ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Τα πλεονεκτήματα του Cloud Computing και της Αρχιτεκτονικής Function as a Service, και οι εφαρμογές τους στο Internet of Things

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΖΑΧΑΡΙΑΣ Β. ΚΩΣΤΟΠΟΥΛΟΣ

Επιβλέπων : Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.

Αθήνα, Οκτώβριος 2018

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ
ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Τα πλεονεκτήματα του Cloud Computing και της Αρχιτεκτονικής Function as a Service, και οι εφαρμογές τους στο Internet of Things

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΚΩΣΤΟΠΟΥΛΟΣ Β. ΖΑΧΑΡΙΑΣ

Επιβλέπων : Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10η Οκτωβρίου 2018.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Θ. Βαρβαρίγου	Ε. Βαρβαρίγος	Δ. Ασκούνης
Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2018

(Υπογραφή)

.....

ΖΑΧΑΡΙΑΣ ΚΩΣΤΟΠΟΥΛΟΣ

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

Περίληψη

Το Internet of Things επεκτείνεται ραγδαία και εμφανίζεται πλέον καθημερινά στη ζωή μας. Ο όγκος των συσκευών και των δεδομένων που καλούνται οι προγραμματιστές να διαχειριστούν αυξάνεται συνεχώς. Δημιουργείται σταδιακά η ανάγκη για νέες τεχνολογίες οι οποίες θα μπορούν να δώσουν ταχύτερα αποτελέσματα και θα διευκολύνουν τους προγραμματιστές.

Λύση στα προβλήματα του IoT έρχεται να δώσει η Function as a Service (FaaS) αρχιτεκτονική, η οποία παρέχει τη δυνατότητα για event driven προγραμματισμό, με την έννοια ότι κάποιο γεγονός αποτελεί το έναυσμα για κάποια ενέργεια, δηλαδή την εκτέλεση μιας λειτουργίας. Η FaaS αρχιτεκτονική, σε συνδυασμό με το Cloud Computing, όπου δηλαδή όλα τα εξαρτήματα μίας εφαρμογής εκτελούνται στο cloud, παρέχουν σημαντικές διευκολύνσεις. Σημαντικό, επίσης, ρόλο έρχεται να παίξει και η αρχιτεκτονική των microservices, όπου, σύμφωνα με την οποία κάθε εφαρμογή αποτελείται από μικρές ανεξάρτητες λειτουργίες που εκτελούνται αυτόνομα.

Όλες αυτές τις πολλά υποσχόμενες τεχνολογίες, έρχονται να συνδυάσουν τα διάφορα orchestration tools, όπως ο Netflix Conductor, το Node-RED και το Openwhisk.

Στην παρούσα διπλωματική εργασία θα διερευνήσουμε την συμβολή αυτών των τεχνολογιών στο IoT. Θα πραγματοποιήσουμε έρευνα ώστε να εντοπίσουμε τα πλεονεκτήματα που μας παρέχουν, ως προς το χρόνο και την εκτέλεση μίας εφαρμογής.

Λέξεις Κλειδιά: Internet of Things, Function as a Service, containers, microcontainers, cloud computing, μονολιθική αρχιτεκτονική, microservices, orchestration tools, AWS Lambda, Netflix Conductor, node-RED, Openwhisk

Abstract

The Internet of Things is being rapidly developed and it is apparent in our everyday life. The amount of the devices and the data that the developers have to handle raises constantly. Because of that, the need of new technologies which will be able to execute actions faster is emerging.

The Function as a Service architecture seems to solves the current IoT problems, which provides us the ability of event driven program development, in the sense that an event can be the trigger of an action. FaaS architecture, combined with cloud computing, in which all the components of an application are executed in the cloud, make everything easier. Moreover, the architecture of microservices, according to which every application is composed of small functions that are executed independently, is also of great importance.

All of these state-of-the-art new technologies come combined with the orchestration tools, like Netflix Conductor, node-RED and Openwhisk.

In this diplomatic thesis we will research the contribution of these technologies in the IoT. We will conduct a research so we can identify the benefits they provide to us, depending on the time and the execution of an application.

Keywords: Internet of Things, Function as a Service, containers, microcontainers, cloud computing, monolithic architecture, microservices, orchestration tools, AWS Lambda, Netflix Conductor, node-RED, Openwhisk

Ευχαριστίες

Καταρχάς, θα ήθελα να ευχαριστήσω την επιβλέπουσα καθηγήτρια Θεοδώρα Βαρβαρίγου που μου ανέθεσε αυτή την εργασία, δίνοντάς μου τη δυνατότητα να έρθω σε επαφή με νέες τεχνολογίες και να διευρύνω τις γνώσεις μου.

Επίσης, ευχαριστώ θερμά τον υποψήφιο Διδάκτορα του Ε.Μ.Π., Βρεττό Μουλό και τον Αχιλλέα Μαρινάκη, για το χρόνο που μου αφιέρωσαν, τη βοήθεια, τις οδηγίες και κατευθύνσεις που μου έδιναν σε όλα τα στάδια εκπόνησης της εργασίας.

Ένα μεγάλο ευχαριστώ στους συμφοιτητές και φίλους μου, την Ανθούσα, τον Παναγιώτη και τον Τάσο, για τα όμορφα φοιτητικά χρόνια που μου χάρισαν.

Το μεγαλύτερο ευχαριστώ, όμως, οφείλω προπαντός στην οικογένεια μου, για τη στήριξη και τη βοήθειά τους κατά τη διάρκεια των σπουδών μου.

Ζαχαρίας Β. Κωστόπουλος

Αθήνα, Οκτώβριος 2018

Πίνακας περιεχομένων

Πίνακας εικόνων

1

Εισαγωγή

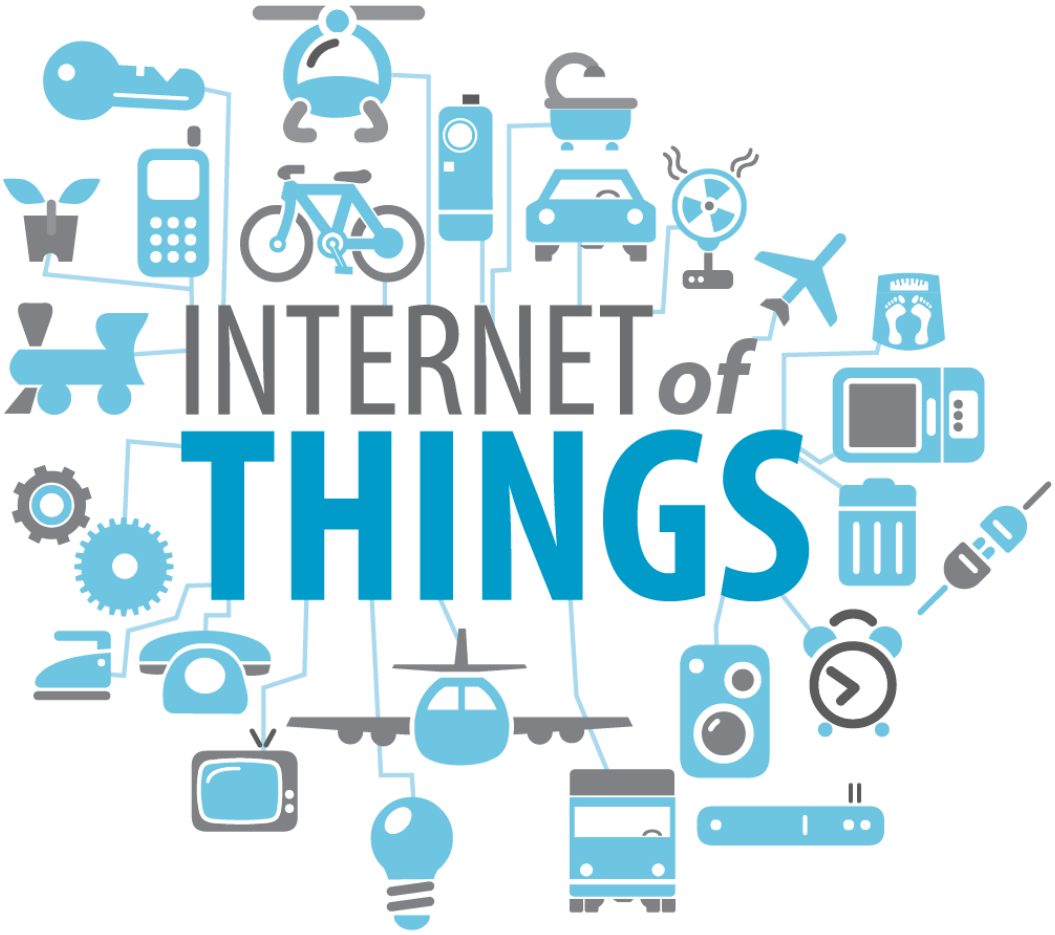
1.1 To Internet of Things

Το Internet of Things (IoT) ορίζεται ως ένα σύστημα διασυνδεδεμένων συσκευών, οι οποίες μπορεί να είναι μηχανικά και ψηφιακά συστήματα, αντικείμενα, ζώα και άνθρωποι, στα οποία αντιστοιχίζεται ένα μοναδικό αναγνωριστικό, και διαθέτουν τη δυνατότητα ανταλλαγής δεδομένων μεταξύ τους, χωρίς την απαίτηση της επικοινωνίας μεταξύ χρηστών ή μεταξύ χρηστών και υπολογιστών.

Ένα αντικείμενο στο IoT μπορεί να είναι μία έξυπνη ηλεκτρική συσκευή σπιτιού, που ειδοποιεί το κινητό τηλέφωνο του χρήστη για τυχόν βλάβη. Ένας άνθρωπος με ηλεκτρικό τσιπ, εμφυτευμένο στο σώμα του για παρακολούθηση της καρδιακής λειτουργίας. Ένα αυτοκίνητο με αισθητήρες που ειδοποιούν τον οδηγό σε περίπτωση υψηλής θερμοκρασίας. Γενικότερα, αντικείμενο του IoT αποτελεί κάθε φυσικό ή τεχνητό μέσο στο οποίο είναι δυνατό να ανατεθεί μία διεύθυνση IP, και να μεταδώσει αυτόματα δεδομένα προς άλλα αντικείμενα.

Ιστορικά, ο όρος του IoT εισήχθη το 1999, από τον Kevin Ashton (συνιδρυτής και executive director του Auto-ID center στο MIT). Ο Ashton αναφέρθηκε στο γεγονός ότι, κατά την περίοδο εκείνη, το σύνολο των δεδομένων που διαχειρίζονταν οι υπολογιστές και το διαδίκτυο βασιζόταν κατά κύριο λόγο στον ανθρώπινο παράγοντα, αφού κάθε πληροφορία έπρεπε να εισαχθεί από τον ίδιο χρήστη. Εάν, όμως, τα αντικείμενα είχαν τη δυνατότητα της συλλογής των δεδομένων αυτόματα, θα μπορούσαμε να παρακολουθούμε και να μετράμε τα δεδομένα

ευκολότερα, μειώνοντας το κόστος και τον χρόνο. Ωστόσο, παρόλο που ο όρος του IoT εμφανίστηκε το 1999, η χρήση του είχε αρχίσει να αναπτύσσεται αρκετά χρόνια πριν.



Εικόνα 1.1 Το IoT

Οι εφαρμογές, και τα πλεονεκτήματα του IoT είναι πολυάριθμα, καθώς, η χρήση του οδηγεί σε σημαντική μείωση κόστους και χρόνου, είτε αυτό πρόκειται για επαγγελματική χρήση μικρής ή μεγάλης κλίμακας, είτε για καθημερινή οικιακή χρήση. Αναφέρονται ενδεικτικά τα εξής παραδείγματα. Έξυπνα οχήματα, που ειδοποιούν άμεσα σε περίπτωση βλάβης, και με αυτό τον τρόπο έχουμε εξοικονόμηση καυσίμων και πιθανών εξόδων επισκευής, καθώς και παρέχεται περισσότερη ασφάλεια. Έξυπνες ηλεκτρικές οικιακές συσκευές, όπως θα μπορούσε να είναι ένα ψυγείο που θα ελέγχει το εσωτερικό του και θα ειδοποιεί το χρήστη για τις ελλείψεις που υπάρχουν, ή ένα πλυντήριο που επιτρέπει τον χειρισμό του μέσω κινητού τηλεφώνου, εξοικονομώντας χρόνο στο χρήστη.

Αναμφίβολα, το IoT αποτελεί ένα σημαντικό κομμάτι της ζωής μας σήμερα, και αυτό φαίνεται από τις ήδη υπάρχουσες εφαρμογές του. Στον τομέα της υγείας, τη βιομηχανία, το εμπόριο, τις μεταφορές, τις επικοινωνίες, την ενέργεια αλλά και σε πολλούς άλλους τομείς, η χρήση του IoT είναι κάτι παραπάνω από εκτεταμένη. Αυτή η αύξηση στη χρήση του, είναι και η αιτία για

πλήθος μελετών και ερευνών, για την διεύρυνση και ανάπτυξη του, αλλά και για την επίλυση των ζητημάτων τα οποία αναπόφευκτα εμφανίζονται.

Στην παρούσα διπλωματική εργασία, θα προσπαθήσουμε να διερευνήσουμε κατά πόσο μπορούμε να συμβάλουμε στην διευκόλυνση των προγραμματιστών του IoT κάνοντας χρήση νέων υποσχόμενων τεχνολογιών.

1.2 Function as a Service

Η ευρύτατη χρήση του IoT δημιουργεί συνεχώς νέες ανάγκες, οι οποίες έγκειται κυρίως στον όγκο των δεδομένων που διαχειρίζονται καθημερινά. Νέες αρχιτεκτονικές σχεδιάζονται συνεχώς, που στοχεύουν στη μείωση κόστους και χρόνου, ώστε να αντιμετωπίζονται εύκολα τα πιθανά προβλήματα. Μία νέα, και πολλά υποσχόμενη αρχιτεκτονική, είναι αυτή του Function as a Service (**FaaS**). Ωστόσο, για να κατανοήσουμε πλήρως την FaaS αρχιτεκτονική θα πρέπει πρώτα να αναλύσουμε την έννοια του *Serverless Computing*.

Το serverless computing αποτελεί ένα μοντέλο προγραμματισμού στο οποίο ο προγραμματιστής βλέπει μόνο τον κώδικα του και ό,τι αφορά την εφαρμογή που φτιάχνει, διότι ένα μεγάλο μέρος των λειτουργικών ζητημάτων παραμένουν κρυφά από αυτόν, ενώ οι υπολογιστικοί πόροι που απαιτούνται για την εφαρμογή χρησιμοποιούνται μόνο κατά την εκτέλεση της.

Πρακτικά, η διαχείριση των πόρων του προγράμματος γίνεται πλήρως στο cloud, γεγονός που παρέχει τη δυνατότητα του serverless, το ότι δηλαδή μία εφαρμογή μπορεί να εκτελεστεί χωρίς να περιορίζεται σε συγκεκριμένο server, αποφεύγοντας έτσι και προβλήματα που μπορούν να προκύψουν από την υπερφόρτωση του server.

Στη βάση του serverless computing, έρχεται να συμπληρώσει η έννοια της FaaS αρχιτεκτονικής, η οποία μας παρέχει τη δυνατότητα να εκτελούμε κώδικα με έναυσμα κάποιο γεγονός, χωρίς να απαιτείται η δημιουργία κάποιας περίπλοκης αρχιτεκτονικής. Αυτό πρακτικά σημαίνει ότι ο προγραμματιστής μπορεί πλέον να ανεβάζει στο cloud μικρά κομμάτια κώδικα, τα οποία εκτελούνται ανεξάρτητα. Το γεγονός αυτό, παρέχει αρκετά πλεονεκτήματα, από τη στιγμή που πλέον ο server δεν είναι αυτός που έχει όλο το βάρος της εκτέλεσης του κώδικα, αλλά η κάθε μία ενέργεια εκτελείται αυτόματα και ανεξάρτητα.

1.2.1 Πλεονεκτήματα της FaaS αρχιτεκτονικής

Η FaaS αρχιτεκτονική, παρέχει αρκετά πλεονεκτήματα. Αρχικά, η χρήση FaaS μειώνει σημαντικά τις αρμοδιότητες του προγραμματιστή, από τη στιγμή που ένα μεγάλο μέρος της διαχείρισης των υποδομών του server διαχειρίζεται από κάποιον άλλο.

Επιπλέον, η FaaS αρχιτεκτονική δίνει περισσότερο χρόνο στον προγραμματιστή, να ασχοληθεί με συγκεκριμένα μέρη του κώδικα, τα οποία επιτελούν μία συγκεκριμένη λειτουργία, με αποτέλεσμα να έχει μεγαλύτερη αποτελεσματικότητα, αλλά και μεγαλύτερη ταχύτητα για την υλοποίηση του κώδικα.

Οι εφαρμογές που δημιουργούνται με χρήση FaaS αρχιτεκτονική, έχουν, επίσης, μεγαλύτερες δυνατότητες επεκτασιμότητας. Αυτό συμβαίνει, διότι είναι αρκετά πιο εύκολο να πετύχουμε επεκτασιμότητα (scalability), επεκτείνοντας αυτόματα κάθε κομμάτι κώδικα ξεχωριστά, σε σχέση με την προσπάθεια επέκτασης του κώδικα ως σύνολο.

Ένα ακόμη, μείζονος σημασίας, πλεονέκτημα που παρέχει η FaaS αρχιτεκτονική, έγκειται στο γεγονός ότι οι πόροι του συστήματος χρησιμοποιούνται μόνο κατά την εκτέλεση κάποιας ενέργειας (μικρό κομμάτι κώδικα). Επομένως, έχουμε μηδενικό προγραμματιστικό κόστος, για τους αδρανείς πόρους, αφού εκτελούνται μόνο όταν είναι απολύτως απαραίτητοι.

Η FaaS αρχιτεκτονική παρέχει ενσωματωμένη διαθεσιμότητα (availability) και ανοχή σφαλμάτων.

1.2.2 Μειονεκτήματα της FaaS αρχιτεκτονικής

Η χρήση της FaaS αρχιτεκτονικής έχει, ωστόσο, κάποια σημαντικά μειονεκτήματα. Αρχικά, υπάρχει μειωμένη διαφάνεια, από τη στιγμή που οι υποδομές του προγραμματιστή διαχειρίζονται από κάποιον άλλο, και η κατανόηση του συνολικού συστήματος μπορεί να είναι αρκετά δύσκολη.

Επίσης, δυσκολίες εμφανίζονται και κατά το debugging των FaaS εφαρμογών. Παρόλο που υπάρχουν συγκεκριμένα εργαλεία για απομακρυσμένο debugging, αυτά δεν είναι απόλυτα αποτελεσματικά, και συνεπώς, απαιτείται βελτίωση στο συγκεκριμένο κομμάτι.

Το γεγονός της αυτόματης επεκτασιμότητας των μεμονωμένων λειτουργιών μιας εφαρμογής σε FaaS αρχιτεκτονική, μπορεί, ακόμη, να δημιουργήσει προβλήματα, διότι υπάρχει περίπτωση για αυτόματη επέκταση του κόστους. Αυτό μπορεί να αποτελέσει αιτία για αδυναμία υπολογισμού του κόστους προκαταβολικά.

Ακόμη ένα πρόβλημα που μπορεί να προκύψει, οφείλεται στο γεγονός ότι στην FaaS αρχιτεκτονική λειτουργούμε αποκλειστικά με μικρές συναρτήσεις οι οποίες εκτελούν συγκεκριμένες ενέργειες. Ωστόσο, σε μία μεγάλης κλίμακας εφαρμογή, θα έχουμε μεγάλο αριθμό συναρτήσεων, γεγονός που μπορεί να δυσκολέψει τον προγραμματιστή στη διαχείριση του συνόλου του κώδικα. Για το λόγο αυτό, απαιτείται η καλύτερη διαχείριση των λειτουργικών μονάδων της εφαρμογής, και η ανάλογη κατηγοριοποίησή τους.

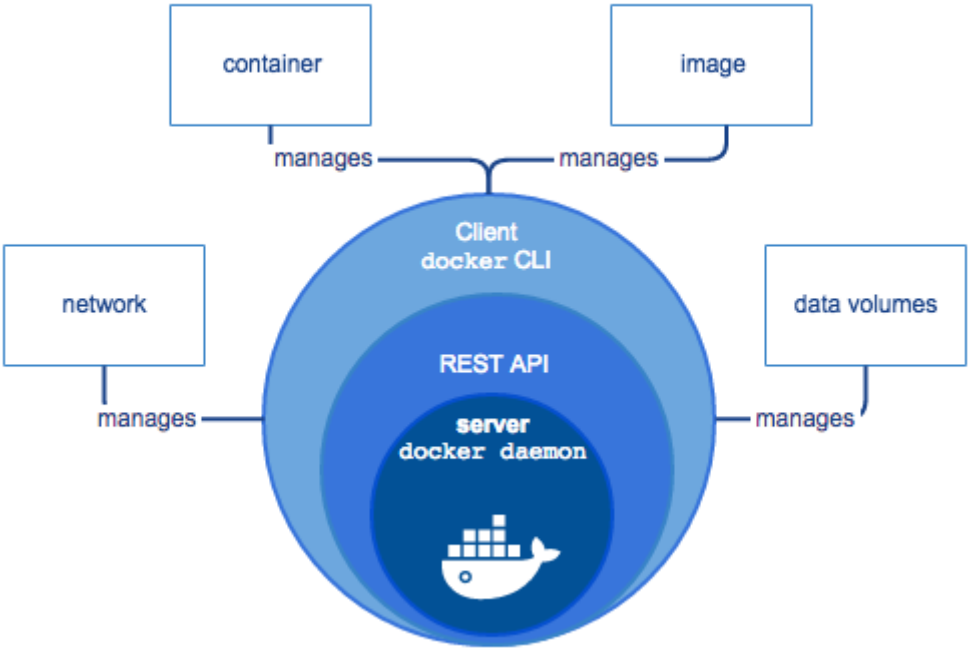
1.2.3 Εφαρμογές της FaaS αρχιτεκτονικής

Η FaaS αρχιτεκτονική, είναι ιδιαίτερα χρήσιμη, σε εφαρμογές web, backend, εφαρμογές διαχείρισης δεδομένων σε πραγματικό χρόνο, chatbots, εφαρμογές με προγραμματισμένες ενέργειες, καθώς και για IT αυτοματισμούς. Όπως είδαμε, η FaaS αρχιτεκτονική έχει σημαντικά πλεονεκτήματα κι αυτός είναι και ο λόγος που το πεδίο εφαρμογής της έχει μεγάλο εύρος. Αξίζει, επίσης, να αναφερθεί, ότι η FaaS είναι ιδιαίτερα χρήσιμη για το IoT. Η FaaS αρχιτεκτονική, έχει ως βασικό χαρακτηριστικό της την εκτέλεση κώδικα, με έναυσμα κάποιο γεγονός. Αυτό, έρχεται πολύ κοντά στη φύση του IoT, αφού σε πολλές εφαρμογές του, έχουμε εκτέλεση κώδικα μετά από συλλογή δεδομένων από κάποια συσκευή, για παράδειγμα κάποιον αισθητήρα, και συνεπώς η εφαρμογή θα μπορούσε εύκολα να επωφεληθεί από τα πλεονεκτήματα της FaaS αρχιτεκτονικής.

Σήμερα υπάρχουν αρκετές υπηρεσίες που παρέχουν τη δυνατότητα για προγραμματισμό σε FaaS αρχιτεκτονική. Τέτοιες υπηρεσίες είναι το Microsoft Azure Functions, οι AWS Lambda Functions και το IBM Openwhisk.

1.3 Containers

Όπως είδαμε, το IoT εξελίσσεται ραγδαία, και η χρήση του αυξάνει με γοργούς ρυθμούς. Αυτό έχει ως αποτέλεσμα την απαίτηση νέων τεχνολογιών για την καλύτερη διαχείριση του. Πέρα από την τεχνολογία της FaaS αρχιτεκτονικής, σημαντικό ρόλο παίζουν πλέον οι Containers.



