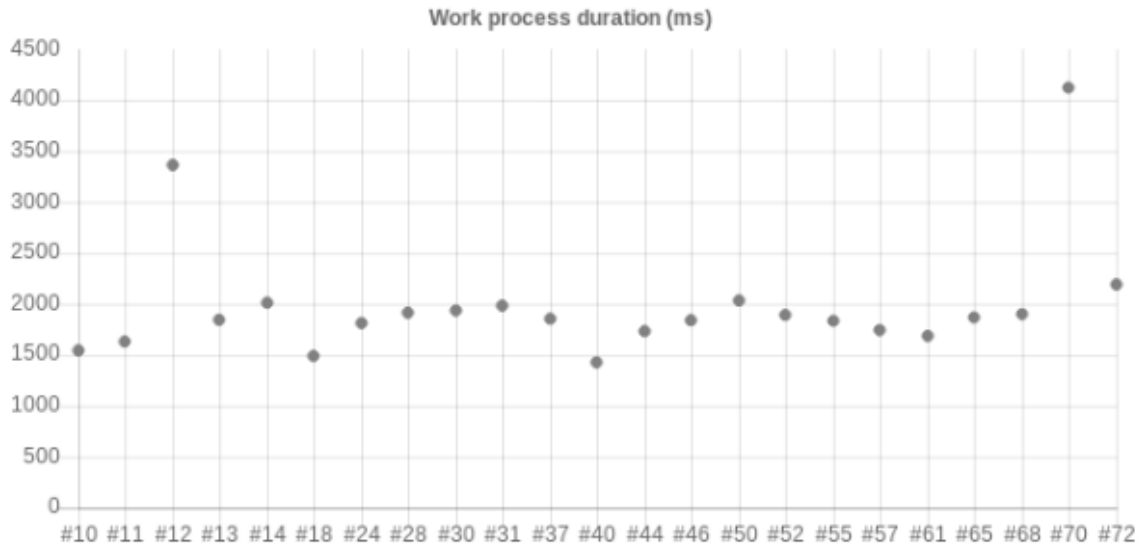
Εικόνα 34: Word Count - Διάρκεια επεξεργασίας κάθε εργασίας, από τον Master και τους Workers

4.2.2.2 Workers

Ακολουθούν στατιστικά στοιχεία για κάθε Worker με σειρά καλύτερης απόδοσης. Σε όλους τους Workers η επεξεργασία των κομματιών των εργασιών είχε σχετικά την ίδια διάρκεια εκτός από το κινητό τηλέφωνο One Plus 2 που η μέση διάρκεια κάθε εργασίας ήταν περίπου 15 δευτερόλεπτα. Βλέπουμε ότι η χρήση των πόρων του δικτύου έχει μεγαλύτερη σημασία για το συγκεκριμένο πρόβλημα. Καθώς ο σταθερός υπολογιστής είχε την καλύτερη απόδοση αφού ήταν συνδεδεμένος στο δίκτυο ενσύρματα μέσω ethernet.

4.2.2.2.1 Σταθερός υπολογιστής (Linux) AMD FX(tm)-8350 Eight-Core Processor

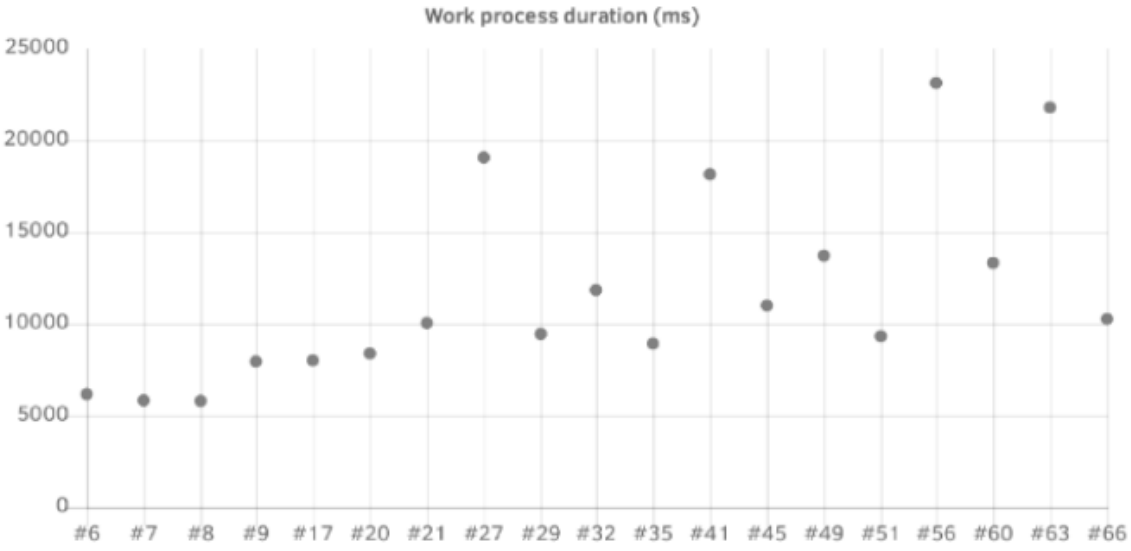
Εικόνα 35: Word Count – σταθερός υπολογιστής - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 36: Word Count - σταθερός υπολογιστής - διάρκεια κάθε εργασίας

4.2.2.2.2 Κινητό τηλέφωνο One Plus 2 (Android) Qualcomm MSM8994 Snapdragon 810 Octa-core

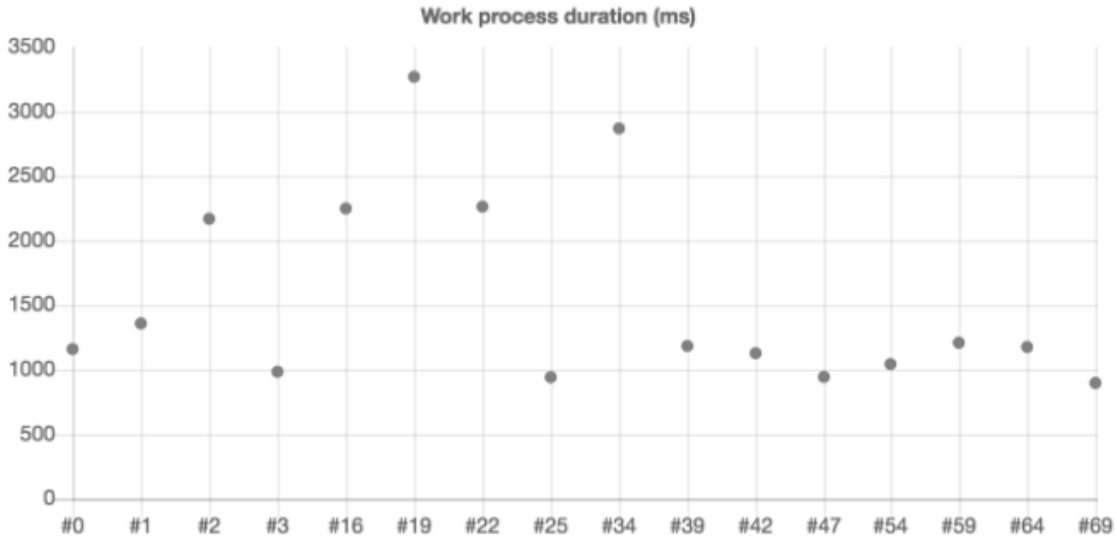
Εικόνα 37: Word Count – κινητό τηλέφωνο One Plus 2 - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 38: Word Count - κινητό τηλέφωνο One Plus 2 - διάρκεια κάθε εργασίας

4.2.2.2.3 Φορητός υπολογιστής Macbook Pro (Mac Os) 2.8GHz Intel Core i7 Quad Core