Εικόνα 1.2 Containers

Ένα container image, είναι ένα ελαφρύ, αυτόνομο, εκτελέσιμο πακέτο ενός μέρους από κάποιο πρόγραμμα, το οποίο εμπεριέχει όλα τα απαραίτητα στοιχεία για την εκτέλεση του: τον κώδικα, το χρόνο εκτέλεσης (runtime), εργαλεία συστήματος, βιβλιοθήκες συστήματος και ρυθμίσεις. Το βασικό χαρακτηριστικό μία εφαρμογής που εκτελείται μέσω ενός container είναι ότι θα εκτελείται πάντοτε με τον ίδιο τρόπο, ανεξαρτήτως του περιβάλλοντος εκτέλεσης της εφαρμογής. Οι containers επομένως, χρησιμεύουν στην απομόνωση του προγράμματος από κάθε εξωτερικό στοιχείο, όπως θα μπορούσαν να είναι διαφορές μεταξύ των διάφορων περιβαλλόντων εκτέλεσης και σχεδιασμού, ενώ με αυτό τον τρόπο συμβάλλουν στην μείωση συγκρούσεων μεταξύ διαφορετικών ομάδων που εκτελούν διαφορετικά λογισμικά στις ίδιες υποδομές.

Οι containers που εκτελούνται σε ένα υπολογιστή, μοιράζονται τον πυρήνα του λειτουργικού συστήματος με τον υπολογιστή. Εκκινούν άμεσα, και χρησιμοποιούν λιγότερη RAM, και λιγότερα υπολογιστικά μέσα από τον υπολογιστή. Οι εικόνες δημιουργούνται από layers του filesystem και μοιράζονται μεταξύ τους κοινά αρχεία. Με αυτό τον τρόπο, ελαχιστοποιείται η χρήση του δίσκου, και αυξάνεται η ταχύτητα κατεβάσματος των εικονικών αρχείων.

Οι containers βασίζονται σε γενικά πρότυπα, και μπορούν να εκτελεστούν στις περισσότερες εκδόσεις των Linux, των Microsoft Windows, αλλά και σε πολλές άλλες υποδομές που περιέχουν εικονικά μηχανήματα (*VMs*), ακόμα και στο cloud.

Επιπλέον, με τη χρήση των containers, επιτυγχάνουμε την ασφάλεια του συστήματος μας, από τη στιγμή που, όπως αναφέρθηκε, έχουμε απομόνωση της εφαρμογής από το εξωτερικό της περιβάλλον, και με αυτό τον τρόπο, τα τυχόν προβλήματα του λογισμικού που εκτελείται εντός του container, περιορίζονται αποκλειστικά στο εσωτερικό του.

1.3.1 Containers και VMs

Ένα σημαντικό σημείο, όσον αφορά τους containers, είναι το γεγονός ότι κατά πολλούς προσομοιάζουν σε μεγάλο βαθμό τις εικονικές μηχανές. Αυτό έγκειται στο γεγονός ότι και οι δύο τεχνολογίες παρέχουν απομόνωση των πόρων του συστήματος, καθώς και αρκετά πλεονεκτήματα διαμοιρασμού, ωστόσο, έχουν εντελώς διαφορετική λειτουργία, διότι οι containers προσομοιώνουν κυρίως το λειτουργικό σύστημα, ενώ οι VMs το hardware.

Οι containers αποτελούν μία προσομοίωση στο επίπεδο της εφαρμογής, και ενοποιούν την ίδια την εφαρμογή μαζί με τις εξαρτήσεις της. Πολλοί containers μπορούν να λειτουργούν στο ίδιο σύστημα, και να μοιράζονται τους πυρήνες του λειτουργικού συστήματος με άλλους containers, ενώ ο καθένας εκτελείται ως ξεχωριστή διεργασία, στο σύστημα του χρήστη. Οι containers χρησιμοποιούν λιγότερο χώρο στο δίσκο σε σχέση με τα VMs (της τάξης των δεκάδων MB), και εκκινούν στιγμιαία.

Τα εικονικά μηχανήματα, από την άλλη μεριά, αποτελούν μία προσομοίωση του φυσικού μηχανήματος, και ουσιαστικά μετατρέπουν τον ένα server σε πολλούς servers. Ο hypervisor (το λογισμικό μέσω του οποίου ο χρήστης δημιουργεί και διαχειρίζεται τα VMs του) παρέχει τη δυνατότητα εκτέλεσης πολλών VM στον ίδιο υπολογιστή. Κάθε VM περιλαμβάνει ένα πλήρες αντίγραφο ενός λειτουργικού συστήματος, μία ή και περισσότερες εφαρμογές, και απαραίτητες βιβλιοθήκες και binaries, τα οποία καταλαμβάνουν μεγάλο χώρο στο δίσκο (της τάξης των δεκάδων GB). Επίσης, τα VMs, πιθανόν να απαιτούν αρκετό χρόνο για να εκκινήσουν.

Παρά τις διαφορές τους, οι containers και τα VMs μπορούν να χρησιμοποιηθούν ταυτόχρονα, μέθοδος η οποία συμβάλλει σε καλύτερα αποτελέσματα. Αυτό πραγματοποιείται με την χρήση container στο εσωτερικό ενός VM (μάλιστα ένα VM θεωρείται ως το καλύτερο σύστημα για την εκτέλεση container). Βασικό προσόν της μεθόδου, είναι ότι οι υπηρεσίες του container έχουν τη δυνατότητα να αλληλοεπιδρούν με τις λειτουργίες του VM. Επομένως, εάν για παράδειγμα η εφαρμογή που εκτελούμε μέσω container απαιτεί την προσπέλαση μίας βάσης δεδομένων, η οποία είναι αποθηκευμένη στο VM, τότε αυτό είναι απόλυτα εφικτό.

Επιπρόσθετα, η κοινή χρήση VM και container, συμβάλει στην βελτιστοποίηση της χωρητικότητας. Τα VMs έγιναν γρήγορα διάσημα, επειδή επέτρεψαν υψηλότερα επίπεδα χρήσης των server. Έτσι σε κάποιο εικονικό διακομιστή, μπορούμε να έχουμε VMs που

περιέχουν containers, αλλά και άλλα απλά VMs. Με αυτόν τον τρόπο, οι διαχειριστές του συστήματος εξασφαλίζουν την μέγιστη αξιοποίηση στο φυσικό υλικό του συστήματος.

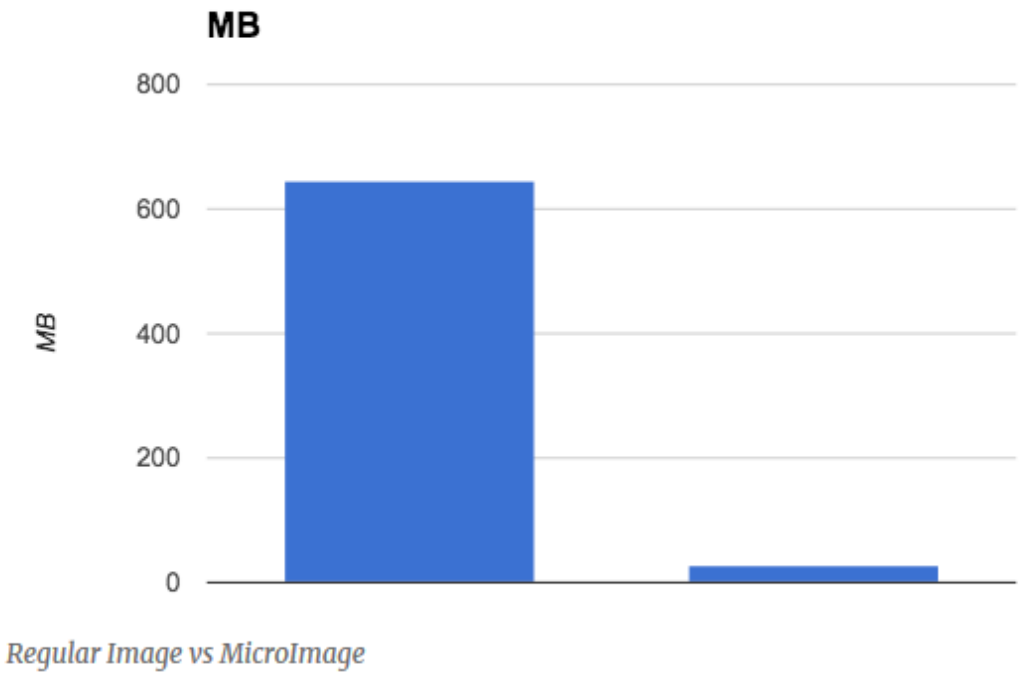
Ωστόσο, υπάρχουν πολλές απόψεις κατά τις οποίες οι containers είναι προτιμότερο να εκτελούνται εκτός των VMs, δηλαδή σε φυσικά μηχανήματα. Παρόλα αυτά, αυτό το οποίο μπορούμε να πούμε με βεβαιότητα, είναι ότι κάθε εφαρμογή χρήζει διαφορετικής αντιμετώπισης, και επομένως είναι στο χέρι του προγραμματιστή να επιλέξει το σύστημα το οποίο δίνει σε αυτόν τα περισσότερο ικανοποιητικά αποτελέσματα.

1.3.2 Microcontainers

Όπως είδαμε, οι containers μας δίνουν τη δυνατότητα να «πακετάρουμε» την εφαρμογή μας μαζί με τις εξαρτήσεις της, σε ένα αυτόνομο εικονικό αρχείο. Με αυτό τον τρόπο μπορούμε να εκτελέσουμε την εφαρμογή μας εντός του container. Ωστόσο, το πρόβλημα που συνήθως δημιουργείται είναι ότι στα εικονικά αρχεία συμπεριλαμβάνονται πολλά άχρηστα αρχεία, και με αυτό τον τρόπο καταλήγουμε σε αρχεία μεγάλου μεγέθους, και επομένως σε μεγάλους containers. Αν πάρουμε για παράδειγμα το docker (ο πιο διάσημος τύπος container), εάν κάνουμε χρήση της επίσημης συλλογής για τη γλώσσα της επιλογής μας, θα καταλήξουμε σε τεράστια εικονικά αρχεία. Ενδεικτικά μία απλή Node.js, Hello world εφαρμογή, με χρήση της επίσημης συλλογής του docker, οδηγεί σε εικονικό αρχείο μεγέθους 643 MB. Αν αναλογιστούμε πως απλά η εφαρμογή έχει μέγεθος το πολύ 1 MB μαζί με τις εξαρτήσεις της, καθώς και το runtime του Node.js καταναλώνει περί τα 20 MB, το μέγεθος του εικονικού αρχείου είναι υπερβολικό.

Λύση στο παραπάνω ζήτημα έρχονται να δώσουν οι microcontainers, οι οποίοι περιέχουν μόνο τις βιβλιοθήκες του λειτουργικού συστήματος και τις γλωσσικές εξαρτήσεις οι οποίες είναι απαραίτητες για την εκτέλεση της εφαρμογής, καθώς και την ίδια την εφαρμογή, και τίποτα περισσότερο.

Έτσι, στο παραπάνω παράδειγμα, η χρήση microcontainer θα έδινε εικονικό αρχείο μεγέθους 29 MB, δηλαδή 22 φορές μικρότερο από το αντίστοιχο με container.



Εικόνα 1.3 Εικονικό αρχείο σε Container και Microcontainer

Η χρήση των microcontainer μας παρέχει σημαντικά πλεονεκτήματα. Αρχικά, όπως αναφέραμε ήδη, γλιτώνουμε σημαντικό χώρο στο δίσκο, από τη στιγμή που χωρίς καμία αλλαγή στον κώδικα, επιτυγχάνουμε εικονικά αρχεία 20 φορές μικρότερα από τους τυπικούς containers.

Επιπλέον, με τη χρήση των microcontainer διευκολύνεται σημαντικά ο διαμοιρασμός. Από τη στιγμή που έχουμε πολύ μικρότερα εικονικά αρχεία, το κατέβασμα τους από κάποιο repository είναι αρκετά πιο γρήγορο, και συνεπώς μπορούμε να τη χρησιμοποιήσουμε σε διαφορετικά μηχανήματα πολύ πιο εύκολα.

Ακόμη, το μικρότερο μέγεθος του εικονικού αρχείου συνεπάγεται μεγαλύτερη ασφάλεια, αφού υπάρχει μικρότερη «επιφάνεια» προς επίθεση.

1.4 Microservices

Ένα σημαντικό σημείο που αφορά την ανάπτυξη εφαρμογών στο IoT είναι και η προγραμματιστική αρχιτεκτονική η οποία θα χρησιμοποιήσουμε. Μία νέα, πολλά υποσχόμενη αρχιτεκτονική, είναι αυτή των microservices. Αποτελεί, ουσιαστικά, μία προσέγγιση κατά την οποία η εφαρμογή μας συνίσταται από ένα σύνολο μικρών και ανεξάρτητων υπηρεσιών. Κάθε μία από αυτές εκτελείται σε δική της αυτόνομη διεργασία. Αυτές, μπορούν να επικοινωνούν μέσω κάποιου ελαφριού μηχανισμού (συνήθως HTTP). Επίσης, αυτές οι υπηρεσίες μπορούν

να υλοποιηθούν αυτόνομα, καθώς και η διαχείριση αυτών είναι μία ξεχωριστή υπηρεσία. Έχουν, ακόμη τη δυνατότητα, να υλοποιηθούν σε διαφορετικές γλώσσες και να χρησιμοποιήσουν διαφορετικά μοντέλα δεδομένων.

1.4.1 Μονολιθική Αρχιτεκτονική

Τα πλεονεκτήματα που μας προσφέρει η αρχιτεκτονική των microservices γίνονται περισσότερο κατανοητά, μετά από σύγκριση της με την κλασική μονολιθική αρχιτεκτονική. Αυτή η αρχιτεκτονική έχει ακριβώς αντίθετη βάση, κατά την οποία η εφαρμογή υλοποιείται σε ένα ενιαίο αρχείο.

Αρχικά, η μονολιθική αρχιτεκτονική προσφέρει πιο εύκολη υλοποίηση, εκτός κι αν το μέγεθος της εφαρμογής είναι πολύ μεγάλο. Επιπλέον, διευκολύνει σημαντικά την επεκτασιμότητα της εφαρμογής, από τη στιγμή που μπορούμε πολύ απλά να εκτελέσουμε πολλά αντίγραφα της ίδιας εφαρμογής. Παρόλα αυτά, η μονολιθική αρχιτεκτονική έχει σημαντικά μειονεκτήματα.

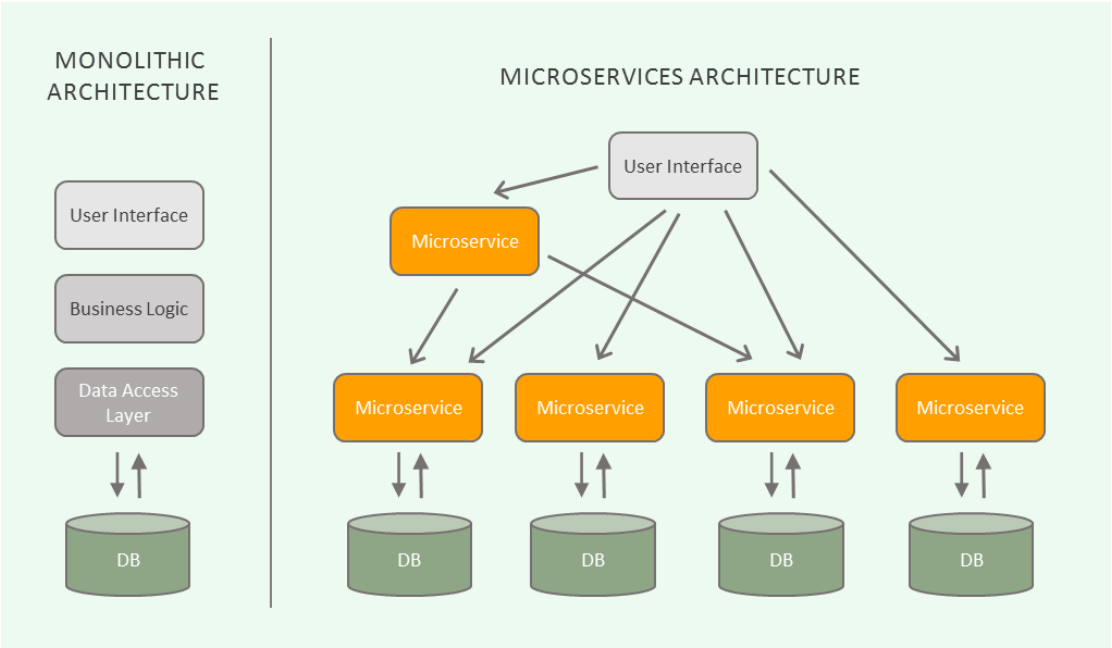
Μία εφαρμογή σε αυτή την αρχιτεκτονική, μπορεί να είναι αρκετά δύσκολη στην κατανόηση και επεξεργασία της. Αυτό γίνεται πιο έντονο όταν αναφερόμαστε σε εφαρμογές μεγάλης κλίμακας, ενώ, επιπλέον, είναι δύσκολη η διαχείριση και επεξεργασία της εφαρμογής από τρίτους.

Επιπρόσθετα, το μεγάλο μέγεθος κώδικα οδηγεί σε μείωση παραγωγικότητας, από τη στιγμή που οι προγραμματιστές δεν έχουν τη δυνατότητα να εργαστούν ξεχωριστά, και επομένως το σύνολο της ομάδας πρέπει να είναι απόλυτα συντονισμένο για την υλοποίηση κάθε αλλαγής στην εφαρμογή.

Ακόμη, η συνεχής εξέλιξη της εφαρμογής γίνεται πιο δύσκολη, διότι κάθε πιθανή αναβάθμιση σε κάποιο μικρό στοιχείο της εφαρμογής, απαιτεί την εκ νέου υλοποίηση του συνόλου της εφαρμογής.

Στο κομμάτι της επεκτασιμότητας, επίσης, η μονολιθική αρχιτεκτονική μπορεί να προκαλέσει δυσκολίες, επειδή, πολλές φορές η εφαρμογής μας διαχειρίζεται μεγάλο όγκο δεδομένων, και συνεπώς η συνεχής αντιγραφή της εφαρμογής θα απαιτεί μεγάλο υπολογιστικό χρόνο.

Λύση σε όλα τα παραπάνω, φαίνεται να δίνει πλέον η αρχιτεκτονική των microservices, η οποία θεωρείται ως την ιδανική αρχιτεκτονική για την ανάπτυξη εφαρμογών στο IoT, πράγμα απόλυτα λογικό, αν σκεφτεί κανείς ότι το IoT περιλαμβάνει πλήθος διαφορετικών συσκευών και υποδομών, και θα ήταν αδύνατη η επεξεργασία όλων από μία ενιαία εφαρμογή.



Εικόνα 1.4 Αντιδιαστολή Μονολιθικής Αρχιτεκτονικής και Microservices

1.4.2 Προκλήσεις των Microservices

Παρόλο που η αρχιτεκτονική των microservices μας παρέχει μεγάλες δυνατότητες, φέρνει και κάποια σημαντικά μειονεκτήματα. Αρχικά, η τεχνική αυτή καθιστά τη δημιουργία ενός κατανεμημένου συστήματος ιδιαίτερα περίπλοκη για τους προγραμματιστές, όπως επίσης και τον έλεγχο της λειτουργίας τους. Αυτό είναι και ένα από τα σημαντικότερα ζητήματα, και απαιτεί τη δημιουργία ενός μηχανισμού επικοινωνίας μεταξύ των υπηρεσιών.

Προφανές είναι, επίσης, ότι οι πολλές υπηρεσίες θα απαιτούν και καλή συνεργασία και συγκέντρωση μεταξύ των προγραμματιστών, ενώ η πολυπλοκότητα της υλοποίησης θα είναι αρκετά μεγάλη, αφού θα πρέπει να υλοποιήσουμε πλήθος υπηρεσιών που πιθανώς να είναι διαφορετικού τύπου.

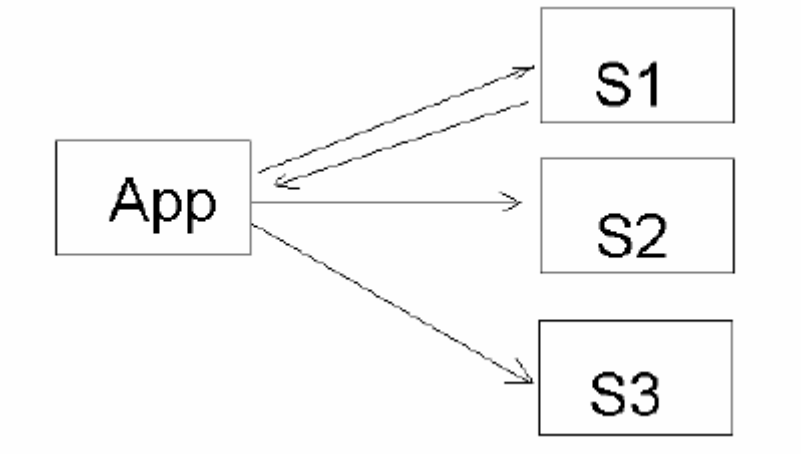
Επιπλέον, οι microservices είναι υπεύθυνες για μεγαλύτερη κατανάλωση μνήμης του υπολογιστή, αφού κάθε υπηρεσία απαιτεί διαφορετικό address space στη μνήμη.

Ωστόσο, το βασικότερο ζήτημα που δημιουργείται με τη χρήση των microservices είναι ο τρόπος διαμοιρασμού του συνολικού συστήματος σε διαφορετικές υπηρεσίες. Μία προφανής λύση σε αυτό, είναι ο διαχωρισμός σε υπηρεσίες αναλόγως της λειτουργίας που επιτελούν.

Κάθε υπηρεσία, ιδανικά, εμπεριέχει ένα μικρό σύνολο υποχρεώσεων. Σε αυτό αναφέρεται και το Single Responsible Principle (SRP, αρχή της μοναδικής υποχρέωσης), κατά το οποίο ορίζεται μία ευθύνη για κάθε κλάση ως αιτία αλλαγής της, και ότι η κλάση πρέπει να έχει μόνο μία αιτία για αλλαγή.

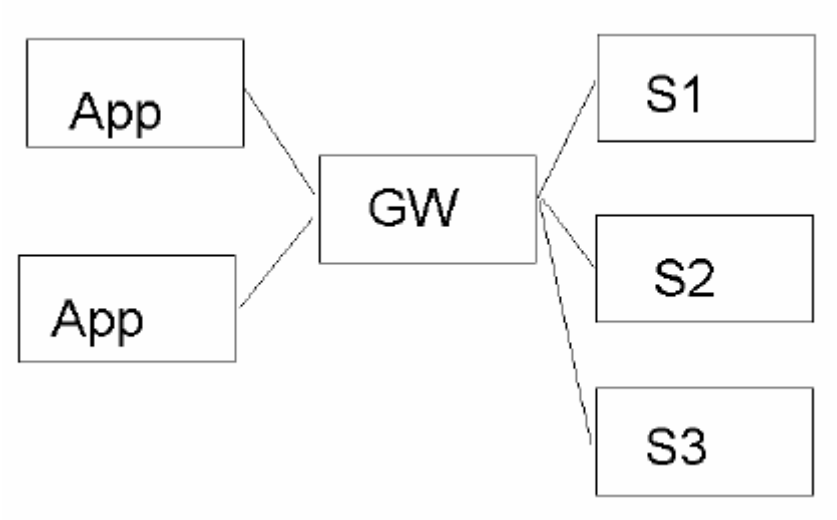
1.4.3 *Επικοινωνία στην αρχιτεκτονική των Microservices*

Ένα ακόμη σημαντικό ζήτημα με τις microservices αφορά τη διαχείριση της επικοινωνίας μεταξύ των διάφορων υπηρεσιών. Υπάρχουν διάφορα μοτίβα τα οποία μπορούμε να χρησιμοποιήσουμε. Το απλούστερο είναι αυτό κατά το οποίο η εφαρμογή έχει τη δυνατότητα να χρησιμοποιεί κάθε μία υπηρεσία άμεσα (εικόνα 1.2).



Εικόνα 1.5 Πρότυπο των άμεσων κλήσεων

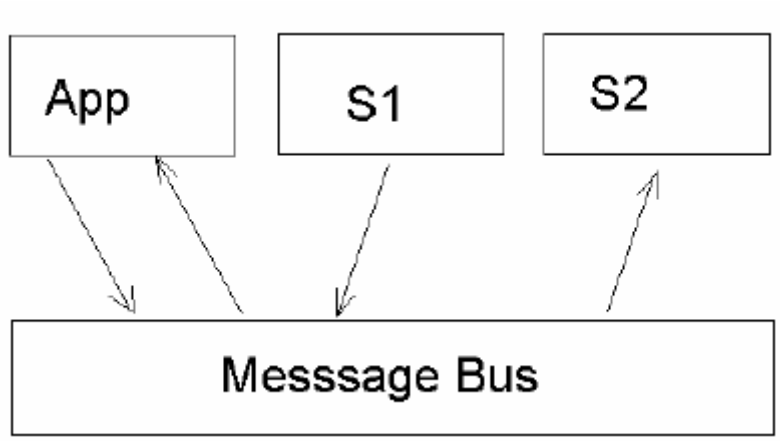
Το πρότυπο αυτό είναι αναμφίβολα ο πιο ευέλικτος τρόπος επικοινωνίας, ωστόσο, είναι πιθανό να οδηγήσει σε πιθανές καθυστερήσεις λόγω απομακρυσμένων κλήσεων. Επομένως, για να αντιμετωπίσουμε το πρόβλημα αυτό, θα πρέπει να μειώσουμε τον αριθμό των απομακρυσμένων κλήσεων. Συνεπώς, έχουμε ένα νέο πρότυπο το οποίο εμπεριέχει μία πύλη δικτύου (gateway), και συνίσταται κυρίως για IoT εφαρμογές (εικόνα 1.3).



Εικόνα 1.6 Πρότυπο του Gateway

Ένα ακόμη πρότυπο επικοινωνίας, είναι αυτό το οποίο περιλαμβάνει ένα service-bus, μία ροή δηλαδή, που αποδέχεται μηνύματα και τα μεταφέρει (εικόνα 1.4). Αυτό το πρότυπο ενδείκνυται επίσης για εφαρμογές IoT, εξ' αιτίας της ασύγχρονης φύσης των περισσότερων υπηρεσιών (για

παράδειγμα, για το μεγαλύτερο μέρος των αισθητήρων, η διαδικασία της εισαγωγής των δεδομένων είναι ασύγχρονη). Επομένως, με τη χρήση του service-bus, η εφαρμογή μπορεί να δημιουργεί κλήσεις, και να τις διοχετεύει, καθώς και να λαμβάνει απαντήσεις στη συνέχεια.



Εικόνα 1.7 Πρότυπο του Service-bus

1.5 AWS LAMBDA

Μέχρι αυτό το σημείο, έχουμε αναφερθεί στο IoT, στην FaaS αρχιτεκτονική και στον serverless προγραμματισμό, στους containers και τους microcontainers, και την αρχιτεκτονική των microservices. Σε αυτή την παράγραφο θα αναφερθούμε στην υπηρεσία της Amazon, AWS Lambda, που συνδυάζει όσα έχουμε αναφέρει μέχρι τώρα.

Η AWS Lambda είναι μία υπηρεσία που μας δίνει τη δυνατότητα προγραμματισμού χωρίς την απαίτηση για παροχή και διαχείριση server. Αποτελεί ουσιαστικά μία υπηρεσία για serverless προγραμματισμό. Η Lambda εκτελεί τον κώδικα όταν και μόνο όταν είναι απαραίτητο, και έτσι, έχουμε κατανάλωση υπολογιστικών πόρων, μόνο κατά την εκτέλεση κώδικα, και όχι όταν αυτός μένει αδρανής. Αυτό που μας παρέχει ουσιαστικά, είναι ότι μπορούμε να προγραμματίσουμε, έχοντας ως μόνη μας υποχρέωση τη σύνταξη του κώδικα, αφού η Lambda αναλαμβάνει όλες τις απαραίτητες διεργασίες αυτόματα. Υποστηρίζει τις γλώσσες Node.js, Java, Python, C# και Go.

Η Lambda, επιπλέον, αποτελεί και υπηρεσία FaaS αρχιτεκτονικής, διότι μας επιτρέπει την εκτέλεση κώδικα, ως απόκριση σε κάποιο συμβάν, τα οποία ωστόσο περιορίζονται στις παροχές της Amazon.

1.5.1 Παροχές της Lambda

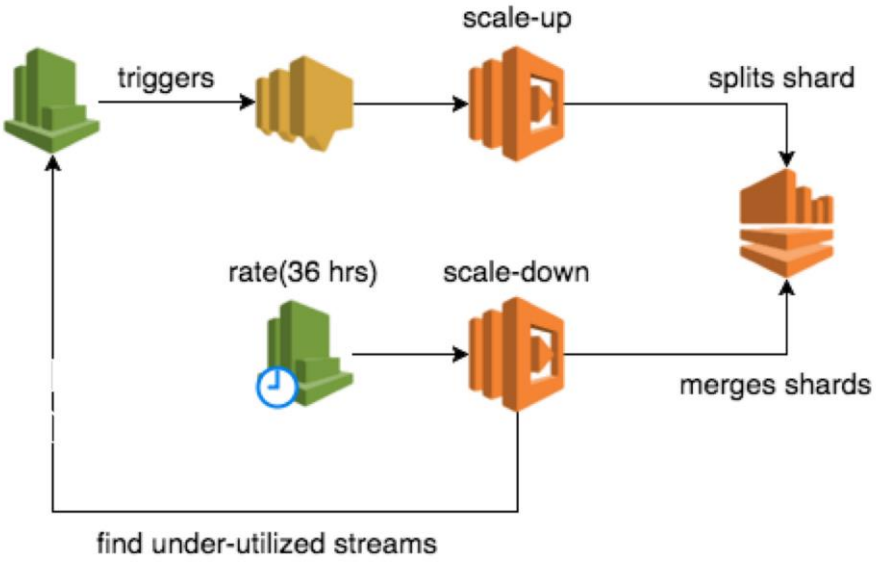
Αρχικά, επιτρέπει τη δημιουργία back-end υπηρεσιών (υπηρεσίες υποστήριξης). Κάνοντας χρήση της Lambda, μπορούμε να έχουμε back-end υπηρεσίες, οι οποίες θα εκτελούνται με

βάση κάποιο trigger από τις παροχές της Amazon, και με αυτό τον τρόπο μπορούμε να εξοικονομούμε πόρους, να αποφεύγουμε αλλαγές στην πλατφόρμα των clients, να εξοικονομήσουμε ενέργεια και μπαταρία και να διευκολύνουμε τις πιθανές αναβαθμίσεις.

Παρόλο που η Lambda είναι μια νέα υπηρεσία, δεν φέρνει καμία αλλαγή όσον αφορά τον γλωσσικό παράγοντα. Μπορούμε να προγραμματίσουμε όπως γνωρίζουμε έως τώρα, καθώς και να χρησιμοποιούμε τις βιβλιοθήκες που επιθυμούμε, στις γλώσσες που η υπηρεσία υποστηρίζει.

Η Lambda μας παρέχει όλες τις απαραίτητες υποδομές για την εκτέλεση του κώδικα μας σε ένα υψηλού επιπέδου και ανεκτικό σε σφάλματα περιβάλλον. Συνεπώς, ο προγραμματιστής δεν έχει πλέον την ευθύνη για αναβάθμιση του λειτουργικού του μηχανήματος που χρησιμοποιεί, ή να επεξεργάζεται τον server που εκτελείται η εφαρμογή. Η Lambda αναλαμβάνει πλήρως την εκτέλεση του κώδικα καθώς και τη διαχείριση των υποδομών, ενώ μας παρέχει τη δυνατότητα για παρακολούθηση της εκτέλεσης με logs και μέσω του Amazon CloudWatch.

Ένα ακόμη, σημαντικό πλεονέκτημα της Lambda είναι η αυτόματη επεκτασιμότητα (scaling). Αυτό πρακτικά, σημαίνει η Lambda θα επεκτείνει αυτόματα τον κώδικα μας, έτσι ώστε να μπορεί να ανταποκριθεί αποτελεσματικά στον αριθμό των κλήσεων που φτάνουν στον κώδικα, χωρίς να υπάρχει κανένα όριο στον αριθμό αυτό. Ουσιαστικά, όταν εκκινεί η εκτέλεση του κώδικα, η Lambda τον εκτελεί σε μερικά millisecond, για ένα event (που κάνει trigger τον κώδικα), και από τη στιγμή που έχουμε αυτόματο scaling, η επίδοση της εκτέλεσης παραμένει υψηλή, όσο αυξάνεται η συχνότητα των events.



Εικόνα 1.8 Scaling της Lambda

Η Lambda μας δίνει τη δυνατότητα για συντονισμό πολλών διεργασιών, για τη δημιουργία μεγάλων ή χρονοβόρων εργασιών, με χρήση της υπηρεσίας AWS Step Functions.

Επιπρόσθετα, οι υπηρεσίες της Lambda διαθέτουν αυξημένα επίπεδα ασφάλειας, για να εξασφαλίζεται η ακίνδυνη επικοινωνία μεταξύ του κώδικα και των υπόλοιπων υπηρεσιών της Amazon.

Τέλος, όπως έχουμε ήδη αναφέρει, το υπολογιστικό κόστος μίας εφαρμογή της Lambda είναι πολύ μικρό, από τη στιγμή που όταν ο κώδικας δεν εκτελείται, έχουμε μηδενική κατανάλωση σε πόρους του συστήματος, Επιπλέον, μας δίνει τη δυνατότητα να επιλέξουμε την ποσότητα της μνήμης για μία διεργασία, και κατανέμει αυτόματα τη χρήση CPU, του δικτύου και του δίσκου.

2

Orchestration Tools

Ένας από τους πιο σημαντικούς παράγοντες στον τομέα της ανάπτυξης εφαρμογών είναι η ταχύτητα. Σήμερα, ο όγκος των δεδομένων που καλείται κάθε εφαρμογή να διαχειριστεί είναι μεγάλος, και αυξάνεται με εκθετικούς ρυθμούς. Είναι προφανές, ότι υπάρχει μεγάλη ανάγκη για νέες λύσεις, τις οποίες οι προγραμματιστές θα χρησιμοποιήσουν, ώστε να μην υπάρξει μείωση στην ταχύτητα των εφαρμογών τους, η οποία είναι ζωτικής σημασίας.

Επιπλέον, καθώς οι εταιρίες αναπτύσσονται, και ασχολούνται με νέες τεχνολογίες και εφαρμογές, εύκολα μπορεί να δημιουργηθεί σύγχυση μεταξύ των εργαλείων που χρησιμοποιούνται, και να οδηγηθούμε στο φαινόμενο του “Toolchain Sprawl”. Αυτό, πρακτικά, δημιουργεί προβλήματα στη συνολική διαχείριση της ανάπτυξης μια εφαρμογής, καθώς και έλλειψη ελέγχου. Αν τα DevOps εργαλεία, δεν είναι ενορχηστρωμένα (orchestrated) με σωστό τρόπο, είναι πιθανό να καθυστερήσουν και να εμποδίσουν την ποιότητα των νέων εκδόσεων της αναπτυσσόμενης εφαρμογής. Τα Orchestration Tools ενσωματώνονται στα υπάρχοντα συστήματα, και μας επιτρέπουν την καλύτερη διαχείριση των DevOps εργαλείων, με αποτελεσματικό τρόπο, και με βάση μία κεντρική διασύνδεση.

2.1 Ορισμός

Τα orchestration tools ορίζονται ως ένα επιπρόσθετο, αυτοματοποιημένο layer, που εναποτίθεται στην κορυφή της προϋπάρχουσας αλυσίδας εργαλείων (toolchain). Μας δίνουν τη δυνατότητα, να ενσωματώνουμε και να οργανώνουμε τα DevOps εργαλεία, σε μία end-to-end διαδικασία. Αυτό δε σημαίνει ότι θα πρέπει να αφαιρέσουμε και να αντικαταστήσουμε τα υπάρχοντα εργαλεία, αλλά ουσιαστικά έχουμε πλέον τη δυνατότητα για προβολή και διαχείριση του συνόλου της διαδικασίας. Με αυτό τον τρόπο, εξοικονομούμε χρόνο για τους

προγραμματιστές, και καταφέρνουμε να έχουμε εκδόσεις με μεγαλύτερη ταχύτητα. Επομένως, επιτυγχάνουμε βελτίωση στην ταχύτητα, την ποιότητα και την επεκτασιμότητα.

2.2 Πλεονεκτήματα των Orchestration Tools

Τα Orchestration Tools είναι ιδιαίτερα χρήσιμα, γιατί συνδέουν απομακρυσμένα και απομονωμένα κομμάτια μιας εφαρμογής, και συνιστούν, με αυτό τον τρόπο, μία αρμονική και απόλυτα αυτοματοποιημένη διαδικασία. Μας παρέχουν τη δυνατότητα για καλύτερο έλεγχο και παρατήρηση της εφαρμογής, έτσι ώστε να μπορούμε να διαχειριστούμε μία κατά τ' άλλα περίπλοκη δομή. Επιπλέον, εκτελούν με δική τους πρωτοβουλίες συνεχείς διανομές της εφαρμογής, εξασφαλίζοντας με αυτό τον τρόπο, επιπλέον επίπεδα αξιοπιστίας και επεκτασιμότητας, και παρέχουν καλύτερη ταχύτητα και ποιότητα. Αυτό επιτυγχάνεται με την αφαίρεση των χειροκίνητων διαδικασιών και ελέγχων, και επομένως έχουμε μεγάλο κέρδος χρόνου, αφού ορισμένες χρονοβόρες λειτουργίες, εκτελούνται πλέον αυτόματα. Με αυτό τον τρόπο, σε μία ομάδα προγραμματιστών, ελαφρύνεται κατά πολύ το έργο τους, και μπορούν να εστιάσουν την προσοχή τους σε άλλα μείζονα ζητήματα. Τελικά, μπορούμε να συμπεράνουμε ότι τα orchestration tools συμβάλλουν στην καλύτερη οργάνωση και συνεργασία σε μία ομάδα, και διευκολύνουν τα DevOps εργαλεία, συμβάλλοντας έτσι, στον τελικό στόχο που είναι ο συνεχής διαμοιρασμός της εφαρμογής.

2.3 Παρουσίαση Orchestration Tools

Όπως είδαμε τα orchestration tools έχουν μεγάλη σημασία για την ανάπτυξη εφαρμογών στο IoT. Στην παράγραφο αυτή θα γίνει μία ανάλυση τω πιο διαδεδομένων orchestration tools που χρησιμοποιούνται σήμερα.

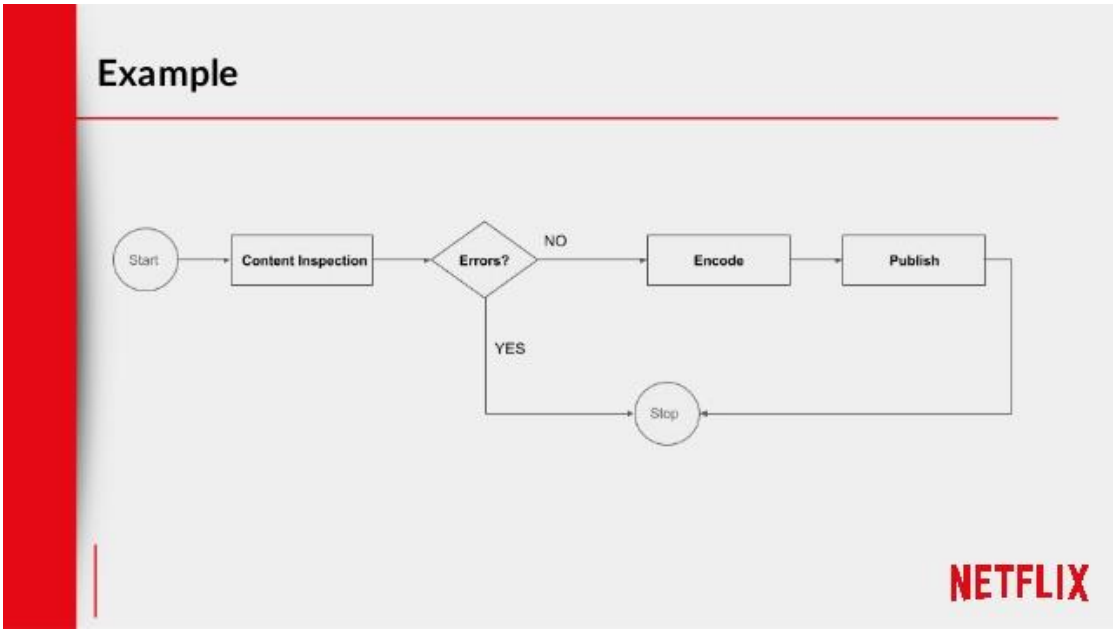
2.3.1 Netflix Conductor

Η Netflix αποτελεί, αναμφισβήτητα, μία από τις ταχύτερα αναπτυσσόμενες εταιρίες παγκοσμίως, ενώ ο όγκος των δεδομένων που διαχειρίζεται αυξάνεται συνεχώς. Οι ομάδες μηχανικών της Netflix, εκτελούν ένα μεγάλο αριθμό εργασιών που προέρχονται από ασύγχρονη ενορχήστρωση άλλων εργασιών που εκτελούνται σε microservices. Ορισμένες από αυτές απαιτούν αρκετό χρόνο εκτέλεσης, ακόμα και μερικές ημέρες, ενώ είναι εργασίες ζωτικής σημασίας για τη σωστή λειτουργία της πλατφόρμας του Netflix, και για την έγκαιρη παράδοση σειρών και ταινιών. Το γεγονός αυτό, συνδυαζόμενο με τη ραγδαία αύξηση του αριθμού των microservices και της πολυπλοκότητας των διαδικασιών, καθιστά την

παρακολούθηση των κατανεμημένων εργασιών ιδιαίτερα δύσκολη, χωρίς την ύπαρξη κάποιας κεντρικής ενορχήστρωσης. Η Netflix, συνεπώς, προχώρησε στη δημιουργία του Conductor.

Ο Conductor κατασκευάστηκε για να καλύψει τις παρακάτω απαιτήσεις, να περιορίσει την ανάγκη για boilerplate [1]κώδικα στις εφαρμογές και για να παρέχει μία διαδραστική ροή. Οι απαιτήσεις είναι:

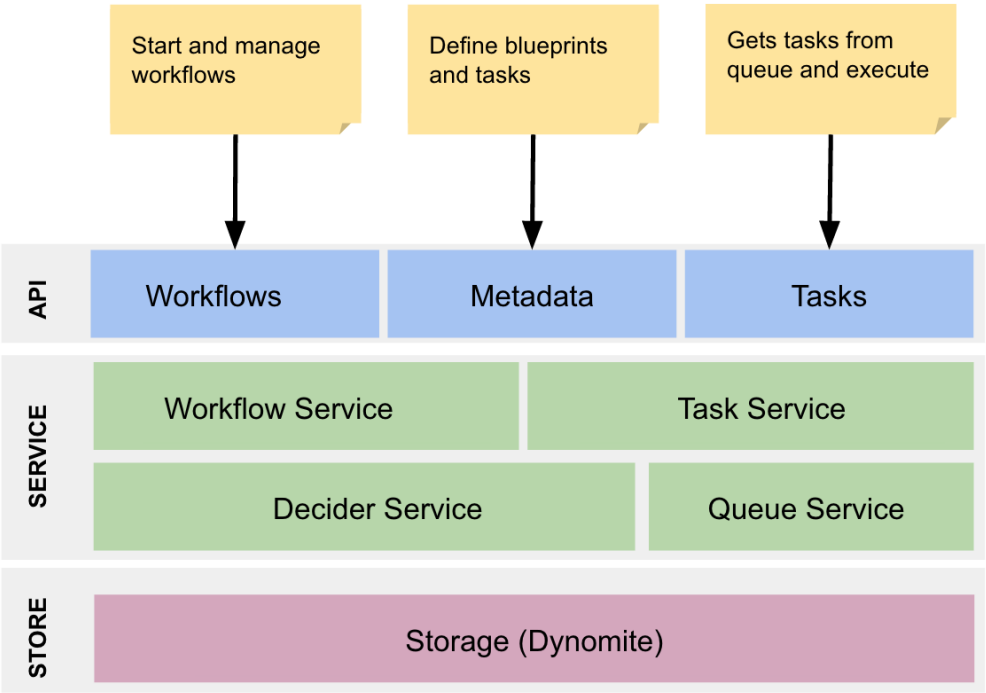
- Δημιουργία περίπλοκων διαδικασιών στις οποίες κάθε αυτόνομη εργασία υλοποιείται με μία microservice.
- Η ροή εκτέλεσης να καθορίζεται από ένα σχέδιο βασισμένο σε JSON DSL.
- Παροχή παρακολούθησης και traceability σε αυτές τις διαδικασίες.
- Διαχείριση των μεταβλητών ελέγχου (παύση, έναρξη, επανεκκίνηση κ.τ.λ.) για καλύτερη DevOps εμπειρία.
- Να επιτρέπει την ευρύτερη επαναχρησιμοποίηση ήδη υπαρχόντων microservices, παρέχοντας έτσι ένα εύκολο δρόμο για τη γρήγορη ενσωμάτωση νέων εργαζομένων.
- Διπροσωπία χρήστη (user interface) η οποία θα οπτικοποιεί τις ροές των διαδικασιών.
- Δυνατότητα για σύγχρονη διαχείριση όλων των εργασιών, όταν είναι αναγκαίο.
- Δυνατότητα για επεκτασιμότητα εκατομμυρίων παράλληλα εκτελέσιμων ροών διαδικασιών.
- Υποστήριξη από μία υπηρεσία αναμονής που αντλείται από τους clients.
- Δυνατότητα για επικοινωνία μέσω HTTP ή άλλου πάροχου.



Εικόνα 2.1 Παράδειγμα ροής εκτέλεσης

[1] Boilerplate: κώδικας ή κομμάτι κώδικα το οποίο επαναλαμβάνεται πολλές φορές αυτούσιο ή με ελάχιστες αλλαγές.

2.3.1.1 *Αρχιτεκτονική του Conductor*



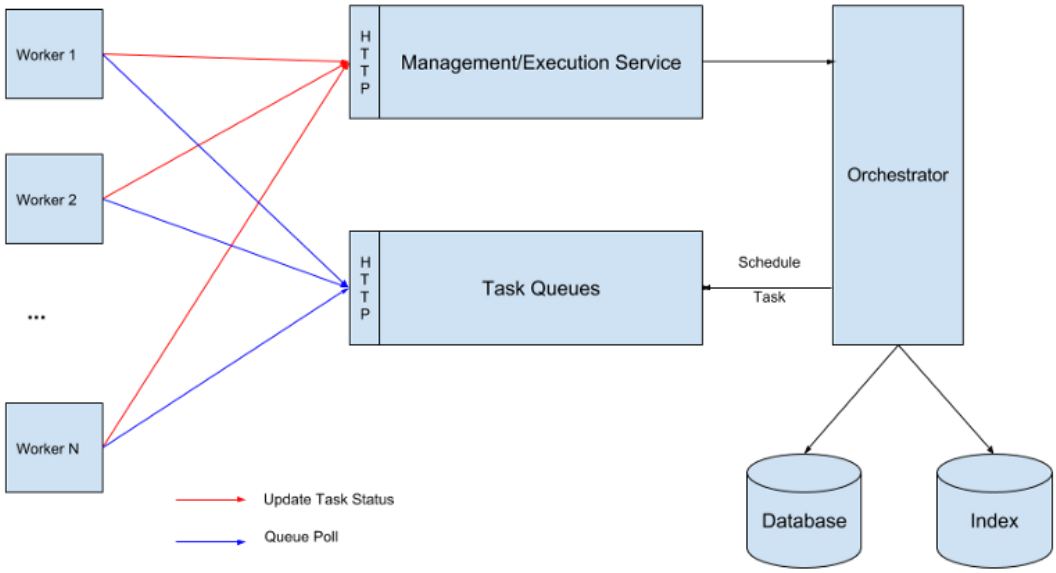
Εικόνα 2.2 Αρχιτεκτονική του Netflix Conductor

Στην καρδιά του συστήματος βρίσκεται μία υπηρεσία «μηχανής κατάστασης», η Decider Service. Όταν προκύψει κάποιο γεγονός (event) της ροής εργασιών (workflow), όπως είναι η ολοκλήρωση κάποια εργασίας ή ένα σφάλμα, ο Decider συνδυάζει το σχέδιο της ροής εργασιών με την τρέχουσα κατάστασή της, εντοπίζει την επόμενη κατάσταση, προγραμματίζει τις επόμενες εργασίες και ανανεώνει την κατάσταση της ροής εργασιών, εφόσον είναι απαραίτητο. Ο Decider κάνει χρήση κατανεμημένης ουράς για τη διαχείριση των προγραμματισμένων εργασιών.