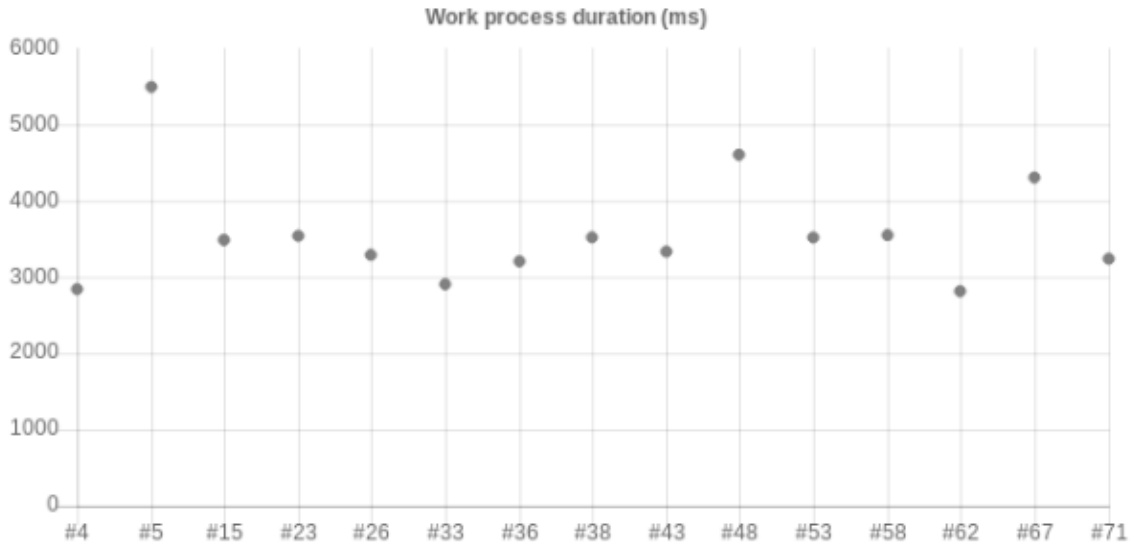
Εικόνα 39: Word Count - φορητός υπολογιστής Macbook Pro - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 40: Word Count - φορητός υπολογιστής Macbook Pro - διάρκεια κάθε εργασίας

4.2.2.2.4 Φορητός υπολογιστής Asus (Linux) AMD A10-7400P Radeon R6, 10 Compute Cores 4C+6G

Εικόνα 41: Word Count - φορητός υπολογιστής Asus - αριθμός πυρήνων, διεργασιών και σύνολο ολοκληρωμένων εργασιών



Εικόνα 42: Word Count - φορητός υπολογιστής Asus - διάρκεια κάθε εργασίας

5 Επίλογος

5.1 Σύνοψη και συμπεράσματα

Η παρούσα διπλωματική εργασία εξετάζει τη χρήση τεχνολογιών ιστού, και ειδικότερα Node.js και WebWorkers για τη δημιουργία ενός συστήματος στο οποίο θα είναι δυνατή η παράλληλη επεξεργασία. Η μελέτη αυτή στηρίχθηκε στην ανάπτυξη μιας βιβλιοθήκης (framework) η οποία χρησιμοποιήθηκε ώστε να λυθούν διάφορα προβλήματα και να εξεταστούν τα αποτελέσματα τους.

Για την ανάπτυξη αυτής της βιβλιοθήκης μελετήθηκαν διάφορες παραπλήσιες εφαρμογές, όπως το BOINC, το Bayanihan και το MapReduce. Από τη μελέτη αυτή έγινε η επιλογή του μοντέλου αρχιτεκτονικής Master-Worker. Η χρήση δηλαδή μιας κεντρικής οντότητας (Master) η οποία θα διαχειρίζεται όλες τις μονάδες επεξεργασίας (Workers), θα κατανέμει σε αυτόνομα κομμάτια την εργασία και τέλος θα την διανέμει για επεξεργασία και θα συγκεντρώνει τα τελικά αποτελέσματα. Αυτή η επιλογή βοηθά στην απλοποίηση του προγραμματιστικού μοντέλου και στην ευκολότερη επίλυση προβλημάτων κατανεμημένης επεξεργασίας, όπως ανίχνευσης λαθών και ανοχής σε σφάλματα.