Οι εργασίες, οι οποίες υλοποιούνται μέσω worker applications, επικοινωνούν μέσω του API layer. Οι workers, το καταφέρνουν αυτό, είτε υλοποιώντας ένα REST endpoint το οποίο καλείται από το orchestration engine, είτε ένα επαναλαμβανόμενο βρόγχο ο οποίος θα ελέγχει για εκκρεμείς εργασίες. Οι workers είναι σχεδιασμένοι ως ανεξάρτητες συναρτήσεις.

Τα API υλοποιούνται μέσω του HTTP πρωτοκόλλου, το οποίο διευκολύνει τη διασύνδεση με διαφορετικούς clients. Είναι ωστόσο, δυνατή η χρήση και άλλων πρωτοκόλλων, όπως το gRPC, και υλοποιείται με εύκολο τρόπο.

Όσον αφορά την αποθήκευση (storage), γίνεται χρήση του Dynomite, ως «μηχανή αποθήκευσης», σε συνδυασμό με το Elasticsearch που χρησιμοποιείται για τη δεικτοδότηση των ροών εκτέλεσης. Τα API αποθήκευσης (storage APIs) είναι αρκετά ευέλικτα, διότι μπορούν να προσαρμοστούν σε διαφορετικά συστήματα αποθήκευσης.



Εικόνα 2.3 Επικοινωνία των workers με το μηχάνημα

2.3.1.2 Δομικά στοιχεία του Netflix Conductor

Workflow

Τα workflows ορίζονται από ένα αρχείο JSON σε βάση DSL. Ένα σχέδιο workflow (workflow blueprint) καθορίζει μία σειρά ενεργειών που πρέπει να εκτελεστούν, οι οποίες μπορεί να είναι είτε ενέργειες ελέγχου (fork, join, decision, sub workflow κ.τ.λ.) ή μία εργασία worker.

Εργασία

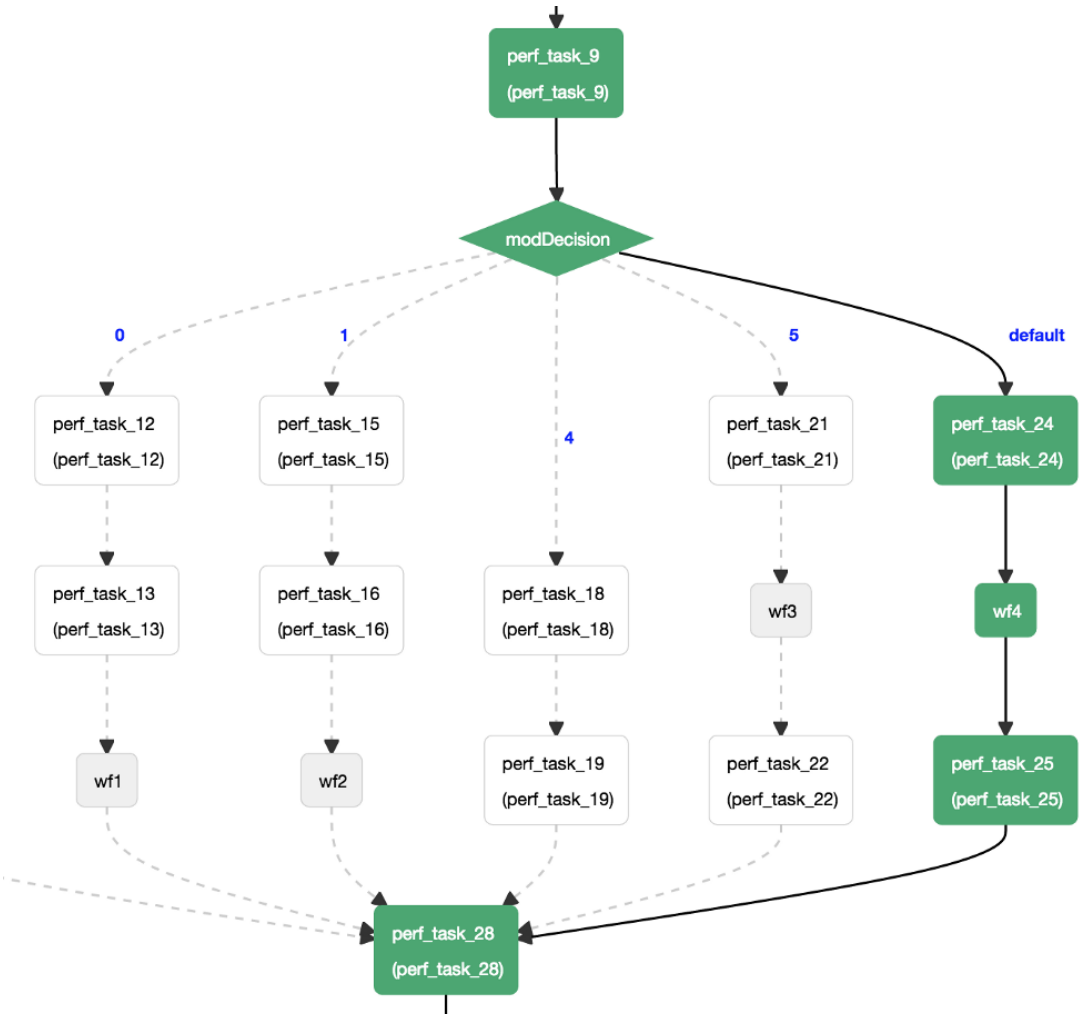
Η συμπεριφορά κάθε εργασίας ελέγχεται από το πρότυπο της το οποίο ονομάζεται task definition. Σε αυτό περιλαμβάνονται παράμετροι ελέγχου, που είναι απαραίτητες για κάθε εργασία. Η εργασία μπορεί να είναι worker που υλοποιείται από την εφαρμογή ή μία εργασία συστήματος που εκτελείται από τον orchestration server.

Inputs/Outputs

Αποτελούν τη μέθοδο που παρέχουν input σε μία εργασία ή μεταφέρουν το output άλλης εργασίας.

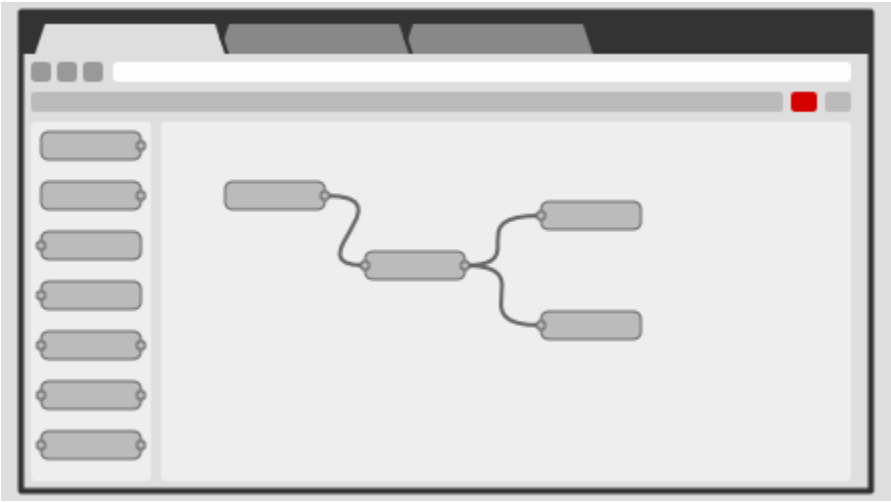
UI

Αποτελεί το βασικό μηχανισμό για παρακολούθηση και αποσφαλμάτωση την εκτέλεση των workflow. Μας παρέχει δυνατότητες αναζήτησης βασισμένες σε διάφορες παραμέτρους, καθώς και ένα γραφικό σχέδιο του workflow blueprint.



Εικόνα 2.4 Workflow blueprint

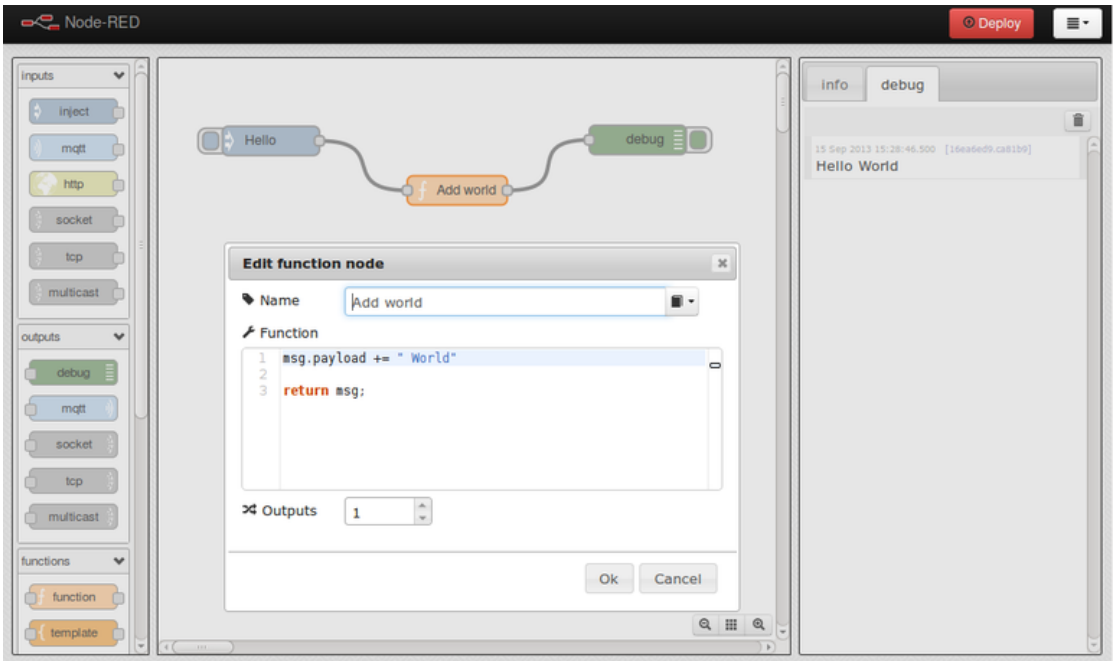
2.3.2 Node-red



Εικόνα 2.5 : Γραφικό περιβάλλον του Node-Red

Το Node-RED είναι ένα εργαλείο, το οποίο παρέχει τη δυνατότητα προγραμματισμού μέσω γραφικού περιβάλλοντος, με τη δημιουργία ροών (flows) υπηρεσιών. Το βασικό χαρακτηριστικό του είναι ότι αποτελείται από κόμβους (nodes), οι οποίοι επιτελούν κάποια λειτουργία ή υπηρεσία, και ελέγχονται από τον προγραμματιστή μέσω drag and drop.

Το Node-Red δημιουργήθηκε με στόχο τη διευκόλυνση των προγραμματιστών στο IoT, οι οποίοι, πολλές φορές, ήταν απαραίτητο να εκτελέσουν λειτουργίες περίπλοκες και χρονοβόρες στην υλοποίησή τους. Η ιδέα, λοιπόν, για το Node-Red, ήταν να φτιαχτεί μία πλατφόρμα, η οποία θα παρείχε έτοιμες όλες αυτές τις χρονοβόρες λειτουργίες.



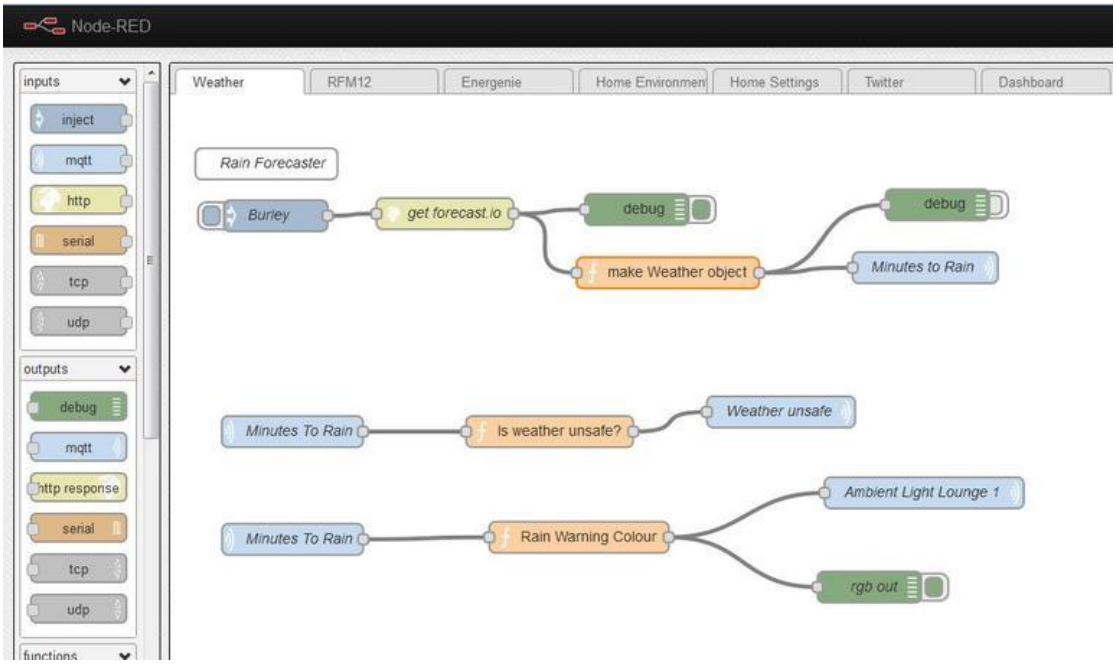
Εικόνα 2.6 Hello world στο Node-Red

Στην εικόνα 2.5 βλέπουμε ένα στιγμιότυπο από την πλατφόρμα του Node-Red, από ένα πρόγραμμα hello world! Το πρόγραμμα αυτό αποτελείται από τρεις απλούς κόμβους. Όλοι οι κόμβοι, εκτελούν κώδικα σε JavaScript για την εκτέλεση κάποιας ενέργειας. Ο πρώτος κόμβος, λέγεται Inject και εκκινεί τη ροή. Στο παράδειγμα μας, έχει επεξεργαστεί, ώστε να στέλνει το μήνυμα "Hello". Ο δεύτερος κόμβος, είναι κόμβος συνάρτησης JavaScript και εκτελεί τον κώδικα που φαίνεται γραμμένος. Προσθέτει δηλαδή στο μήνυμα "Hello" που έχει φτάσει από τον προηγούμενο κόμβο, το string " world". Τέλος ο κόμβος debug εμφανίζει το συνολικό μήνυμα στη δεξιά πλευρά της πλατφόρμας.

Βλέπουμε, ακόμη, ότι δεξιά και αριστερά (ή μόνο στη μία πλευρά) κάθε κόμβου, υπάρχει ένας γκρι κύκλος. Αυτός ορίζει το input και το output του κόμβου. Μέσω αυτό μπορούμε να συνδέσουμε τους κόμβους μεταξύ τους, κάνοντας κλικ σε ένα κόμβο και κρατώντας το πατημένο, αφήνουμε το ποντίκι όταν φτάσει στον κόμβο που επιθυμούμε να συνδέσουμε.

Επίσης, η πλατφόρμα μας παρέχει πληροφορίες για τους κόμβους που χρησιμοποιούμε, στη δεξιά πλευρά της σελίδας, στην καρτέλα info. Κάνοντας κλικ σε οποιοδήποτε κόμβο, εμφανίζονται αναλυτικές πληροφορίες για αυτόν.

Συνεπώς, είδαμε ένα απλό παράδειγμα εφαρμογής στο Node-Red το οποίο εμφάνιζε απλά ένα μήνυμα. Αυτό όμως που κάνει το Node-Red να ξεχωρίζει, είναι ότι μας παρέχει τη δυνατότητα για χρήση web services και hardware, με εύκολο τρόπο.



Εικόνα 2.7 Weather forecast του Dom Bramley

Ένα παράδειγμα χρήσης του node-RED είναι αυτό του Dom Bramley (εργαζόμενος της IBM) ο οποίος δημιούργησε μία IoT εφαρμογή για να ενημερώνεται για το κατά πόσο ο καιρός θα είναι ιδανικός για φωτογραφίσεις. Ο Bramley έφτιαξε μία ροή στο node-RED, στην οποία ένας κόμβος αντλούσε δεδομένα πρόγνωσης του καιρού ανά τρία λεπτά και ένας δεύτερος κόμβος

υπολόγιζε τα λεπτά που απέμεναν μέχρι να βρέξει στη συγκεκριμένη περιοχή. Στη συνέχεια, υπήρχαν δύο κόμβοι που ήταν υπεύθυνοι για την ενημέρωση των αποτελεσμάτων που είχαν εξαχθεί, όπου ο ένας άλλαζε το χρώμα μιας λάμπας αναλόγως με τον αριθμό των λεπτών που απέμεναν μέχρι να έρθει η βροχή και ο δεύτερος απενεργοποιούσε μία άλλη λάμπα σε περίπτωση που ο χρόνος της αναμενόμενης βροχής ήταν αρκετά μικρός.

Το παραπάνω, είναι ένα ενδιαφέρον παράδειγμα χρήσης του Node-Red στο IoT, ενώ επιπλέον μας δείχνει τη δυνατότητα για επαναχρησιμοποίηση έτοιμων κόμβων. Από τη στιγμή που ο Bramley χρησιμοποίησε απλά ένα έτοιμο κόμβο για εκτέλεση HTTP request, όπου απλά έθετε το URL του API, θα μπορούσαμε πολύ απλά να το χρησιμοποιήσουμε για οτιδήποτε.

Οι ροές που δημιούργησε ο Bramley, βρίσκονται στη βιβλιοθήκη του Node-Red, και έτσι ο κάθε προγραμματιστής μπορεί να τις χρησιμοποιήσει με τον τρόπο που θέλει, να κάνει αλλαγές, ακόμα και να πάρει μόνο συγκεκριμένα μέρη, όπως για παράδειγμα ο έλεγχος του φωτός σε μία λάμπα.

Επιπλέον, ο κάθε προγραμματιστής έχει τη δυνατότητα να δημιουργήσει δικές του λειτουργίες, οι οποίες μπορούν να γίνουν ένας κόμβος που προστίθεται στο οικοσύστημα του Node-Red. Με αυτό τον τρόπο, δίνεται η δυνατότητα στους χρήστες να συνεισφέρουν οι ίδιοι στην πλατφόρμα που χρησιμοποιούν, ενώ επιτυγχάνεται μεγάλη ταχύτητα εξέλιξης αφού νέοι κόμβοι προστίθενται καθημερινά.

2.3.2.1 Πλεονεκτήματα και περιορισμοί του Node-red

Ένα ιδιαίτερα σημαντικό πλεονέκτημα που μας παρέχει το node-RED είναι ότι μπορεί να συνδυάσει υπηρεσίες web και hardware. Αποτελεί δηλαδή, ένα χρήσιμο εργαλείο για προγραμματισμό του IoT.

Το runtime του Node-Red είναι γραμμένο σε Node.js, και εκμεταλλεύεται πλήρως τα χαρακτηριστικά του. Αυτό το καθιστά ιδανικό για εκτέλεση σε φθηνές συσκευές όπως το Raspberry Pi ή στο cloud. Επιπλέον, μπορεί να εκτελεστεί σε διάφορες πλατφόρμες (on premise, cloud, συσκευές edge, container).

Βασικό πλεονέκτημα του Node-red είναι το γεγονός ότι το μέρος του κώδικα που απαιτείται να γράψει ο προγραμματιστής είναι ιδιαίτερα μικρό, από τη στιγμή που βασικές λειτουργίες παρέχονται αυτόματα με τη μορφή ενός απλού κόμβου. Επιπρόσθετα, η οικογένεια των κόμβων αυτών μεγαλώνει συνεχώς και όλοι οι χρηστές έχουν τη δυνατότητα να τους χρησιμοποιούν. Συνεπώς, ο προγραμματιστής αναλαμβάνει ρόλο ρυθμιστικό, από την άποψη ότι πρέπει να δώσει τα απαραίτητα στοιχεία σε διάφορα API για τη σωστή λειτουργία τους και επιπλέον, καλείται να γράψει συναρτήσεις σε JavaScript για την εκτέλεση πρόσθετων λειτουργιών.

Οι ροές είναι αποθηκευμένες ως αρχεία JSON, τα οποία μπορούν εύκολα να εξαχθούν και να εισαχθούν, διευκολύνοντας έτσι το διαμοιρασμό τους. Ωστόσο, δεν υπάρχουν δυαδικά αρχεία για διαμοιρασμό και εγκατάσταση σε συσκευές που δεν εκτελούν το Node-Red, το οποίο

σημαίνει ότι πιθανώς δεν είναι εφικτό να εκτελεστεί σε πολύ μικρές συσκευές που δεν διαθέτουν την δυνατότητα να εκτελέσουν το Node-red.

Αξίζει ακόμη να αναφερθεί, ότι το node-RED αποτελεί ιδιαίτερα ανεπτυγμένο framework, μιας και χρησιμοποιείται ήδη τα τελευταία πέντε χρόνια, και διαθέτει πλούσιο documentation και παραδείγματα. Επιπλέον, είναι εύκολο στην εγκατάσταση και στη δημιουργία μίας πρώτης ροής.

Ένας ακόμη παράγοντας που κάνει το Node-Red να ξεχωρίζει είναι το γεγονός ότι βρίσκει συνεχώς νέα πεδία εφαρμογής. Παρόλο που εισήχθη ως μία πλατφόρμα για IoT, σταδιακά φαίνεται να εισέρχεται δυναμικά στο χώρο της εκπαίδευσης, αφού πλέον σχολεία χρησιμοποιούν το Node-Red για να εισάγουν τα παιδιά στον προγραμματισμό, αφού η drag n drop φύση της πλατφόρμας το καθιστά αρκετά εύκολο.

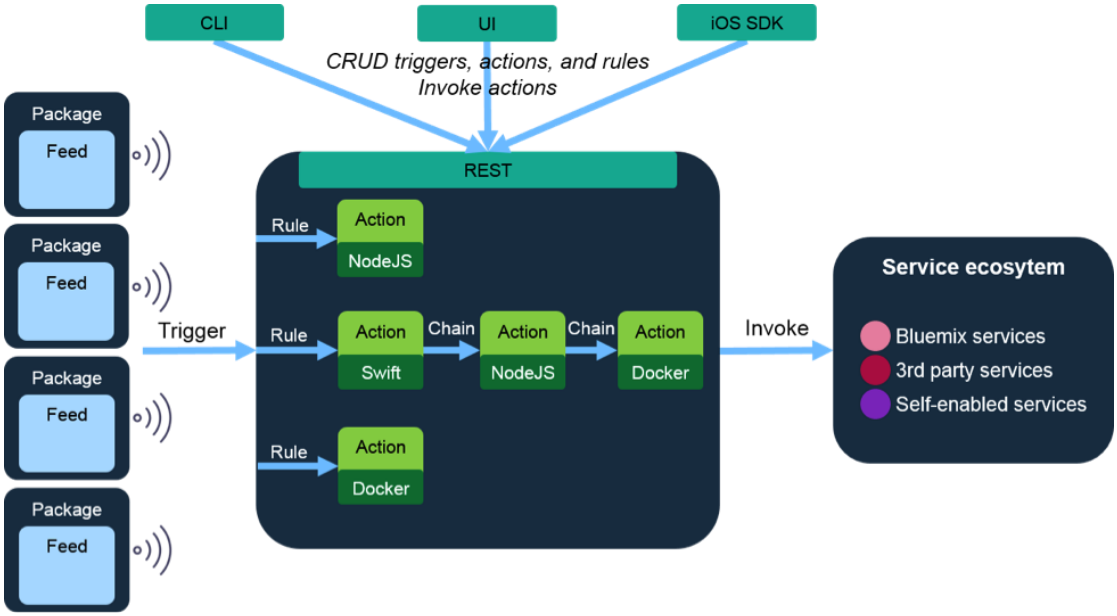
Ένας βασικός περιορισμός του node-RED είναι αυτός της γλώσσας αφού οι συναρτήσεις του γράφονται μόνο σε JavaScript, γεγονός που μειώνει το πλήθος των ενεργειών που μπορούμε να εκτελέσουμε.

2.3.2.2 Γνωστές εφαρμογές του Node-Red

Η IBM χρησιμοποιεί πλέον το Node-Red σε μεγάλο βαθμό. Στο Emerging Technologies lab στο Hursley της Αγγλίας, οι ερευνητές χρησιμοποίησαν μέσω του Node-Red, τον ίδιο ασύρματο διακόπτη, για να ανοιγοκλείνουν ένα ανεμιστήρα ή μία λάμπα. Αυτό στο Node-Red, ισοδυναμεί με μία απλή αλλαγή στη σύνδεση δύω κόμβων.

Άλλη μία ενδιαφέρουσα εφαρμογή από το ίδιο εργαστήριο, είναι ο χειρισμός ενός drone μέσω σφυρίγματος.

2.3.3 *Openwhisk*

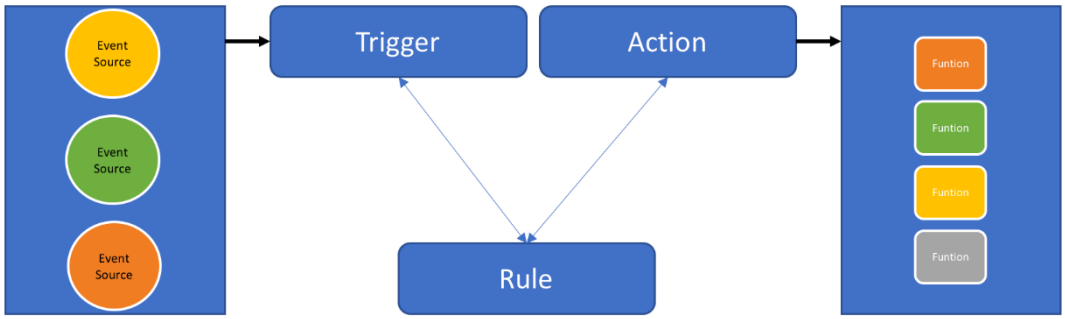


Εικόνα 2.8 Η ιδέα και αρχιτεκτονική του Openwhisk (IBM)

Το openwhisk αποτελεί μία open source πλατφόρμα για serverless προγραμματισμό, που εκτελεί κώδικα σε απάντηση σε κάποιο γεγονός (event). Ανακοινώθηκε το Φεβρουάριο του 2016, από την IBM. Είναι ενσωματωμένο στο Bluemix, την cloud πλατφόρμα της IBM, ενώ παρέχεται επίσης ως open source, για λειτουργία σε τοπικό σύστημα πίσω από το firewall. Επιπλέον, η Adobe μαζί με την IBM έχουν προσθέσει το Openwhisk στην Apache.

Αποτελείται από τα παρακάτω δομικά συστατικά:

- **Triggers:** Είναι ένα γεγονός (event) το οποίο «πυροδοτείται» όταν κάποια συνθήκη έρθει σε ισχύ. Μπορούν να συνδέονται με γεγονότα που πυροδοτούνται από εξωτερικές υπηρεσίες, όπως είναι μία αλλαγή σε έναν Cloudant πίνακα, ένα νέο μήνυμα στην ουρά του messaging hub, ένα commit στο GitHub, ή ένας αισθητήρας του IoT που στέλνει δεδομένα. Επίσης, trigger θα μπορούσαν να είναι και περιοδικοί συναγερμοί.
- **Action:** Είναι ο κώδικας που διαχειρίζεται το γεγονός (event handler). Γράφεται από τον προγραμματιστή και εκτελείται άμεσα μέσω HTTP κλήσης ή μέσω ενός trigger. Υποστηρίζει τις γλώσσες NodeJS (JavaScript), Python, Java, Swift, Scala, Go και όποια άλλη γλώσσα επιθυμεί ο προγραμματιστής ως εικονικό αρχείο docker.
- **Rules:** Είναι οι κανόνες εκτέλεσης των actions με βάση τους triggers. Καθορίζουν, δηλαδή, ποιο action θα εκτελεστεί όταν πυροδοτηθεί ένας trigger, ενώ πολλοί trigger μπορούν να αντιστοιχίζονται σε ένα action.
- **Sequences:** Είναι ακολουθίες από συνεχόμενα actions.
- **Packages:** Αποτελούν εξωτερικές υπηρεσίες.



Εικόνα 2.9 Openwhisk

Ο πιο απλός τρόπος να καταλάβει κανείς το Openwhisk, είναι αν το προσομοιάσει με ένα Pub/Sub σύστημα, όπου ο subscriber έχει τη δυνατότητα να ενσωματώσει και να εκτελέσει κώδικα. Οι triggers μπορούν να προσομοιωθούν με τους publisher, και οι actions με τους subscribers. Οι rules, παίζουν το ρόλο του θέματος που συνδέει τους subscribers με τους publishers.

Σημαντικό χαρακτηριστικό του openwhisk είναι η υποστήριξη API gateway, συνεπώς μπορούμε να εξάγουμε τα action ως API. Με αυτό τον τρόπο μπορούμε να εφαρμόσουμε αργότερα πολιτικές ασφάλειας και περιορισμό της απόδοσης, να παρακολουθήσουμε τη χρήση του API, και να καθορίσουμε πολιτικές διαμοιρασμού.

Το παραπάνω καθιστά και τη βασική διαφορά του Openwhisk, με τον ανταγωνιστή του, τη Lambda. Από τη στιγμή που το Openwhisk λειτουργεί μέσω API, οι triggers μπορούν να είναι οτιδήποτε που θα κάνει κλήση σε αυτό το API. Αντιθέτως, οι χρήστες της Lambda, είναι περιορισμένοι, αφού μπορούν να χρησιμοποιήσουν μόνο ότι τους παρέχει η Amazon ως trigger.

2.3.3.1 Πως λειτουργεί το Openwhisk;

Για να χρησιμοποιήσουμε το Openwhisk, μπορούμε είτε μέσω του Bluemix όπου γράφουμε τα actions μέσω browser, είτε με τοπική υλοποίηση στον υπολογιστή μας. Όσον αφορά το δεύτερο, μπορούμε να βρούμε πλούσιο documentation για την εγκατάσταση του. Υπάρχουν διάφοροι τρόποι για τοπική εγκατάσταση, είτε μέσω docker, είτε μέσω kubernetes είτε μέσω Virtual Machine και Vagrant.

Στη συνέχεια, μπορούμε να γράψουμε συναρτήσεις και να τις υλοποιήσουμε ως actions στο openwhisk, είτε μέσω του web editor, είτε σε αρχεία της γλώσσας της επιλογής μας. Εάν δουλεύομε σε τοπική υλοποίηση, τότε χρησιμοποιούμε το command line περιβάλλον του Openwhisk, το whisk cli, και μέσω αυτού εκτελούμε εντολές για: τον καθορισμό action, την εκτέλεση action, καθορισμό trigger και καθορισμό rule καθώς και άλλες ρυθμίσεις της πλατφόρμας όπως η IP του host που τρέχει το Openwhisk, το authentication και άλλα.

Για να καταλάβουμε καλύτερα τη λειτουργία του Openwhisk, θα δούμε αναλυτικά πως υλοποιούμε ένα action στο Openwhisk. Επομένως, έστω ότι έχουμε ένα αρχείο με όνομα action.js (σε local deployment).

```
function main() {
    console.log('Hello World');
    return { hello: 'world' };
}
```

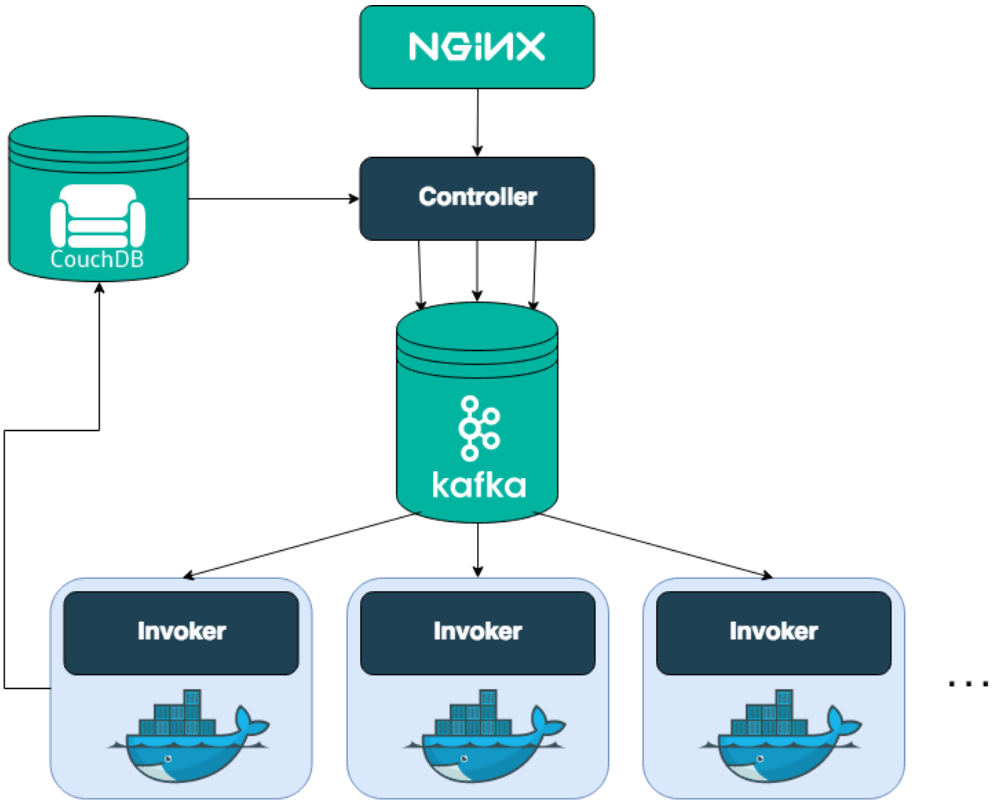
Το αρχείο περιέχει την παραπάνω main συνάρτηση, η οποία γράφει “Hello World” στην κονσόλα, και επιστρέφει ένα object με την τιμή “world” για key ‘hello”. Στη συνέχεια, δημιουργούμε το action στο whisk cli, εκτελώντας την ακόλουθη εντολή.

```
wsk action create myAction action.js
```

Το action έχει δημιουργηθεί, και για την εκτέλεση του εκτελούμε την επόμενη εντολή.

```
wsk action invoke myAction --result
```

Η παράμετρος –resutl χρησιμοποιείται ώστε η γραμμή εντολών να μας επιστρέψει το αποτέλεσμα της εκτέλεσης του action, ότι, δηλαδή, επιστρέφει η παραπάνω συνάρτηση.



Εικόνα 2.10 Openwhisk: behind the scenes

Είδαμε, λοιπόν, πως φτιάχνουμε και εκτελούμε ένα action. Τι συμβαίνει όμως στο παρασκήνιο; Αρχικά, το API του Openwhisk που βλέπει ο χρήστης είναι HTTP based, και σχεδιασμένο με RESTful design. Συνεπώς, κάθε εντολή στο whisk cli, μεταφράζεται σε ένα HTTP request στο σύστημα του Openwhisk. Επομένως, η εκτέλεση της action που κάναμε πριν ισοδυναμεί με τις παρακάτω εντολές:

```
POST /api/v1/namespaces/$userNamespace/actions/myAction
```

```
Host: $openwhiskEndpoint
```

Να σημειωθεί ότι η μεταβλητή $userNamespace αντιστοιχεί στο χρήστη που υλοποιεί το action.

Η πρώτη αυτή επαφή με το σύστημα, γίνεται με χρήση του NGIX, ένα HTTP και reverse proxy server, που χρησιμοποιείται κυρίως για SSL termination και προώθηση HTTP κλήσεων στο επόμενο μέρος.

Στη συνέχεια, το HTTP request προωθείται από το NGIX στον controller, ο οποίος είναι μία Scala-based υλοποίηση του REST API και χρησιμεύει ως τη διεπαφή για όλες τις λειτουργίες που μπορεί να κάνει ο χρήστης, όπως η εκτέλεση ενός action.

Ο controller αρχικά, προσπαθεί να καταλάβει τι προσπαθεί να κάνει ο χρήστης, ανάλογα με το είδος του request που έλαβε. Στο παράδειγμα μας λαμβάνει POST request σε ήδη υπάρχον action, οπότε καταλαβαίνει πως πρόκειται για εκτέλεση (invocation).

Ο controller ακολούθως, εξετάζει αν ο χρήστης έχει δικαίωμα εκτέλεσης του συγκεκριμένου action, ελέγχοντας τα παρεχόμενα credentials στη βάση δεδομένων Subjects, σε ένα instance της CouchDB. Εφόσον επιβεβαιωθεί η εγκυρότητα του χρήστη η εκτέλεση προχωράει στο επόμενο στάδιο.

Εδώ ο Controller φορτώνει το action από τη βάση δεδομένων whisks της CouchDB. Η εγγραφή που επιστρέφεται περιέχει τον προς εκτέλεση κώδικα, συγχωνευμένο με τις παραμέτρους εκτέλεσης (καμία στην περίπτωση μας) καθώς και τυχόν περιορισμούς (για παράδειγμα χρόνος εκτέλεσης).

Ο Load Balancer, ο οποίος αποτελεί μέρος του Controller, αναλαμβάνει να εντοπίσει ποιος «εκτελεστής» θα εκτελέσει το action. Αυτοί ονομάζονται invokers, και ο Load Balancer μπορεί να τους παρακολουθεί συνεχώς ως προς την κατάσταση που βρίσκονται, και ανάλογα με το ποιος είναι ελεύθερος του αναθέτει την εκτέλεση του action.

Στη προκειμένη φάση, υπάρχει μεγάλη πιθανότητα σφάλματος αν το σύστημα διακόψει τη λειτουργία του και χάσουμε την εκτέλεση του action, καθώς και πιθανότητα να εκτελούνται πολλές διεργασίες και να πρέπει να περιμένουμε αρκετά για την εκτέλεση του action. Τη λύση σε αυτό το πρόβλημα έρχεται να τη δώσει ένα σύστημα μηνυμάτων, η Kafka (“a high-throughput, distributed, publish-subscribe messaging system”). Η επικοινωνία μεταξύ Controller και Invoker γίνεται αποκλειστικά μέσω Kafka, ελαφρύνοντας έτσι τη μνήμη του συστήματος, και εξασφαλίζοντας ότι τα μηνύματα δε θα χαθούν σε περίπτωση διακοπής λειτουργίας του συστήματος.

Επομένως, όταν ο Controller θέλει να εκτελέσει ένα action, στέλνει ένα μήνυμα στην Kafka, που περιέχει το action που θα εκτελέσουμε και τις παραμέτρους εκτέλεσης. Το μήνυμα αυτό προωθείται στον Invoker που έχει επιλεγεί από τον Controller για την εκτέλεση του action. Όταν η Kafka ενημερωθεί πως έλαβε το μήνυμα, επιστρέφει το HTTP request με ένα

ActivationId το οποίο μπορεί να χρησιμοποιήσει στη συνέχεια ο χρήστης για να αποκτήσει πρόσβαση στο αποτέλεσμα του action.

Ωστόσο, την καθαρή εκτέλεση του action την εκτελεί ο Invoker, που ουσιαστικά είναι η καρδιά του Openwhisk. Αυτός έχει υλοποιηθεί σε Scala, και για να μπορεί να εκτελεί τα actions απομονωμένα και με ασφάλεια χρησιμοποιεί Docker.

Επομένως, για κάθε εκτέλεση ενός action, δημιουργείται ένα Docker container, στο οποίο εισάγεται ο κώδικας του action, και εκτελείται με βάση τις παραμέτρους που περάστηκαν σε αυτό. Το αποτέλεσμα επιστρέφεται και καταστρέφεται το container. Με αυτόν τον τρόπο μπορούν να γίνουν αρκετές βελτιώσεις απόδοσης ώστε να έχουμε πολύ χαμηλό χρόνο εκτέλεσης.

Τελικά το αποτέλεσμα αποθηκεύεται στη βάση δεδομένων με όνομα activations της CouchDB, με το συγκεκριμένο ActivationId που αναφέραμε παραπάνω. Στο παράδειγμα μας, ο Invoker παίρνει το αποτέλεσμα σε μορφή JSON object και τα logs από το Docker, φτιάχνει μία εγγραφή activation, και την εισάγει στη βάση δεδομένων. Αυτό θα έχει την ακόλουθη μορφή:

```
{
   "activationId": "31809ddca6f64cfc9de2937ebd44fbb9",
   "response": {
       "statusCode": 0,
       "result": {
           "hello": "world"
       }
   },
   "end": 1474459415621,
   "logs": [
       "2016-09-21T12:03:35.619234386Z stdout: Hello World"
   ],
   "start": 1474459415595,
}
```

Βλέπουμε πως πέρα από το log του Docker, έχουμε κι άλλες πληροφορίες όπως η χρονική στιγμή έναρξης εκτέλεσης, καθώς υπάρχουν και περισσότερες τις οποίες παραλείψαμε, γιατί δεν παρουσιάζουν κάποιο ενδιαφέρον την προκειμένη στιγμή.

Τέλος, ο χρήστης μπορεί να εκτελέσει την ακόλουθη εντολή, ώστε να λάβει μόνο το αποτέλεσμα του action:

```
wsk activation get 31809ddca6f64cfc9de2937ebd44fbb9
```

2.3.3.2 Πλεονεκτήματα του Openwhisk

Το Openwhisk έχει κάποια σημαντικά πλεονεκτήματα, τα οποία το κάνουν να διαφέρει σε μεγάλο βαθμό από τις άλλες serverless πλατφόρμες, και αυτά είναι:

- Το openwhisk διαχειρίζεται αυτόματα σημαντικές λεπτομέρειες της εκτέλεσης του κώδικα, όπως την επεκτασιμότητα (scaling), την εξισορρόπηση του φόρτου εργασίας

(load balancing), το logging, η αντιμετώπιση και ανεκτικότητα των σφαλμάτων (fault tolerance) και οι ουρές μηνυμάτων (message queues).

- Επιτρέπει τη συγγραφή κώδικα σε διαφορετικές γλώσσες, αφού οι προγραμματιστές μπορούν να γράψουν τις συναρτήσεις τους (actions στο openwhisk) σε JavaScript, Python, Java, και Swift, ή να πακετάρουν τον κώδικα τους σε όποια άλλη γλώσσα επιθυμούν ως εικονικό αρχείο docker.
- Είναι open source, και βασίζεται σε δοκιμασμένες open source υποδομές όπως Docker, Kafka, Consul και Akka και μπορούν να επεκταθούν με νέες γλώσσες και δυνατότητες.
- Παρέχει τη δυνατότητα για υλοποίηση σε δημόσια, ιδιωτικά και υβριδικά μοντέλα επιτρέποντας έτσι την πρόσβαση της serverless αρχιτεκτονικής και εκτός των ήδη παρεχόμενων δημοσίων cloud υπηρεσιών.
- Βασίζεται σε ένα ανοικτό οικοσύστημα που υποστηρίζει και επιτρέπει το διαμοιρασμό των microservices μέσω των packages του Openwhisk.
- Παρέχει ένα πλούσιο περιβάλλον από δομικά στοιχεία που προέρχονται από πολλούς και διαφορετικούς τομείς (analytics, cognitive, data, IoT, κ.τ.λ.).
- Αποκρύπτει την πολυπλοκότητα των υποδομών και δίνει, με αυτό τον τρόπο, στους προγραμματιστές, τη δυνατότητα να επικεντρωθούν στην επιχειρηματική λογική.
- Παρέχει ένα νέο μοντέλο κοστολόγησης, όπου «χρεωνόμαστε» με την κλήση, και όχι με την ώρα όπως το παραδοσιακό μοντέλο.

3

Προβλήματα και λύσεις

Όπως έχουμε αναφέρει στα προηγούμενα κεφάλαια, το IoT αναπτύσσεται ραγδαία, και η χρήση του επεκτείνεται συνεχώς. Ο όγκος των δεδομένων που οι συσκευές καλούνται να διαχειριστούν είναι ολοένα και μεγαλύτερος, και επομένως οι απαιτήσεις σε επίπεδο απόδοσης και επίδοσης είναι υψηλές. Για το λόγο αυτό, οι νέες τεχνολογίες που αναλύσαμε προηγουμένως έρχονται να παίξουν σπουδαίο ρόλο.

3.1 Προκλήσεις στο IoT

Σύμφωνα με το άρθρο "The Programming Challenges of IoT" (Esposito, 2014) υπάρχουν δύο τύπου προκλήσεων: Data and control και Information and business logic.

3.1.1 Data and control

Power

Το πρόβλημα αυτό είναι προφανές. Οι περισσότερες IoT συσκευές είναι ασύρματες, πράγμα που σημαίνει μεγάλη κατανάλωση ενέργειας. Η μία λύση είναι να δημιουργούνται αποδοτικά οι αλγόριθμοι ώστε να μη γίνεται χρήση επεξεργαστή χωρίς να εκτελείται κάποια εργασία. Η δεύτερη λύση είναι πιο περίπλοκη, και αφορά την απενεργοποίηση των συσκευών όταν δε χρησιμοποιούνται, και την επαναχρησιμοποίησή τους σε αντίθετη περίπτωση.

Latency

Το κύριο πρόβλημα εδώ εντοπίζεται στο hardware. Τα chips των IoT συσκευών είναι συνήθως πολύ μικρά, πράγμα που σημαίνει ότι δε μπορούν να έχουν τόσες δυνατότητες όσες επιτρέπει η τωρινή τεχνολογία των τρανζίστορ.

Ένας άλλος λόγος για αυτό το πρόβλημα είναι η υποδομή του δικτύου από τη στιγμή που ο αριθμός των IoT συσκευών αυξάνεται, το διαθέσιμο bandwidth περιορίζεται. Αυτό είναι ακόμα

πιο έντονο, από τη στιγμή που οι περισσότερες συσκευές καταναλώνουν πόρους, ακόμα κι αν δεν εκτελούν κάποια εργασία.

Unreliability

Οι συσκευές IoT είναι συνήθως φθηνές, και είναι πιο εύκολο να εμφανίσουν δυσλειτουργίες. Συνεπώς, η αξιοπιστία των αποτελεσμάτων τους είναι αμφισβητήσιμη και απαιτείται έλεγχος.

3.1.2 Information and Business logic

Vast and thin data

Ο όγκος των δεδομένων που παράγονται από IoT συσκευές είναι ιδιαίτερα μεγάλος, με αποτέλεσμα η επεξεργασία τους να είναι ιδιαίτερα δύσκολη, καθώς και η εξαγωγή χρήσιμων αποτελεσμάτων. Αυτό επιπλέον, συνεπάγεται ότι απαιτείται η χρήση νέων αλγορίθμων και δομών δεδομένων, που οι προγραμματιστές πιθανώς να μην είναι ιδιαίτερα εξοικειωμένοι.