Επίσης, από τη μελέτη αυτών των εφαρμογών εμφανίστηκε η ανάγκη να απλοποιηθεί η διαδικασία υποβολής ενός νέου προβλήματος προς επίλυση αλλά και η διευκόλυνση της εισαγωγής νέων μονάδων επεξεργασίας στο σύστημα. Τα συστήματα που μελετήθηκαν χρησιμοποιούν διάφορες γλώσσες προγραμματισμού όπως Java και C, σε αντίθεση με την εφαρμογή της παρούσας εργασίας όπου επιλέχθηκε η χρήση της γλώσσας προγραμματισμού JavaScript.

Η επιλογή αυτή μας επιτρέπει αρχικά να απλοποιήσουμε τη διαδικασία υλοποίησης μιας λύσης ενός προβλήματος, καθώς η JavaScript είναι μια ευρέως διαδεδομένη γλώσσα και υπάρχει μία πληθώρα βιβλιοθηκών εύκολα προσβάσιμων με τον διαχειριστή πακέτων Npm, οι οποίες μπορούν να χρησιμοποιηθούν κατά την ανάπτυξη του κώδικα. Επίσης, μας επιτρέπει να χρησιμοποιήσουμε το Node.Js για τη λειτουργία του Master και τους περιηγητές ιστού και πιο συγκεκριμένα τους Web Workers για την εκτέλεση της επεξεργασίας (Workers) προγραματίζοντας μόνο σε μία γλώσσα.

Με τη χρήση περιηγητών ιστού διευκολύνεται η διαδικασία εισαγωγής νέων μονάδων επεξεργασίας καθώς μπορεί να γίνει πολύ εύκολα με την πλοήγηση κάποιου χρήστη σε μια ιστοσελίδα σε αντίθεση με τις εφαρμογές GIMPS, BOINC κλπ. που

απαιτούν εγκατάσταση κάποιας εφαρμογής. Η χρήση βέβαια των περιηγητών γίνεται και από την εφαρμογή Bayanihan που όμως χρησιμοποιεί την τεχνολογία Java Applet κάτι που περιορίζει τις συσκευές που μπορούν να υποστηριχθούν. Σε αντίθεση, οι Web Workers υποστηρίζονται ευρέως και μπορούν να λειτουργήσουν σε υπολογιστές αλλά και φορητές συσκευές όπως έξυπνα τηλέφωνα και tablets.

Τα παραπάνω έγιναν φανερά κατά την χρήση της βιβλιοθήκης (framework) όταν χρησιμοποιήθηκε για την επίλυση δύο προβλημάτων, τον υπολογισμό του π με την μέθοδο Monte Carlo και τη μέτρηση εμφάνισης λέξεων μέσα σε ένα αρχείο (Word Count). Ήταν σχετικά εύκολη η υλοποίηση των προβλημάτων αλλά και για την εκτέλεση της επεξεργασίας η απλή πλοήγηση σε μια ιστοσελίδα διευκόλυνε την διαδικασία εισαγωγής διαφόρων συσκευών στο σύστημα. Η πληκτρολόγηση ενός URL ή ακόμη και το σκανάρισμα ενός QR Code είναι αρκετό για ξεκινήσει η επεξεργασία.

Κατά την εκτέλεση των δύο προβλημάτων παρατηρήθηκε ότι με την επιλογή των Web Workers λόγω ότι λειτουργούν ανεξάρτητα από την εκτέλεση του κυρίως προγράμματος δεν επηρεάζουν την αλληλεπίδραση του χρήστη με το περιεχόμενο της ιστοσελίδας που επισκέπτονται. Αυτό έχει σαν αποτέλεσμα να μπορεί να γίνει η επεξεργασία και κατά την διάρκεια που ο χρήστης πλοηγείται στο διαδίκτυο.