3.2 Προκλήσεις στο cloud computing

Ασφάλεια των δεδομένων και ιδιωτικότητα

Από την στιγμή που τα δεδομένα βρίσκονται στο δίκτυο μίας εταιρίας, υπάρχει ρίσκο για την ιδιωτικότητα τους.

Έλλειψη τυποποίησης

Οι πάροχοι cloud υπηρεσιών λειτουργούν χωρίς να υπάρχουν καθορισμένες οδηγίες για αυτούς. Αυτό μπορεί να προκαλέσει σύγχυση και προβλήματα ασφαλείας, αφού κάθε πάροχος χρησιμοποιεί και διαφορετικά πρωτόκολλα.

Ιδιοκτησία δεδομένων

Αποτελεί βασική πρόκληση του cloud computing. Το πρόβλημα έγκειται στο γεγονός ότι με την παραχώρηση των δεδομένων ενός χρήστη σε έναν πάροχο cloud, αυτός χάνει την ιδιοκτησία τους. Οι περισσότερες εταιρίες, δηλώνουν στα συμβόλαια τους, ότι προχωρούν σε αυτή την ενέργεια, ώστε να έχουν μεγαλύτερη νομική προστασία, και έτσι να διατηρούν τα δεδομένα των χρηστών ασφαλή από εξωτερικές απειλές.

3.3 Στόχος της διπλωματικής

Στόχος της παρούσας διπλωματικής εργασίας είναι να διερευνήσουμε τις εφαρμογές του cloud computing και της FaaS αρχιτεκτονικής στον τομέα του IoT. Θα προσπαθήσουμε να παρουσιάσουμε τα πλεονεκτήματα των νέων τεχνολογιών, σε μία πλατφόρμα IoT, η οποία είναι

βασισμένη στο Node-Red, ώστε να μας βοηθήσουν να ξεπεράσουμε κάποια βασικά προβλήματα.

Βασικός περιορισμός στο Node-Red είναι αυτός της γλώσσας, από τη στιγμή που μας δίνει δυνατότητα για συγγραφή μόνο σε Node.JS. Επιπλέον, έχουμε να αντιμετωπίσουμε το πρόβλημα του scalability, αφού για μεγάλες και δύσκολες ενέργειες το Node-Red δεν ανταποκρίνεται ικανοποιητικά.

3.4 Συμβολή του Openwhisk

Στη λύση που παρουσιάζουμε, θα κάνουμε χρήση του Openwhisk. Η επιλογή μας για χρήση serverless, FaaS πλατφόρμας είναι προφανής, αν αναλογιστούμε τα πλεονεκτήματα που μας παρέχουν.

Ένα ακόμη βασικό πρόβλημα, που θα αντιμετωπίσουμε, είναι ότι θέλουμε να δημιουργήσουμε μία εφαρμογή η οποία δε θα επιβαρύνει τη συσκευή που χρησιμοποιούμε, ώστε να μπορεί να εκτελεστεί και σε ελαφριά μηχανήματα (όπως το raspberry). Για το λόγο αυτό, επιλέγουμε τη χρήση FaaS αρχιτεκτονική;, έτσι ώστε να μειώσουμε σημαντικά την υπολογιστική πολυπλοκότητα και τη χρήση των πόρων του συστήματος. Φροντίζουμε συνεπώς, οι event driven ενέργειες να εκτελούνται με βάση αυτά.

Αναφερόμενοι στο ίδιο ζήτημα, βασικός παράγοντας για το πόσο ελαφριά θα είναι η εφαρμογή μας είναι η επεκτασιμότητα. Όταν ερχόμαστε αντιμέτωποι με δύσκολες και χρονοβόρες πράξεις, το Node-Red δεν δίνει γρήγορα (και πιθανώς καθόλου) αποτελέσματα. Αντιθέτως, η FaaS τεχνολογία, και συγκεκριμένα το Openwhisk, εξασφαλίζει σημαντική βελτίωση του scaling.

Η τεχνολογία της FaaS αρχιτεκτονικής, επίσης, μας βοηθάει στην αντιμετώπιση του γλωσσικού παράγοντα, αφού μας δίνει τη δυνατότητα για προγραμματισμό εκτός του Node.JS της Node-Red, επεκτείνοντας με αυτό τον τρόπο το εύρος των δυνατοτήτων ως προς τις ενέργειες που μπορούμε να πραγματοποιήσουμε.

Όσον αφορά τους λόγους που επιλέξαμε να χρησιμοποιήσουμε το Openwhisk, και όχι κάποια άλλη FaaS πλατφόρμα, είναι διότι μας παρέχει περισσότερες δυνατότητες. Το Openwhisk αποτελεί open source πλατφόρμα, και μας επιτρέπει να χρησιμοποιήσουμε εξωτερικά ερεθίσματα ως έναυσμα για τις ενέργειες που εκτελεί, γεγονός που το κάνει να ξεχωρίζει σε σχέση με τους αντιπάλους του, όπως π.χ. η Lambda που μας περιορίζει σε όσα μας παρέχει η Amazon.

3.5 Η αρχιτεκτονική της λύσης

Βάση της εφαρμογής αποτελεί το Node-Red. Εκεί δημιουργήσαμε τις ροές κατά τις οποίες θα εκτελούμε find queries σε μία Mongo όπου έχουμε αποθηκευμένα δεδομένα από το yelp, και θα μετράμε τον χρόνο ανταπόκρισης, κάνοντας αποκλειστικά χρήση είτε του Node-Red είναι του openwhisk, για την εκτέλεση του query. Στη συνέχεια θα πραγματοποιούμε στατιστική ανάλυση με βάση τους χρόνους που συλλέξαμε, ώστε να διαπιστώσουμε ποια από τις δύο πλατφόρμες έχει ταχύτερη ανταπόκριση.

Στο τεχνικό κομμάτι, εκτελούμε μέσω του Openwhisk μεγάλο μέρος της στατιστικής ανάλυσης, με την έννοια ότι κάθε νέα εγγραφή στη βάση δεδομένων, θα δίνει το έναυσμα για την ανάλυση. Επιπλέον δίνεται η δυνατότητα για χρήση διαφορετικών γλωσσών, σε αυτό το σημείο, εκμεταλλευόμενοι με αυτό τον τρόπο διευκολύνσεις που μας δίνουν άλλες γλώσσες για στατιστική ανάλυση. Εμείς, υπολογίζουμε τις μετρικές με JavaScript και Python.

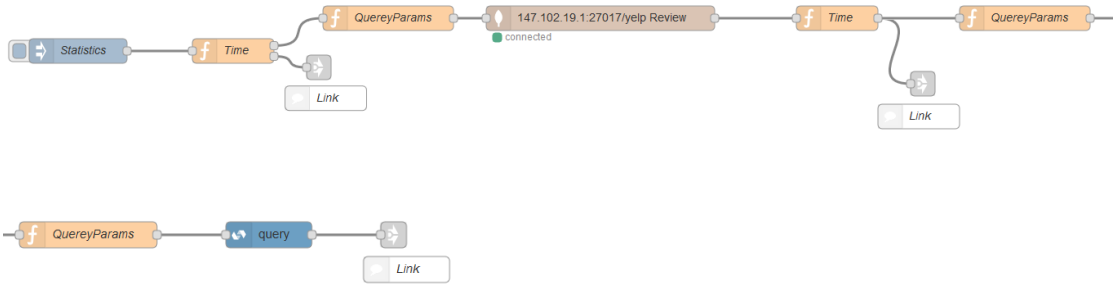
Τέλος, για να συγκρίνουμε το scalability του Node-Red με αυτό του Openwhisk, δίνουμε τη δυνατότητα για καθορισμό του αριθμού των εγγραφών που το query θα επιστρέψει, έτσι ώστε να εξετάσουμε τους χρόνους ανταπόκρισης, σε όλο και μεγαλύτερο όγκο δεδομένων.

4

Τεχνική Ανάλυση

Στο κεφάλαιο αυτό θα παρουσιάσουμε αναλυτικά τις ροές που δημιουργήσαμε στο Node-Red, καθώς και τα actions και triggers που δημιουργήσαμε στο Openwhisk, και τις λειτουργίες που επιτελούν.

4.1 Διαδικασία Συλλογής Δεδομένων



Εικόνα 4.1 Ροή συλλογής δεδομένων

Ο κόμβος inject, ανά τακτά χρονικά διαστήματα, εκκινεί τη ροή, εκτελούνται τα δύο queries, και μετράμε τον χρόνο απόκρισης.

Ο χρόνος απόκρισης για το query σε node-red υπολογίζεται με τον ακόλουθο τρόπο: Προσθέσαμε δύο επιπλέον κόμβους συνάρτησης, έναν αμέσως πριν από τον κόμβο mongo, και έναν αμέσως μετά, και σε αυτούς υπολογίζουμε την ακριβή ημερομηνία τη χρονική στιγμή αυτή σε ms, και έπειτα, υπολογίζουμε την διαφορά αυτών των δύο τιμών, που είναι και ο χρόνος (σε ms) για τον οποίο εκτελείται ο ενδιάμεσος κόμβος. Ο κώδικας των κόμβων time είναι:

```
var time = {payload: {Time: new Date().valueOf()}};
return [msg, time];
```

Ο πρώτος κόμβος “Time” εκτελεί τον κώδικα που αναφέρθηκε παραπάνω. Ομοίως και ο δεύτερος, με τη διαφορά ότι, προσθέτει ένα όνομα, ώστε να γνωρίζουμε σε ποιο κόμβο αντιστοιχεί κάθε τιμή, καθώς και τον αριθμό των εγγραφών που επιστρέφει το query.
Ο κόμβος mongo, είναι αυτός που εκτελεί το find query στη βάση, και μετράμε την απόκριση του. Ο κόμβος QueryParams περνάει τις απαραίτητες παραμέτρους για την εκτέλεση του query.Ο κώδικας είναι:

```
msg = {stars : 5};
msg.limit = context.global.lim;
return msg;
```

Οι γκρι κόμβοι, μετά τους κόμβους συνάρτησης “Time” είναι τύπου link, και ο ρόλος τους είναι η αποστολή δεδομένων στην επόμενη ροή για τον υπολογισμό της διαφοράς και την αποθήκευση της. Ωστόσο, ο τελευταίος κόμβος link μας παραπέμπει στη ροή CountTime η οποία χρησιμεύει για την επιστροφή του χρόνου εκτέλεσης του query σε mongo μέσω openwhisk.
Ο κόμβος query, είναι ένας κόμβος openwhisk που εκτελεί το action, που έχουμε φτιάξει ήδη στο openwhisk, με όνομα query. Ο κώδικας του action είναι ο ακόλουθος:

```
function main(params){
var MongoClient = require('mongodb').MongoClient
var url = 'mongodb://147.102.19.1:27017/'
return new Promise(function(resolve, reject) {
   MongoClient.connect(url, (err, db) => {

db.db('yelp').collection('Review').find({stars:5}).limit(param
s.lim).toArray().then((docs) => {
           resolve({result : docs});
           db.close();
       }).catch((err) => {
           reject(err)
       });
   })
})
}
```

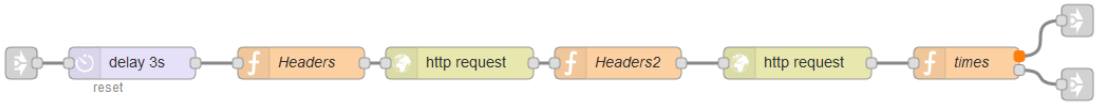
Και αντίστοιχα ο κώδικας της QueryParams:

```
msg.payload = {lim: context.global.lim};
return msg;
```

Σε αυτό το σημείο πρέπει να σημειωθεί ότι το query που εκτελούμε επιστρέφει όλες τις επιχειρήσεις που έχουν βαθμολογία 5 αστεριών στη βάση του yelp (stars:5). Επιπλέον, ο

αριθμός των εγγραφών που θέλουμε να επιστρέψει το query είναι ίσο με την τιμή της global μεταβλητής lim, η οποία καθορίζεται από το χρήστη, όπως θα δούμε στη συνέχεια.

Τέλος, η διαδικασία με την οποία μετράμε το χρόνο εκτέλεσης του query από το Openwhisk φαίνεται στην παρακάτω ροή.



Εικόνα 4.2 Ροή υπολογισμού χρόνου εκτέλεσης για το Openwhisk

Σε αυτή τη ροή, εκτελούμε HTTP requests στο API του Openwhisk ώστε να λάβουμε το χρόνο εκτέλεσης της τελευταίας action που εκτελέστηκε (στην προκειμένη περίπτωση του query). Ο λόγος που πραγματοποιούμε αυτή τη διαδικασία και όχι αυτή που κάνουμε για τον υπολογισμό του χρόνου στο Node-Red είναι ο εξής. Εάν χρησιμοποιούσαμε την προηγούμενη διαδικασία, θα είχαμε ένα μεγάλο σφάλμα, αφού ο χρόνος που θα βρίσκαμε θα συνυπολόγιζε και το χρόνο επικοινωνίας του openwhisk με το node-red, ο οποίος θα μπορούσε να είναι αρκετά μεγάλος, αν αναλογιστεί κανείς ότι το Openwhisk εκτελείται στο cloud, οπότε η επικοινωνία του με το Node-Red εξαρτάται από την ποιότητα του δικτύου. Επομένως, μέσω HTTP request επιτυγχάνουμε ακρίβεια στον υπολογιστικό χρόνο, καθαρά στο Openwhisk, το οποίο είναι και αυτό που μας ενδιαφέρει.

Στην αρχή της ροής βλέπουμε ένα κόμβο link, ο οποίος συνεχίζει τη ροή από τον τελευταίο κόμβο link, στην πρώτη ροή.

Στη συνέχεια κάνουμε χρήση ενός κόμβου delay, ο οποίος σταματάει την εκτέλεση για 3 sec. Προχωρήσαμε στην επιλογή αυτή, διότι μετά από δοκιμές, παρατηρήσαμε, ότι το API του Openwhisk απαιτεί λίγο χρόνο να ανανεωθεί, ώστε να λάβουμε τα σωστά αποτελέσματα.

Στη συνέχεια εκτελούμε δύο διαφορετικά HTTP request, όπου το πρώτο μας επιστρέφει το τελευταίο activation, δηλαδή το id της τελευταίας action που εκτελέστηκε. Το επόμενο request, μας επιστρέφει όλες τις πληροφορίες για το activation με id αυτό που λάβαμε από το πρώτο request. Στις πληροφορίες που λαμβάνουμε, υπάρχουν οι παράμετροι start και end, που είναι οι χρόνοι έναρξης και λήξης της εκτέλεσης του action, τις οποίες και στέλνουμε στην επόμενη ροή για αποθήκευση στη βάση.

Οι κόμβοι Headers περιέχουν όλες τις απαραίτητες παραμέτρους για την εκτέλεση του request, δηλαδή το authentication του API του Openwhisk και το url. Οι κώδικες από τις δύο συναρτήσεις Headers είναι οι ακόλουθοι.

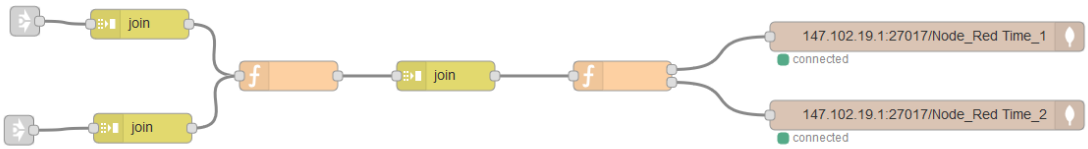
```
msg.headers = {
    "Authorization": "*****",
    "Content-Type": "application/json"
};
msg.url='http://<apihost>/api/v1/namespaces/<your_namespace>/a
ctivations?limit=1';
return msg;,
```

```
var id = msg.payload[0].activationId;
msg.payload=msg.payload[0];
msg.headers = {
    "Authorization": "*****",
    "Content-Type": "application/json"
};
msg.url =
'http://<apihost>/api/v1/namespaces/<your_namespace>/activatio
ns/'+id;
return msg;
```

Τέλος, ο κόμβος συνάρτησης times, παίνρει τις τιμές που θέλουμε από το request και τις επιστρέφει ως outputs.

```
var start={payload:{Time: msg.payload.start}};
var end={payload:{Time: msg.payload.start, Topic: "Openwhisk",
Docs:context.global.lim}};
return [start,end];
```

4.2 *Διαδικασία αποθήκευσης μετρήσεων*



Εικόνα 4.3 Ροή αποθήκευσης μετρήσεων

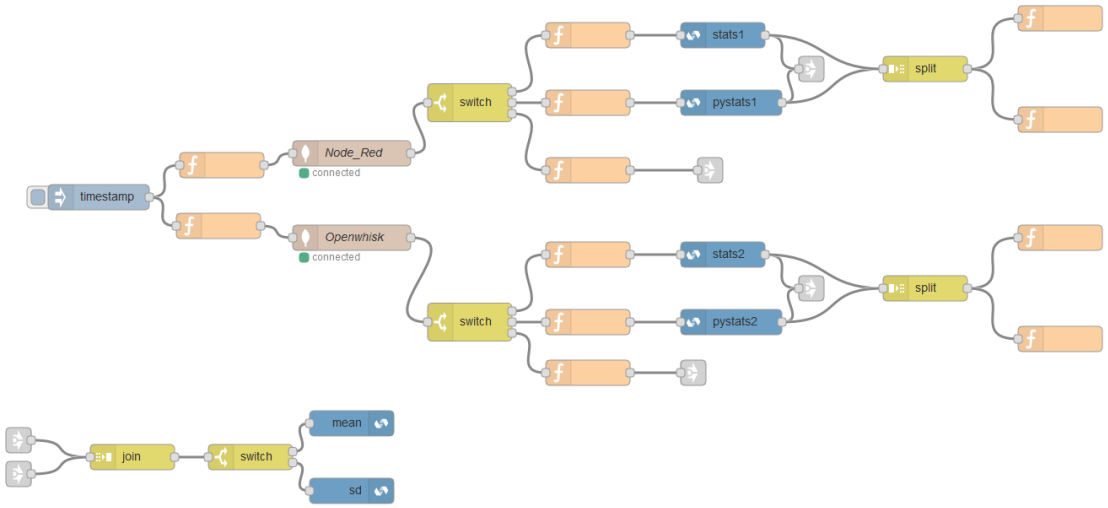
Για να υπολογιστεί ο χρόνος, όταν είναι έτοιμες και οι δύο τιμές, χρησιμοποιήθηκαν κόμβοι τύπου join, οι οποίοι αναμένουν στην είσοδο τους ένα συγκεκριμένο αριθμό μηνυμάτων.

Όταν και οι δύο τιμές φτάσουν στον κόμβο join η ροή συνεχίζεται, και έπειτα, με έναν ακόμα κόμβο συνάρτησης υπολογίζεται η διαφορά. Ο επόμενος κόμβος join περιμένει να υπολογιστούν οι τιμές και για τις δύο πηγές.

Στη συνέχεια, πραγματοποιείται η αποθήκευση των μετρήσεων στην βάση δεδομένων. Ο κόμβος συνάρτησης, μετατρέπει απλώς τα δεδομένα σε κατάλληλη μορφή, ώστε να αποθηκευτούν στη βάση δεδομένων.

Για τη βάση δεδομένων, έχουμε τα ακόλουθα: Node_Red είναι το όνομα της βάσης και Time_1 και Time_2 οι συλλογές για τους χρόνους που μετράμε, από τις δύο πηγές αντίστοιχα.

4.3 Εύρεση στατιστικών



Εικόνα 4.4 Ροή υπολογισμού στατιστικών

Για την εύρεση την καλύτερης πηγής χρησιμοποιήσαμε δύο διαφορετικούς τρόπους. Ο πρώτος χρησιμοποιεί σαν μετρική το μέσο του συνόλου των δεδομένων που υπάρχουν στην βάση, ανάλογα με τον αριθμό των εγγραφών. Συνεπώς, η πηγή με τον χαμηλότερο μέσο όρο θεωρείται καλύτερη.

Ο δεύτερος τρόπος χρησιμοποιεί την τυπική απόκλιση των μετρήσεων, και θεωρούμε καλύτερη την πηγή με τη χαμηλότερη τυπική απόκλιση.

Και στους δύο τρόπους, για τον υπολογισμό των μετρικών δίνουμε την επιλογή για υπολογισμό με τρείς διαφορετικές μεθόδους. Η πρώτη, είναι απλός υπολογισμός μέσω του node-red, η δεύτερη μέσω action στο openwhisk σε JavaScript και η τρίτη μέσω action στο Openwhisk σε python.

Στη ροή αυτή, αρχικά, εκτελούμε ένα find στη mongo (ένα σε κάθε συλλογή για Node-red και Openwhisk), για συγκεκριμένο αριθμό εγγραφών, ώστε να μπορούμε να κάνουμε συγκρίσεις. Να σημειωθεί ότι εδώ αντλούμε δεδομένα από τη βάση δεδομένων που φτιάξαμε στη ροή CreateFile. Στους κόμβους συνάρτησης, καθορίζουμε τις παραμέτρους, που είναι ο αριθμός των επιστρεφόμενων εγγραφών, καθώς και την παράμετρο Docs που είναι το φίλτρο της αναζήτησης μας, ώστε να επιστρέψει δεδομένα χρόνου για συγκεκριμένο αριθμό εγγραφών στη βάση του yelp.

Στη συνέχεια, ο κόμβος swtich επιλέγει σε ποια γλώσσα θα υπολογιστούν οι μετρικές, ανάλογα με το τι έχει επιλέξει ο χρήστης.

Στην περίπτωση του Node-Red, υπολογίζονται οι μετρικές, αποθηκεύονται σε global μεταβλητές, και η εκτέλεση συνεχίζει στον κάτω κόμβο link. Στην περίπτωση του openwhisk, είτε σε javascript είτε σε python, έχουμε αρχικά ένα κόμβο συνάρτησης ώστε να δώσουμε τα

δεδομένα στη μορφή που απαιτεί το Openwhisk. Στη συνέχεια, ο κόμβος split, διαχωρίζει τις δύο τιμές που δίνει ως έξοδο το Openwhisk (μέσο όρο και τυπική απόκλιση) ώστε να αποθηκευτούν σε global μεταβλητές, και η εκτέλεση συνεχίζει, όπως και με το Node-Red, στους κάτω κόμβους link.

Παρακάτω φαίνεται ο κώδικας της συνάρτησης του node-red καθώς και τα δύο actions στο Openwhisk.

Κώδικας στο Node-Red:

```
var sum = 0;
for (var i = 0; i < msg.payload.length; i++) {
      sum += msg.payload[i].Time;
}

avg = sum/msg.payload.length;
sum = 0;

for (var i = 0; i < msg.payload.length; i++) {
      sum += Math.pow(msg.payload[i].Time - avg,2);
}

var s = Math.sqrt(sum/(msg.payload.length - 1));

context.global.mean1 = avg;
context.global.sd1 = s;
```

Action σε JavaScript:

```
function avg(params){
    var s=0;
    var count=Object.keys(params).length;
    for(var key in params) {
      if (params.hasOwnProperty(key)) {
        s += parseInt(params[key]);
      }
    }
    return s/count;
}

function ta(params,avg){
    var s=0;
    var count=Object.keys(params).length;
    s=0;
    for(var key in params){
      if (params.hasOwnProperty(key)){
```

```
        s += Math.pow(parseInt(params[key])-avg,2);
      }
    }
    return Math.sqrt(s/(count-1));
}

function main(params){
    var a=avg(params);
    var b=ta(params,a);
    return {payload : {mean1 : a, sd1 : b}};
}
```

Action σε Python:

```
from __future__ import division
def main(args):
        import json
        arr=json.dumps(args)
        a=json.loads(arr)

        s=0;
        for x in a['params']:
                s+=x['Time']
        mean1=s/len(a['params'])

        s1=0
        for x in a['params']:
                s1+=(x['Time']-mean1)**2
        import math
        sd1=math.sqrt(s1/(len(a['params'])-1))

        return {"payload":{"mean1":mean1,"sd1":sd1}}
```

Τα παραπάνω είναι όμοια και για τις περιπτώσεις του Node-Red και Openwhisk, απλά επεξεργάζονται διαφορετικά δεδομένα.

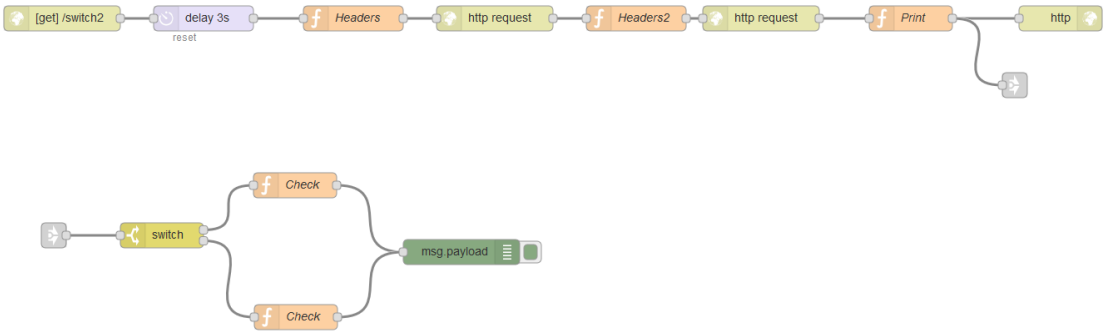
Τέλος, οι κάτω κόμβοι link καταλήγουν σε ένα κόμβο join, οποίος περιμένει να λάβει δύο μηνύματα, εξασφαλίζοντας έτσι ότι έχουμε υπολογίσει τις μετρικές και για τις δύο περιπτώσεις, και ανάλογα με το κριτήριο που έχει επιλέξει ο χρήστης, μέσος όρος ή τυπική απόκλιση, ο κόμβος switch οδηγεί στην εκτέλεση του αντίστοιχου Openwhisk trigger, ο οποίος εκτελείται, και αυτομάτως η εκτέλεση συνεχίζει στην επόμενη ροή, εξ αιτίας του Openwhisk action που εκτελέστηκε.

4.4 Εμφάνιση καλύτερης πηγής

Η προηγούμενη ροή καταλήγει στην εκτέλεση ενός trigger στο Openwhisk, ο οποίος με τη σειρά του εκτελεί ένα συγκεκριμένο action. Έχουμε δύο διαφορετικούς trigger, και δύο διαφορετικά action αντίστοιχα, ανάλογα με την επιλογή μετρικής από το χρήστη. Ο trigger mean εκτελεί το action με όνομα choice1 και, αντίστοιχα, ο trigger sd εκτελεί το action με όνομα choice 2. Ο κώδικας του choice1 είναι ο ακόλουθος.

```
var request = require('request');
function main(params){
    var choice = 1;
    request('http://<node-red-host>:1880/switch', function(err,
res, body) {});
    return({'choice':choice});
}
```

Για το action choice2 έχουμε ίδιο κώδικα απλά θέτουμε ως choice την επιλογή 2. Αυτό που κάνει αυτό το action, είναι ένα http request στο url που φαίνεται παραπάνω, το οποίο έχει ως αποτέλεσμα τη συνέχεια της εκτέλεσης στη ροή switch (εικόνα 4.5).



Εικόνα 4.5 Ροή εμφάνισης καλύτερης πηγής

Ο πρώτος κόμβος, στο πάνω σκέλος της ροής αυτής, θέτει ως url στον κόμβο αυτό το http://<node-red-host>:1880/switch, ώστε όταν το action κάνει request, να συνεχίσει η εκτέλεση από τον κόμβο αυτό.

Στη συνέχεια, για να βρούμε το αποτέλεσμα από το action που εκτελέστηκε στο openwhisk, εκτελούμε δύο διαδοχικά HTTP requests στο API του Openwhisk. Αρχικά, εντοπίζουμε το τελευταίο activation, και πρακτικά το activation id της τελευταίας action που εκτελέστηκε. Στη συνέχεια, με βάση το id που βρήκαμε, κάνουμε άλλη μία κλήση για να βρούμε το αποτέλεσμα

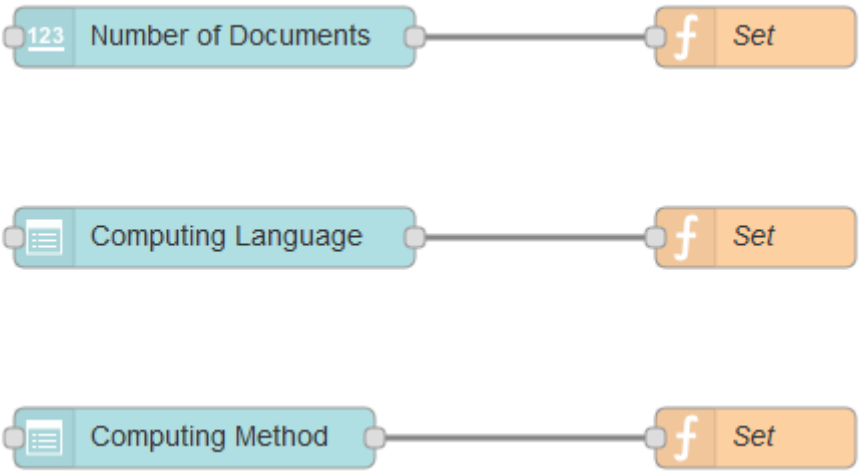
του action. Τελικά αυτό το αποτέλεσμα, το διοχετεύουμε στη συνέχεια της ροής μέσω του κόμβου link.

Τέλος, το κάτω μέρος της παραπάνω ροής, ελέγχει για το ποια επιλογή μετρικής έχει γίνει από το χρήστη, και αναλόγως ελέγχουμε ποια έχει τη μικρότερη τιμή και εμφανίζουμε το αποτέλεσμα. Ο κώδικας των συναρτήσεων check φαίνεται παρακάτω.

```
var x = context.global.mean1 - context.global.mean2;
var best="";
var mean=0;
if (x <= 0) {
    best="Node-Red"
    mean=context.global.mean1;
} else {
    best="Openwhisk"
    mean=context.global.mean2;
}
msg.payload={"best":best,"mean":mean}
return msg;
```

Ο κώδικας είναι όμοιος και στις δύο συναρτήσεις, απλώς στην κάτω, έχουμε όπου mean1 και mean2, sd1 και sd2.

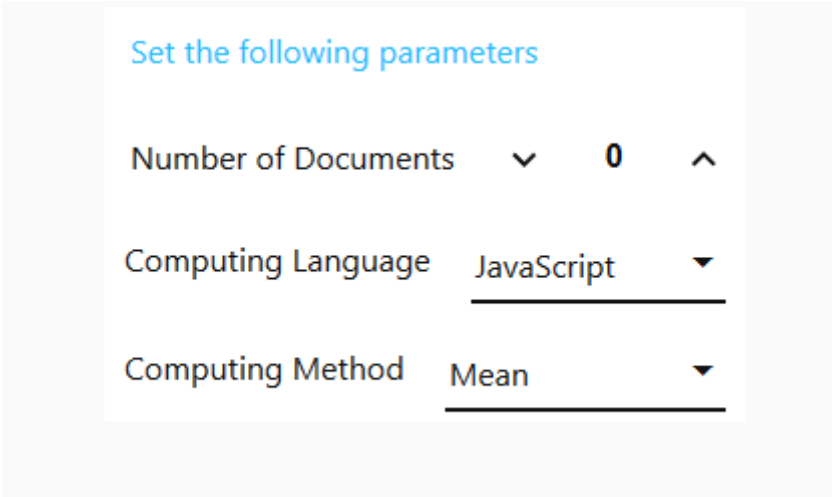
4.5 Ροή UI χρήστη



Εικόνα 4.6 Ροή UI

Η ροή αυτή κάνει χρήση της βιβλιοθήκης του Node-Red, node-red-dashboard, όπου ουσιαστικά δημιουργούμε ένα γραφικό περιβάλλον, για να δώσουμε στο χρήστη τη δυνατότητα να επιλέγει τις παραμέτρους της πλατφόρμας που δημιουργήσαμε.

Εμφανίζουμε, λοιπόν, μία φόρμα, όπου μπορούμε να κάνουμε τρεις επιλογές: αριθμός εγγραφών που θα επιστρέψει το query στην βάση του yelp (αυτόματα αυξάνεται ανά 100), γλώσσα υπολογισμού μετρικών (επιλογή μεταξύ JavaScript, python και Node-Red), και υπολογιστικής μεθόδου (μέσος όρος ή τυπική απόκλιση). Η συνάρτηση set, λαμβάνει την επιλογή του χρήστη, και τη θέτει σε μία global μεταβλητή, που χρησιμοποιούνται από τις υπόλοιπες ροές όπως είδαμε έως τώρα. Το γραφικό περιβάλλον είναι προσβάσιμο στο url: http://<node-red-host>:1880/ui.



Εικόνα 4.7 Φόρμα επιλογής παραμέτρων

5

Συγκριτική Μελέτη

Στο κεφάλαιο αυτό θα παρουσιάσουμε τα αποτελέσματα της μελέτης που πραγματοποιήσαμε. Όπως αναφέραμε σε προηγούμενο κεφάλαιο, έγιναν διαδοχικές εκτελέσεις find queries σε μία βάση mongo, και μετρούσαμε το χρόνο απόκρισης του Node-Red και του Openwhisk, για διαφορετικό αριθμό εγγραφών κάθε φορά. Επιπλέον, υλοποιήσαμε μέθοδο για την εύρεση μέσου χρόνου και της τυπικής απόκλισης των τιμών αυτών. Παράλληλα με αυτά, απεικονίσαμε σε διαγράμματα ένα μέρος των μετρήσεων για επιλεγμένες τιμές του αριθμού των εγγράφων.

5.1 Μετρικές και Διαγράμματα

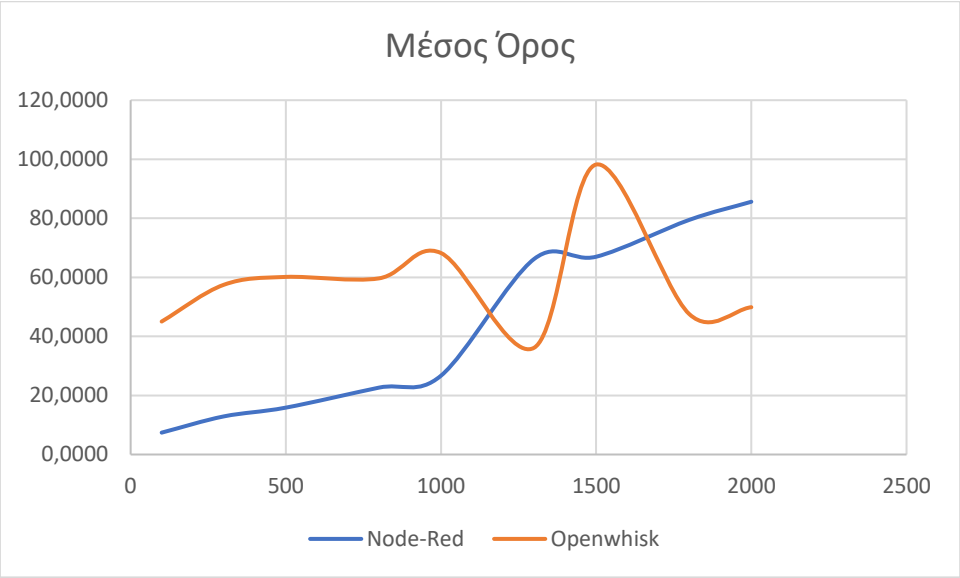
5.1.1 Μετρικές στο σύνολο των μετρήσεων

Επιλέξαμε συγκεκριμένες τιμές του αριθμού των εγγράφων, για τις οποίες πραγματοποιήσαμε τις μετρήσεις μας. Ακολουθεί πίνακας με τους μέσους χρόνους απόκρισης για τις δύο πλατφόρμες για τις επιλεγμένες τιμές αριθμού εγγράφων.

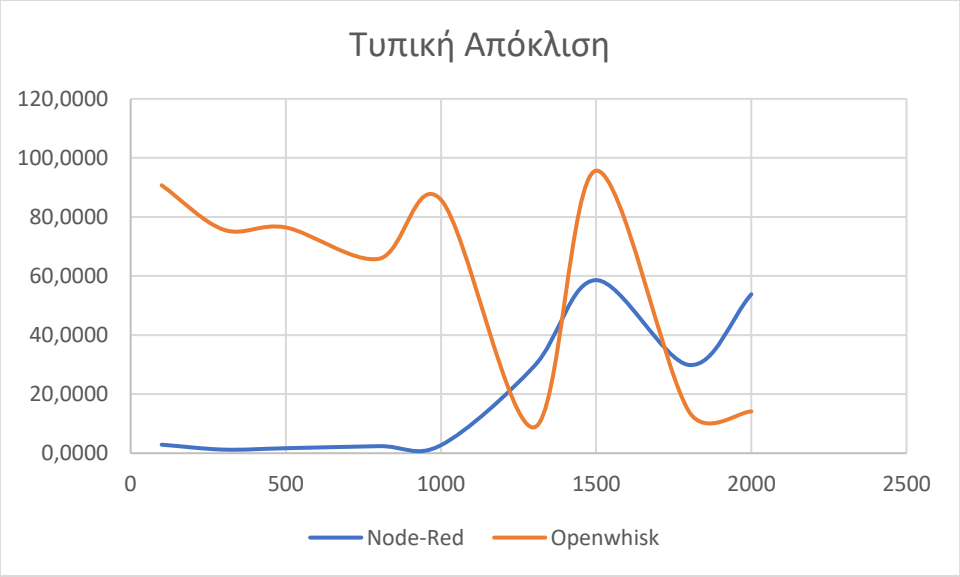
	Μέσος Όρος		**Τυπική Απόκλιση**	
Docs	**Node-Red**	**Openwhisk**	**Node-Red**	**Openwhisk**
100	7,4118	45,0396	2,8954	90,7570
300	12,9000	57,4727	1,2187	75,6958
500	15,8725	60,1485	1,6926	76,4519
800	22,6525	59,6496	2,3979	65,8514
1000	26,7000	68,2314	2,7212	85,8333
1300	66,2188	36,1575	29,4394	8,7563
1500	66,9911	98,2035	58,6121	95,6881
1800	79,4628	47,5667	29,8800	13,9662
2000	85,6078	49,8835	53,8326	14,1430

Πίνακας 5.1 Πίνακας 5.1 Αποτελέσματα μετρικών σε ms

Για την ευκολότερη εξαγωγή συμπερασμάτων από τις μετρικές αυτές παραθέτουμε τα παρακάτω διαγράμματα.



Εικόνα 5.1 Μέσοι χρόνοι απόκρισης σε ms



Εικόνα 5.2 Απόκλιση μετρήσεων

Όπως βλέπουμε στα ανωτέρω διαγράμματα, ο μέσος χρόνος απόκρισης του Nod-Red έχει αύξοντα ρυθμό και αρχικά είναι χαμηλότερος από αυτόν του Openwhisk. Ο μέσος χρόνος του Openwhisk αντίθετα, παραμένει σε μεγάλο βαθμό σταθερός, ενώ σταδιακά μειώνεται.

Όσον αφορά το Node-Red, το παραπάνω είναι απόλυτα λογικά, από τη στιγμή που λειτουργεί με την κλασική μονολιθική αρχιτεκτονική, επομένως όσο αυξάνεται ο όγκος των δεδομένων, αυξάνεται και ο χρόνος εκτέλεσης.

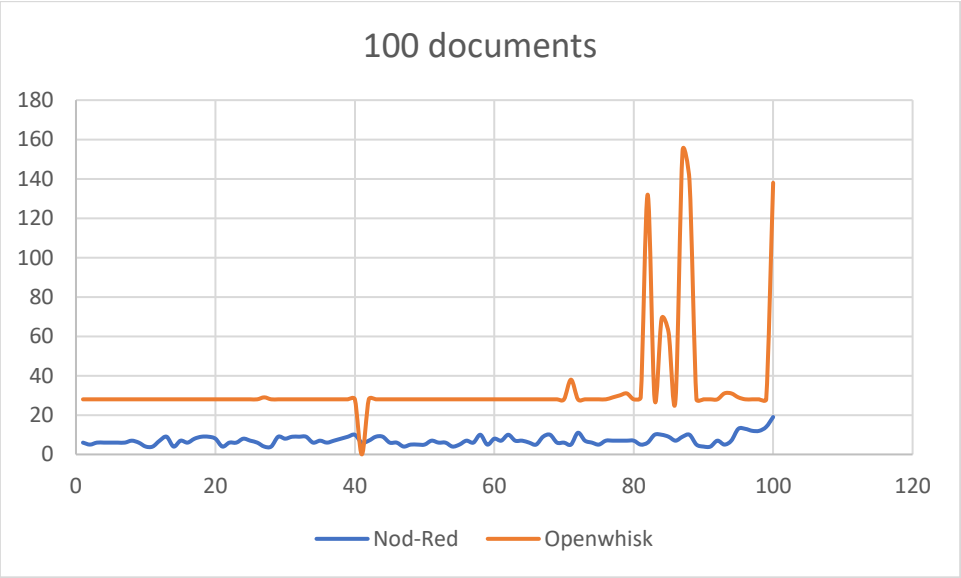
Όσον αφορά το Openwhisk, αρχικά πρέπει να αναφερθούμε στο λόγο που ξεκινά με υψηλότερο χρόνο από ότι το Node-Red. Αυτό συμβαίνει, διότι το Openwhisk λειτουργεί στο cloud, με αποτέλεσμα να απαιτείται απομακρυσμένη επικοινωνία αυτού με τη βάση δεδομένων, καθώς και πολλαπλή μεταφορά των δεδομένων, αφού τα επιστρέφει ως αποτέλεσμα. Αντιθέτως, το Node-Red εκτελείται στο ίδιο περιβάλλον με τη βάση, και επομένως κερδίζει σημαντικό χρόνο. Παρόλα αυτά, παρατηρούμε ότι ο χρόνος στο Openwhisk μειώνεται σταδιακά με την αύξηση του αριθμού των δεδομένων, πράγμα που είναι αποτέλεσμα του scalability που παρέχει το Openwhisk.

Όμοιες παρατηρήσεις μπορούμε να κάνουμε και για την τυπική απόκλιση, ενώ οι όποιες απότομες αυξομειώσεις στις τιμές μας, πιθανώς οφείλονται σε είτε κακή είτε καλή ποιότητα δικτύου τη στιγμή της εκτέλεσης του query.

5.1.2 Διαγράμματα χρόνου εκτέλεσης

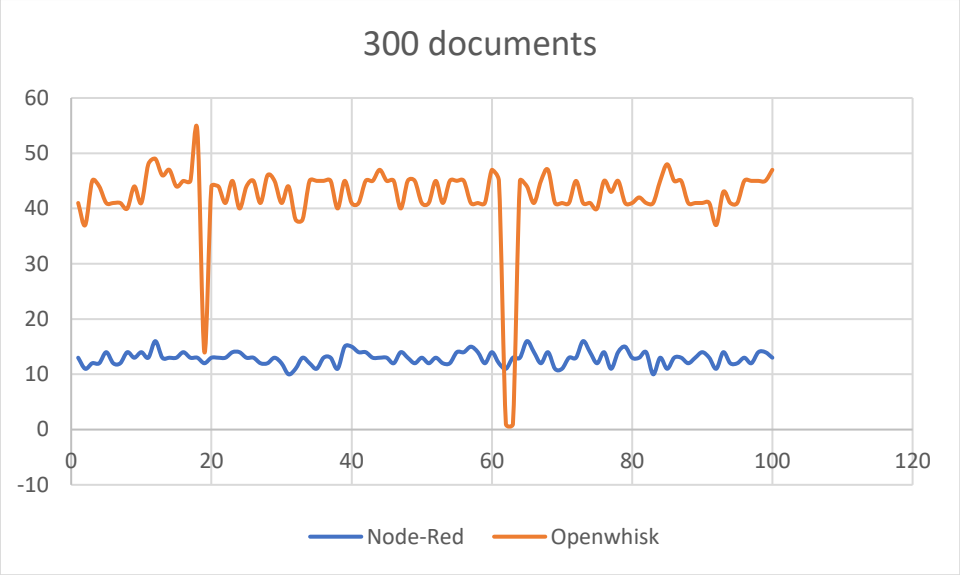
Ακολουθούν τα διαγράμματα του χρόνου εκτέλεσης του query στις δύο πλατφόρμες για τις διάφορες τιμές του αριθμού των εγγραφών στη βάση.