Τέλος, από τα αποτελέσματα της επεξεργασίας αυτών των προβλημάτων γίνεται φανερό ότι προβλήματα τα οποία απαιτούν κυρίως επεξεργαστική ισχύ και η τελική επεξεργασία μετά την συγκέντρωση των αποτελεσμάτων δεν είναι απαιτητική για τον Master, όπως ο υπολογισμός του π είναι ιδανικά ώστε να παραλληλοποιηθούν με την χρήση της παρούσας βιβλιοθήκης. Αυτό φαίνεται από τον συνολικό χρόνο που χρειάστηκε η επίλυση του προβλήματος υπολογισμού του π, όπου διήρκησε περίπου είκοσι έξι λεπτά ενώ αν η επεξεργασία γινόταν σειριακά θα διαρκούσε περίπου έξι ώρες και σαράντα λεπτά. Από την άλλη μεριά προβλήματα τα οποία είναι περισσότερο απαιτητικά σε πόρους I/O (όγκος δεδομένων - δίκτυο) και απαιτούν περισσότερη επεξεργαστική ισχύ κατά την τελική επεξεργασία στο Master, σαν αυτό της καταμέτρησης λέξεων ενός αρχείου, μπορούν να λυθούν με τεχνολογίες ιστού, όπως και το παρόν σύστημα, αλλά παρουσιάζουν προκλήσεις ως προς την διαχείριση του όγκου των δεδομένων. Από τα αποτελέσματα της επεξεργασίας του προβλήματος βλέπουμε ότι παρουσιάστηκε μία επιτάχυνση στο συνολικό χρόνο επεξεργασίας που ήταν περίπου τρία λεπτά και τριάντα δευτερόλεπτα. Ενώ αν η επεξεργασία γινόταν σειριακά θα διαρκούσε περίπου πέντε λεπτά

και τριάντα δευτερόλεπτα. Όμως η χρήση των πόρων του δικτύου που κατά μέσο όρο ήταν περίπου τριάντα δευτερόλεπτα για την μεταφορά της κάθε εργασίας και των αποτελεσμάτων της, είχε σαν αποτέλεσμα τη μεγάλη επιβάρυνση του Master κάτι που μας εφιστά την προσοχή για τυχόν βελτιώσεις στο μέλλον.

5.2 Όρια και περιορισμοί

Θεωρώντας ότι η χρήση της εφαρμογής δεν θα γίνεται από κακόβουλους χρήστες, έχει παραληφθεί η εξέταση αλλά και η υλοποίηση τεχνικών ώστε να προστατευτεί το σύστημα από χρήστες οι οποίοι στέλνουν λανθασμένα αποτελέσματα. Θα μπορούσε να λυθεί με διάφορες τεχνικές όπως η βαθμολόγηση του Worker που στέλνει τη λύση, η λύση του ίδιου κομματιού εργασίας από πολλούς ώστε να γίνει δυνατή η επαλήθευση, κ. α.

Επίσης, για λόγους οικονομίας χρόνου έγιναν διάφορες άλλες παραβλέψεις κυρίως στην ανάπτυξη του κώδικα, σε σχέση με θέματα διαχείρισης μνήμης. Τέλος, για τους ίδιους λόγους η δοκιμή της εφαρμογής έγινε σε σχετικά μικρά προβλήματα και από περιορισμένο αριθμό συνδεδεμένων Workers.

5.3 Μελλοντικές Επεκτάσεις

Πέρα από τις βελτιώσεις που μπορούν να γίνουν στην εφαρμογή αλλά και την υλοποίηση ενός συστήματος επιβεβαίωσης των αποτελεσμάτων, θα μπορούσε να αναπτυχθεί μία άλλη εφαρμογή η οποία θα στηριχθεί στον κώδικα της παρούσας με σκοπό να προσφέρει την υπολογιστική ισχύ του δικτύου σαν υπηρεσία. Να μπορεί κάποιος δηλαδή χρησιμοποιώντας μια διαδικτυακή εφαρμογή να υποβάλει το δικό του πρόβλημα και το σύστημα να αναλαμβάνει όλα τα υπόλοιπα. Μία τέτοια εφαρμογή έχει κι άλλες προκλήσης όπως η διαχείριση της υποδομής (infrastructure) στην οποία θα εκτελείται η εφαρμογή. Επίσης δημιουργεί την ανάγκη ελέγχου του κώδικα τον οποίο έχει υποβάλει για εκτέλεση κάποιος χρήστης ώστε να μην είναι επιβλαβής για το σύστημα αλλά και να αποφευχθούν κακόβουλες ενέργειες όπως κατανεμημένες επιθέσεις άρνησης εξυπηρέτησης (DDoS).