100 Εγγραφές



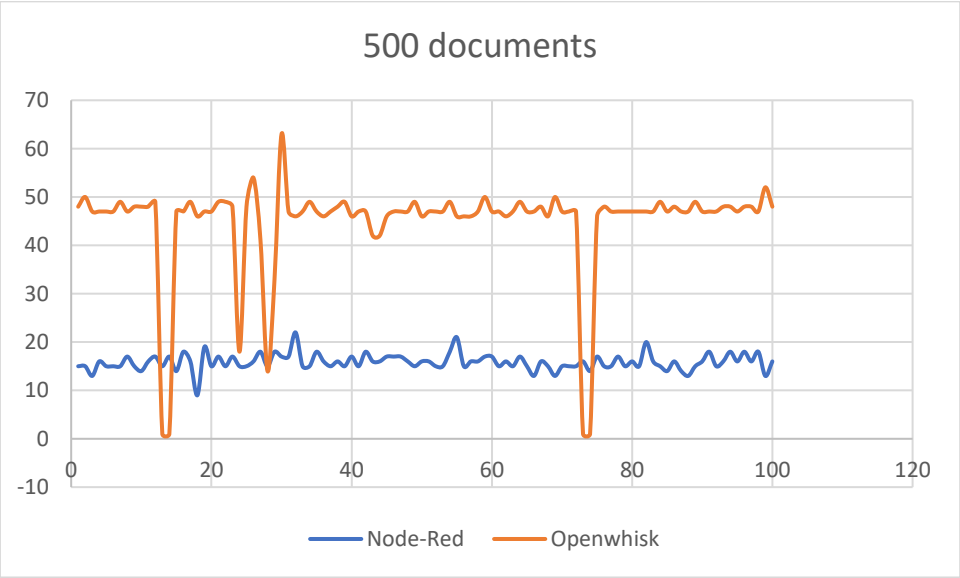
Εικόνα 5.3

300 Εγγραφές



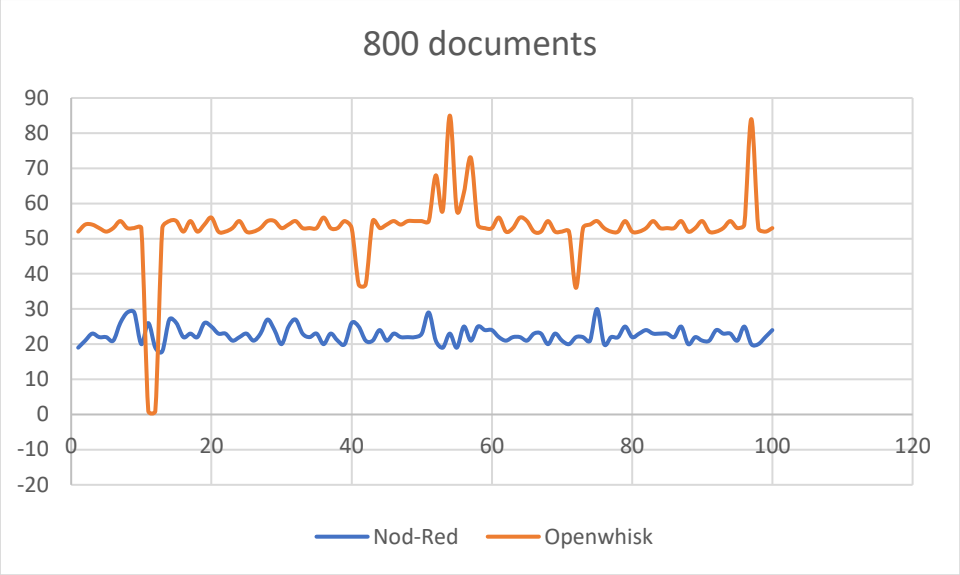
Εικόνα 5.4

500 Εγγραφές



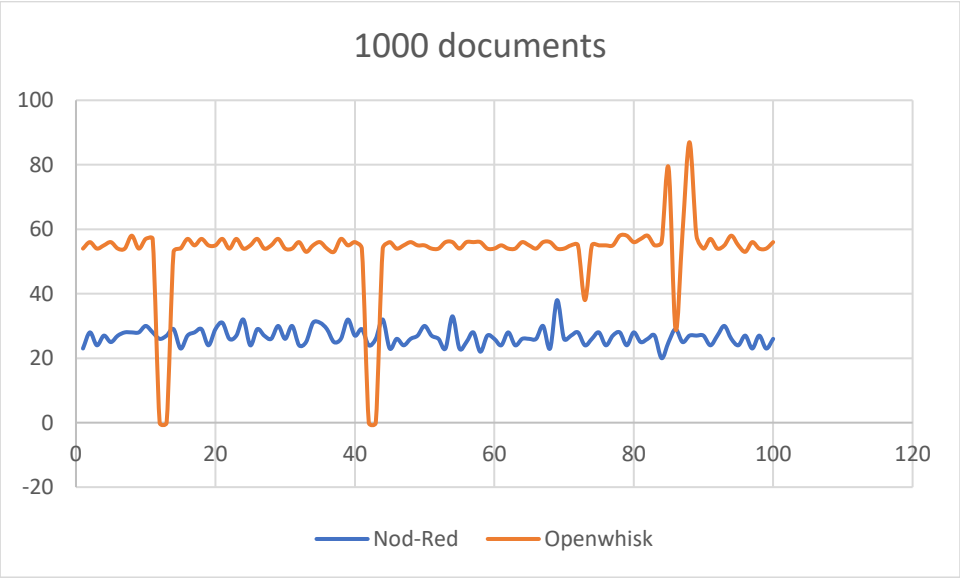
Εικόνα 5.5

800 Εγγραφές



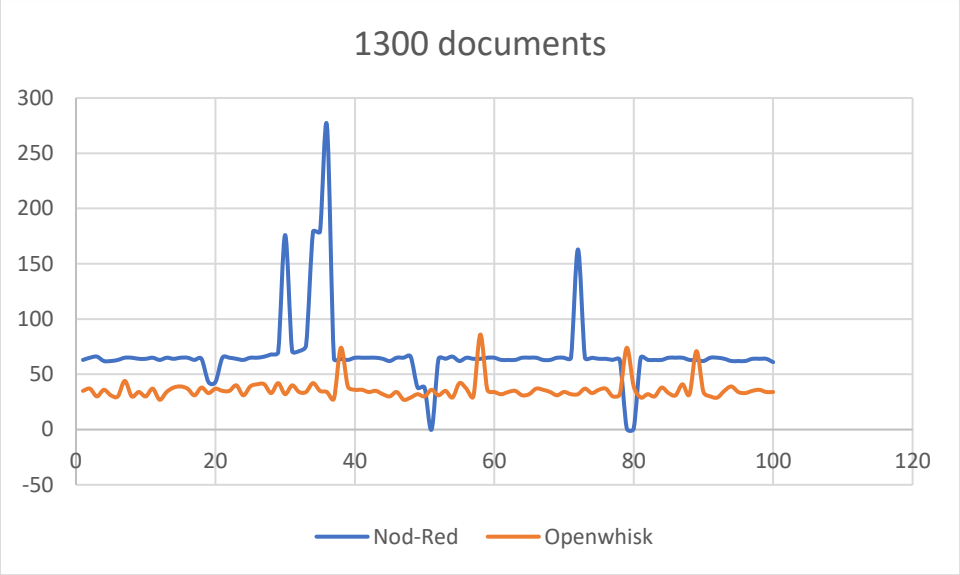
Εικόνα 5.6

1000 Εγγραφές



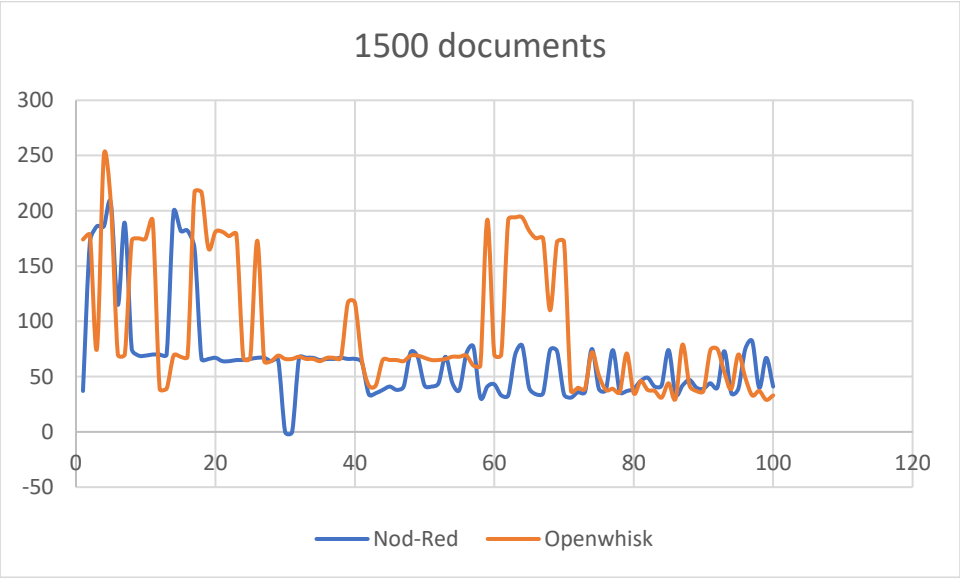
Εικόνα 5.7

1300 Εγγραφές



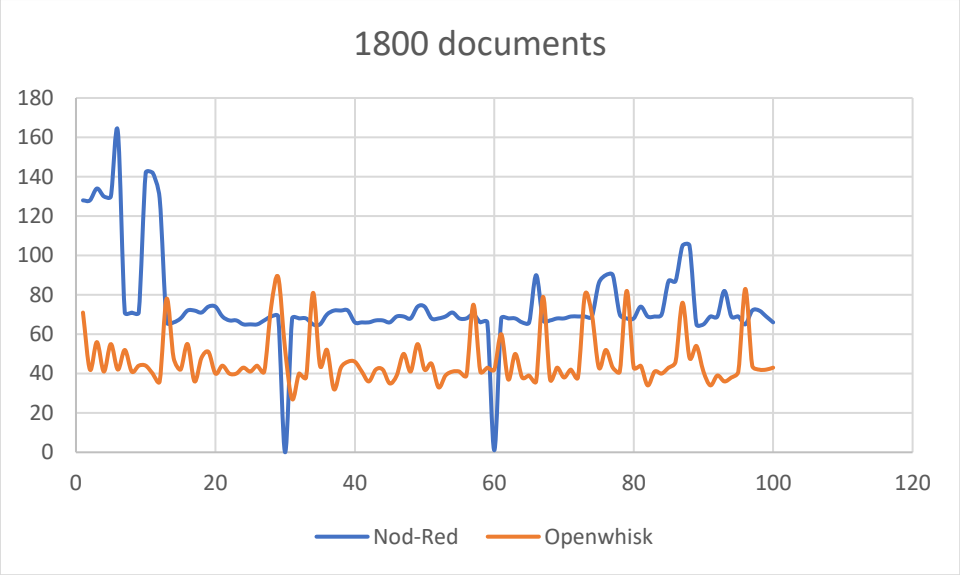
Εικόνα 5.8

1500 Εγγραφές



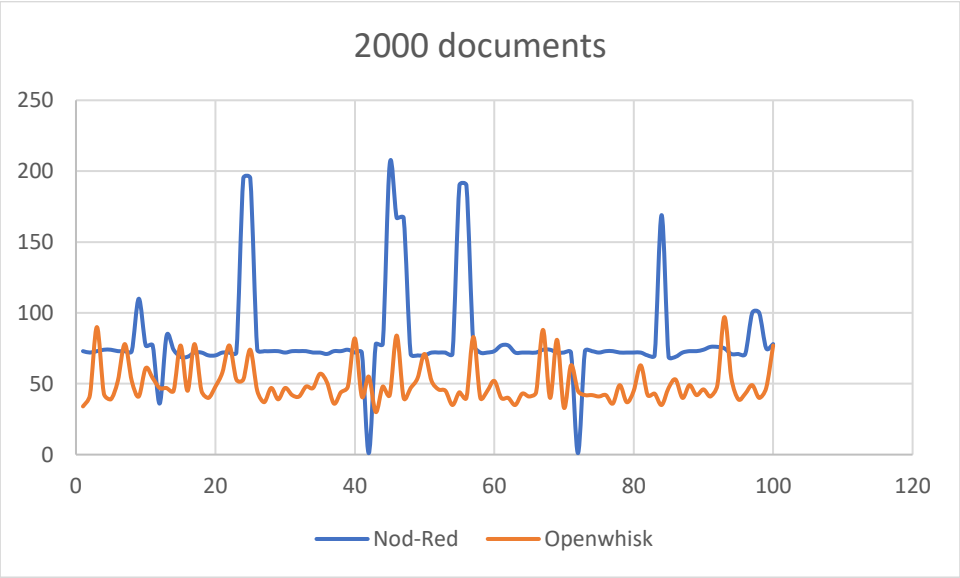
Εικόνα 5.9

1800 Εγγραφές



Εικόνα 5.10

2000 Εγγραφές



Εικόνα 5.11

Με βάση τα ανωτέρω διαγράμματα, έχουμε τις ακόλουθες παρατηρήσεις. Αρχικά, όπως παρατηρήσαμε και από τη μελέτη του μέσου όρου, βλέπουμε υψηλότερο χρόνο απόκρισης από το Openwhisk σε σχέση με το Node-Red. Όσο αυξάνεται, ωστόσο, ο αριθμός των εγγραφών έχουμε μείωση στο χρόνο του Openwhisk, ώσπου φτάνει να είναι καλύτερο από τις 1300 εγγραφές και έπειτα (διάγραμμα 5.8). Επιπλέον, παρατηρούμε μεγαλύτερη απόκλιση στις τιμές του Openwhisk, με κάποιες εξαιρέσεις όπου το Node-Red είχε κι αυτό μεγάλες αποκλίσεις. Αυτό, όπως αναφέραμε, οφείλεται πιθανώς στην ποιότητα του δικτύου κατά την ώρα εκτέλεσης.

6

Επίλογος

Στο κεφάλαιο αυτό θα παρουσιάσουμε τα συμπεράσματα τις μελέτης μας, και θα αναφερθούμε σε μελλοντικές επεκτάσεις και προτάσεις για βελτιστοποίηση.

6.1 Συμπεράσματα

Στην παρούσα διπλωματική εργασία, επιχειρήσαμε να παρουσιάσουμε τα πλεονεκτήματα που μας παρέχουν οι νέες τεχνολογίες του Cloud Computing και FaaS στο IoT. Δημιουργήσαμε μία πλατφόρμα στο Node-Red ώστε να διεξάγουμε την πειραματική μας μελέτη. Μετρήσαμε το χρόνο εκτέλεσης query σε μία βάση Mongo, του Node-Red και του Openwhisk, και πραγματοποιήσαμε στατιστική ανάλυση.

Με βάση την μελέτη που πραγματοποιήσαμε, θα παρουσιάσουμε αναλυτικά τα πλεονεκτήματα που το Openwhisk μας παρείχε. Αρχικά, παρατηρήσαμε τις δυνατότητες για γρήγορα αποτελέσματα που έχει το Openwhisk, αφού μας παρέχει μεγάλο scalability. Όπως είδαμε, όταν η πλατφόρμα ήρθε αντιμέτωπη με μεγάλο όγκο δεδομένων, το Openwhisk έδινε καλύτερα αποτελέσματα.

Επιπλέον, κάτι το οποίο δεν παρατηρήσαμε με τη στατιστική μελέτη, βασίζεται στη φύση λειτουργίας του Openwhisk. Όπως έχουμε αναφέρει, βασικό χαρακτηριστικό της τεχνολογίας του cloud computing, είναι η μη κατανάλωση πόρων του συστήματος σε περιόδους που η υπηρεσία που υλοποιεί δεν εκτελείται. Με αυτόν τον τρόπο, έχουμε επιπλέον εξοικονόμηση σε υπολογιστικούς πόρους, καθώς και μικρότερη επιβάρυνση του συστήματος, εξασφαλίζοντας έτσι μεγαλύτερη αποδοτικότητα.

Το Openwhisk, ακόμη, κερδίζει στο γλωσσικό παράγοντα, αφού μας επιτρέπει τη συγγραφή σε διάφορες γλώσσες, ξεφεύγοντας από τον περιορισμό του Node-Red στο JavaScript. Όπως είδαμε και στην πλατφόρμα που παρουσιάσαμε, υπήρχε η δυνατότητα για εκτέλεση Openwhisk

action και σε python, ενώ γενικά μπορούμε να φτιάξουμε actions σε Java, Scala, Swift, Go, PHP και Ruby. Με αυτόν τον τρόπο, μας δίνεται η δυνατότητα να εκτελέσουμε διαδικασίες που πιθανώς να μην μπορούσαμε -είτε καθόλου, είτε με την ίδια ευκολία- στο Node-Red.

Συμπερασματικά, βλέπουμε πως η τεχνολογία του cloud computing, έρχεται να αλλάξει σημαντικά τα σημερινά δεδομένα στην ανάπτυξη εφαρμογών. Η αρχιτεκτονική των microservices, σε συνδυασμό με πλατφόρμες όπως το Openwhisk, παρουσιάζουν μία νέα, πιο εξελιγμένη εποχή στον προγραμματισμό. Μία μεγάλη εφαρμογή μπορεί να υλοποιείται τμηματικά, όπου κάθε τμήμα της θα είναι ένα microservice, και όλα θα υπάρχουν στο cloud. Οι δυνατότητες που διανοίγονται με αυτό τον τρόπο είναι μεγάλες, από τη στιγμή που μπορούμε να εκμεταλλευτούμε όλα τα πλεονεκτήματα αυτών των τεχνολογιών, τα οποία, σε μία εφαρμογή μεγάλης κλίμακας, είναι ζωτικής σημασίας.

Παρόλα αυτά, κατά την πραγματοποίηση της μελέτης μας, παρατηρήσαμε και κάποια προβλήματα και τυχόν μειονεκτήματα. Το βασικότερο όλων, αφορά στην ποιότητα του δικτύου, γεγονός μείζονος σημασίας, αφού η λογική του cloud computing απαιτεί δικτυακή επικοινωνία με άλλα μηχανήματα. Ουσιαστικά, ενώ μπορεί το Openwhisk να μας παρέχει πληθώρα δυνατοτήτων, μία κακή ποιότητα δικτύου μπορεί να οδηγήσει σε αντίθετα αποτελέσματα, από τη στιγμή που πολύτιμος χρόνος σπαταλάται για την επικοινωνία του μηχανήματος μας με το cloud. Στην ανάλυση που πραγματοποιήσαμε, είδαμε ότι αρχικά το Openwhisk απαιτούσε περισσότερο χρόνο εκτέλεσης των queries, ακριβώς για τον παραπάνω λόγο.

Συνεπώς, αυτό που μπορούμε να σχολιάσουμε, είναι ότι μία καλή ποιότητα δικτύου είναι απαραίτητη ώστε να έχουμε πλήρη αξιοποίηση των πλεονεκτημάτων των νέων τεχνολογιών. Επιπλέον, ο προγραμματιστής με τη σειρά του, οφείλει να μπορεί να εντοπίσει τη βέλτιστη λύση, τον τρόπο δηλαδή, που η χρήση της νέας τεχνολογίας δίνει καλύτερα αποτελέσματα. Ωστόσο, όπως καθετί καινούργιο, απαιτείται εξοικείωση και πειραματισμός, ώστε να μπορούμε να εκμεταλλευτούμε αυτά που μας παρέχει με βέλτιστο τρόπο.

6.2 Μελλοντικές επεκτάσεις

Στη μελέτη που πραγματοποιήσαμε, μπορούν να πραγματοποιηθούν διάφορες βελτιώσεις. Αρχικά, θα μπορούσε να γίνει αλλαγή στο αντικείμενο από το οποίο κάνουμε χρονικές μετρήσεις. Ένα query σε μεγάλη βάση δεδομένων ήταν ενδεικτικό για τη μελέτη μας, ωστόσο, θα μπορούσε να γίνει χρήση άλλης δύσκολης διεργασίας και μέτρηση του χρόνου αυτής, η οποία θα απέφευγε το μειονέκτημα της απαίτησης για μεταφορά μεγάλου όγκου δεδομένων.

Επιπρόσθετα, θα μπορούσε να γίνει μεγαλύτερη επίδειξη των πλεονεκτημάτων της event driven αρχιτεκτονικής του Openwhisk, η οποία στην παρούσα μελέτη παρουσιάστηκε σε μικρό βαθμό.

Τέλος, ενδιαφέρον θα είχε η απόπειρα για πλήρη υλοποίηση της εφαρμογής στο cloud με τη μορφή του IaaS, όπου ο προγραμματιστής δε θα είχε άμεση επαφή με το σύστημα που εκτελεί την εφαρμογή, αφού ακόμα και το hardware βρίσκεται στο cloud.

Παράρτημα – Κώδικας Openwhisk

A. Action query

```
function main(params){
var MongoClient = require('mongodb').MongoClient
var url = 'mongodb://147.102.19.1:27017/'
return new Promise(function(resolve, reject) {
   MongoClient.connect(url, (err, db) => {

db.db('yelp').collection('Review').find({stars:5}).limit(param
s.lim).toArray().then((docs) => {
           resolve({result : docs});
           db.close();
       }).catch((err) => {
          // console.log(err.stack);
           reject(err)
       });
   })
})
}
```

B. Action stats1

```
function avg(params){
    var s=0;
    var count=Object.keys(params).length;
    for(var key in params) {
      if (params.hasOwnProperty(key)) {
        s += parseInt(params[key]);
```

```
      }
    }
    return s/count;
}

function ta(params,avg){
    var s=0;
    var count=Object.keys(params).length;
    s=0;
    for(var key in params){
      if (params.hasOwnProperty(key)){
        s += Math.pow(parseInt(params[key])-avg,2);
      }
    }
    return Math.sqrt(s/(count-1));
}

function bad(params,avg,ta){
    var s=0;
    var count=Object.keys(params).length;
    for(var key in params) {
      if (params.hasOwnProperty(key)) {
        if(parseInt(params[key])>avg+ta) s++;
      }
    }
    return 100-100*(s/count);
}


function main(params){
    var a=avg(params);
    var b=ta(params,a);
```

```
    var c=bad(params,a,b);
    return {payload : {"mean1" : a, "sd1" : b, "good1" : c}};
}
```

C. Action pyStats1

```
from __future__ import division
def main(args):
        import json
        arr=json.dumps(args)
        a=json.loads(arr)

        s=0;
        for x in a['params']:
                s+=x['Time']
        mean1=s/len(a['params'])

        s1=0
        for x in a['params']:
                s1+=(x['Time']-mean1)**2
        import math
        sd1=math.sqrt(s1/(len(a['params'])-1))

        bad1=0
        for x in a['params']:
                if x['Time']>(mean1+sd1):
                        bad1+=1
        good1=100-100*bad1/len(a['params'])

        return
{"payload":{"mean1":mean1,"sd1":sd1,"good1":good1}}
```

D. Action stats2

```
function avg(params){
    var s=0;
    var count=Object.keys(params).length;
    for(var key in params) {
      if (params.hasOwnProperty(key)) {
        s += parseInt(params[key]);
      }
    }
    return s/count;
}

function ta(params,avg){
    var s=0;
    var count=Object.keys(params).length;
    s=0;
    for(var key in params){
      if (params.hasOwnProperty(key)){
        s += Math.pow(parseInt(params[key])-avg,2);
      }
    }
    return Math.sqrt(s/(count-1));
}

function bad(params,avg,ta){
    var s=0;
    var count=Object.keys(params).length;
    for(var key in params) {
      if (params.hasOwnProperty(key)) {
        if(parseInt(params[key])>avg+ta) s++;
      }
```

```
    }
    return 100-100*(s/count);
}


function main(params){
    var a=avg(params);
    var b=ta(params,a);
    var c=bad(params,a,b);
    return {payload : {"mean2" : a, "sd2" : b,"good2" : c}};
}
```

E. Action pyStats2

```
from __future__ import division
def main(args):
        import json
        arr=json.dumps(args)
        a=json.loads(arr)

        s=0;
        for x in a['params']:
                s+=x['Time']
        mean2=s/len(a['params'])

        s1=0
        for x in a['params']:
                s1+=(x['Time']-mean2)**2
        import math
        sd2=math.sqrt(s1/(len(a['params'])-1))

        bad2=0
```

```
        for x in a['params']:
                if x['Time']>(mean2+sd2):
                        bad2+=1
        good2=100-100*bad2/len(a['params'])

        return
{"payload":{"mean2":mean2,"sd2":sd2,"good2":good2}}
```

F. Action choice1

```
var request = require('request');

function main(params){
    var choice = 1;
    request('http://147.102.19.1:1880/switch',    function(err,
res, body) {});
    return({'choice':choice});
}
```

E. Action choice2

```
var request = require('request');

function main(params){
    var choice = 2;
    request('http://147.102.19.1:1880/switch',    function(err,
res, body) {});
    return({'choice':choice});
}
```

Βιβλιογραφία

[1] TechRepublic. (2017). *How IBM's Node-RED is hacking together the internet of things*. [online] Available at: http://www.techrepublic.com/article/node-red/

[2] IBM OpenTech. (2017). *What makes serverless architectures so attractive? - IBM OpenTech*. [online] Available at: https://developer.ibm.com/opentech/2016/09/06/what-makes-serverless-attractive/

[3] Kiran , Oliver. "What Makes IBM OpenWhisk Different? Openness." *What Makes IBM OpenWhisk Different? Openness*, Thenewstack, 17 Oct. 2016, thenewstack.io/exploring-pros-cons-serverless-computing-ibm-openwhisk/.

[4] Interview: Andreas Nauerz Of Openwhisk/bluemix Ryan Brown - https://serverlesscode.com/post/interview-andreas-nauerz-bluemix-openwhisk/

[5] What Is Internet Of Things (iot)? - Definition from Whatis.com https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

[6] Bowei, Ryab. "An Introduction to Serverless and FaaS (Functions as a Service)." *An Introduction to Serverless and FaaS (Functions as a Service)*, Medium, 5 Nov. 2017, medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2.

[7] Aws Lambda – Product Features https://aws.amazon.com/lambda/features/

[8] Netflix Conductor: A Microservices Orchestrator – Netflix Techblog – Medium

Netflix Technology Blog - https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40

[9] Gulati, Vinitesh. "Serverless Architecture and IBM OpenWhisk." *Prolifics*, Prolifics, www.prolifics.com/blog/serverless-architecture-and-ibm-openwhisk.

[10] Cui, Yan. "Auto-Scaling Kinesis Streams with AWS Lambda." *Auto-Scaling Kinesis Streams with AWS Lambda*, A Cloud Guru, 5 May 2017, read.acloud.guru/auto-scaling-kinesis-streams-with-aws-lambda-299f9a0512da.

[11] Heidloff, Niklas. "What Is Node-RED? How Can It Be Used for the Internet of Things?" *What Is Node-RED? How Can It Be Used for the*

Internet of Things?, Niklas Heidloff, 2 Feb. 2015, heidloff.net/article/21.01.2015081841NHEAL8.htm.

[12] Apache/incubator-openwhisk Apache - https://github.com/apache/incubator-openwhisk/blob/master/docs/about.md

[13] Internet Of Things (iot)

https://www.sas.com/el_gr/insights/big-data/internet-of-things.html