Βιβλιογραφία - Αναφορές

[1] George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair, 2012. DISTRIBUTED SYSTEMS Concepts and Design 5th Edition

[2] Andrew S.Tanenbaum, Maarten Van Steen, 2007. Distributed Systems: Principles and Paradigms 2nd Edition

[3] Ajay D. Kshemkalyani, Mukesh Singhal, 2008. Distributed Computing Principles, Algorithms, and Systems

[4] Salim Hariri, Manish Parashar, 2004. Tools and Environments for Parallel and Distributed Computing

[5] Luis F. G. Sarment, 2001. Volunteer Computing. PhD Thesis at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[6] Mersenne Research, Inc., 1996-2018. GIMPS History [online] Available at: <https://www.mersenne.org/various/history.php> [Accessed August 2018]

[7] distributed.net. Main Page [online] Available at: <https://www.distributed.net/Main_Page> [Accessed August 2018]

[8] distributed.net. History Available at: <https://www.distributed.net/History> [Accessed August 2018]

[9] BOINC. User Manual [online] Available at: <https://boinc.berkeley.edu/wiki/User_manual> [Accessed August 2018]

[10] Wikipedia. Berkeley Open Infrastructure for Network Computing [online] Available at: <https://en.wikipedia.org/wiki/Berkeley_Open_Infrastructure_for_Network_Computing> [Accessed August 2018]

[11] Node.js. About [online] Available at: <https://nodejs.org/en/about/> [Accessed August 2018]

[12] Wikipedia. Node.js [online] Available at: <https://en.wikipedia.org/wiki/Node.js> [Accessed August 2018]

[13] World Wide Web Consortium (W3C), 2015. Web Workers [online] Available at: <https://www.w3.org/TR/workers/> [Accessed August 2018]

[14] caniuse.com. Web Workers [online] Available at: <https://caniuse.com/#search=web%20workers> [Accessed August 2018]

[15] Socket.io. Documentation [online] Available at: <https://socket.io/docs/> [Accessed August 2018]

[16] angular.io. Angular Documentation [online] Available at: <https://angular.io/docs> [Accessed August 2018]

[17] Wikipedia. Angular (application platform) [online] Available at: <https://en.wikipedia.org/wiki/Angular_(application_platform)> [Accessed August 2018]

[18] webpack.js. Documentation [online] Available at: <https://webpack.js.org/concepts/> [Accessed August 2018]

[19] typescriptlang.org. Documentation [online] Available at: <https://www.typescriptlang.org/docs/home.html> [Accessed August 2018]

[20] Wikipedia. TypeScript [online] Available at: <https://en.wikipedia.org/wiki/TypeScript> [Accessed August 2018]

[21] npmjs.com. Npm Documentation [online] Available at: <https://docs.npmjs.com/getting-started/what-is-npm> [Accessed August 2018]

[22] Wikipedia. npm (software) [online] Available at: <https://en.wikipedia.org/wiki/Npm_(software)> [Accessed August 2018]

[23] Jeffrey Dean and Sanjay Ghemawat, 2008. *MapReduce: Simplified Data Processing on Large Clusters*

[24] hadoop.apache.org. Main Page [online] Available at: <http://hadoop.apache.org/> [Accessed August 2018]

[25] hadoop.apache.org. HDFS Architecture [online] Available at: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> [Accessed August 2018]

[26] chartjs.org. Chart.js Documentation [online] Available at: <http://www.chartjs.org/docs/latest/> [Accessed August 2018]

[27] Npm. rxjs [online] Available at: <https://www.npmjs.com/package/rxjs> [Accessed August 2018]

[28] Wikipedia. Pi [online] Available at: <https://en.wikipedia.org/wiki/Pi> [Accessed August 2018]