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

Εξόρυξη, Επεξεργασία, Αποθήκευση και Οπτικοποίηση δεδομένων από αισθητήρες σε πραγματικό χρόνο (Live)

Διπλωματικη Εργασια

του

ΣΤΕΦΑΝΟΥ Γ. ΣΟΥΛΙΩΤΗ

Επιβλέπων: Βασιλική Καντερέ
Επίκουρη Καθηγήτρια Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Εξόρυξη, Επεξεργασία, Αποθήκευση και Οπτικοποίηση δεδομένων από αισθητήρες σε πραγματικό χρόνο (Live)

Διπλωματικη Εργασια

του

ΣΤΕΦΑΝΟΥ Γ. ΣΟΥΛΙΩΤΗ

Επιβλέπων: Βασιλική Καντερέ
Επ. Καθηγήτρια Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 3η Μαρτίου 2021

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Βασιλική Καντερέ	Νεκτάριος Κοζύρης	Συμεών Παπαβασιλείου
Επ. Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

(Υπογραφή)

.....

Στεφανος Γ. Σουλιωτης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η εξόρυξη, η επεξεργασία, η αποθήκευση και η οπτικοποίηση δεδομένων από αισθητήρες σε πραγματικό χρόνο, μέσω μιας κοινής αρχιτεκτονικής, αποτελεί μια όλο και μεγαλύτερη ανάγκη παγκοσμίως, όσο αυξάνονται τα αυτοδιαχειριζόμενα συστήματα. Η καταγραφή και συλλογή των δεδομένων από κτιριακές εγκαταστάσεις, όπως η θερμοκρασία τους, η κατανάλωση ενέργειας τους, η παραγωγή ενέργειάς τους αλλά και άλλα χαρακτηριστικά τους οποιασδήποτε φύσης επιτυγχάνεται μέσω εφαρμογών του IoT (Internet of Things). Διασυνδεδεμένες συσκευές και αισθητήρες συλλέγουν μεγάλο όγκο δεδομένων σε πραγματικό χρόνο, τα οποία παράλληλα αποστέλλουν με ταχεία και ελαφριά πρωτόκολλα MQTT σε σύστημα ανταλλαγής μηνυμάτων το οποίο βασίζεται στην τεχνική publish-subscribe. Ένα τέτοιο συστήματα είναι το Apache Kafka το οποίο εγγυάται μικρές καθυστερήσεις και ελάχιστες απώλειες ενώ παράλληλα διαχειρίζεται μεγάλο όγκο δεδομένων. Στη συνέχεια, τα δεδομένα υποβάλλονται σε παράλληλη και real time επεξεργασία και ανάλυση μέσω κατάλληλων frameworks, όπως το Apache Spark Streaming, το Kafka Streams και το Apache Flink. Μετά την επεξεργασία τους ο μεγάλος και γρήγορα ανανεούμενος όγκος δεδομένων οπτικοποιείται σε ένα πρόγραμμα παρατήρησης και αποθηκεύεται σε βάσεις δεδομένων με τρόπο που να είναι ευκολότερη και πιο γρήγορη η αναζήτησή τους. Με αυτόν τον τρόπο τα άλλοτε πολλά, μεμονωμένα και ασύνδετα δεδομένα αποκτούν μορφή η οποία είναι ποιοτικά ανώτερη λόγω των μετατροπών που υφίσταται από τις εφαρμογές ανάλυσης και επεξεργασίας. Τέλος, αξιολογείται το σύστημα όσον αφορά τις εγγυήσεις για μικρές καθυστερήσεις και ελάχιστες απώλειες στην ροή της πληροφορίας, καθώς και την δυνατότητα κλιμακωσιμότητάς του.

Λέξεις κλειδιά:
Internet of Things, Cloud Computing, Edge Computing, MQTT, Apache Kafka, Apache Spark, Spark Streaming, Kafka Streams, Apache Flink, Flask-SocketIO

Abstract

Mining, processing, storing and visualizing data from sensors in real time, through a common architecture, is a growing need worldwide as self-managed systems grow. The recording and collection of data from building installations, such as their temperature, energy consumption, energy production and other characteristics of any nature is achieved through IoT (Internet of Things) applications. Interconnected devices and sensors collect large volumes of data in real time, which at the same time send with fast and light MQTT protocols to a message exchange system based on the publish-subscribe technique. One such system is Apache Kafka which guarantees small delays and minimal losses while managing a large amount of data. The data is then processed and analyzed in real time via appropriate frameworks, such as Apache Spark Streaming, Kafka Streams and Apache Flink. After processing, the large and quickly updated volume of data is visualized in an observer program and stored in databases in a way that makes them easier and faster to search. In this way the once large, individual and unconnected data acquires a form which is qualitatively superior due to the transformations that exist from the analysis and processing applications. Finally, the system is evaluated in terms of guarantees for small delays and minimal losses in the flow of information, as well as its scalability.

Keywords:
Internet of Things, Cloud Computing, Edge Computing, MQTT, Apache Kafka, Apache Spark, Spark Streaming, Kafka Streams, Apache Flink, Flask-SocketIO

Ευχαριστίες

Με την περάτωση της διπλωματικής μου εργασίας, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες σε όλους όσους συνέβαλλαν στην εκπόνησή της. Ιδιαίτερα, θα ήθελα να ευχαριστήσω την καθηγήτρια κ. **Βασιλική Καντερέ**, για την εμπιστοσύνη που μου έδειξε, δίνοντάς μου την ευκαιρία να μελετήσω ένα τόσο ενδιαφέρον θέμα, για την πολύτιμη καθοδήγηση και την επίβλεψή της. Θερμές ευχαριστίες θα ήθελα να απευθύνω και στον υποψήφιο διδάκτορα **Παρασκευά Κερασιώτη**, ο οποίος με τις πολύτιμες συμβουλές του συνέβαλε στην ολοκλήρωση της παρούσας διπλωματικής εργασίας. Τέλος, θα ήθελα να ευχαριστήσω βαθύτατα την οικογένειά μου, καθώς και τους φίλους μου, που με στήριξαν καθ' όλη τη διάρκεια των σπουδών μου.

Περιεχόμενα

Κατάλογος Σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός της εργασίας

Σκοπός της εργασίας είναι η ανάπτυξη μιας αρχιτεκτονικής για την εξόρυξη, επεξεργασία, αποθήκευση και την οπτικοποίηση (visualization) δεδομένων από αισθητήρες σε πραγματικό χρόνο.

Συγκεκριμένα μελετάται η πορεία δεδομένων και μετρήσεων πραγματικού χρόνου, από την συλλογή τους από αισθητήρες μέσω εφαρμογών IoT (Internet of Things), την διάθεση τους μέσω πρωτοκόλλων ανταλλαγής μηνυμάτων, την παράλληλη επεξεργασία και μετατροπή τους από εφαρμογές, με σκοπό να έχουμε όσο το δυνατόν μικρότερες απώλειες και μικρότερες καθυστερήσεις. Αυτή είναι περιληπτικά και η πορεία που ακολουθεί μεγάλος όγκος πραγματικών δεδομένων, με τελικό στόχο τον έλεγχο και την διαχείριση κτιριακών εγκαταστάσεων σε πραγματικό χρόνο.

Η μελέτη αυτή ξεκινά με την περιγραφή της εξέλιξης του IoT, της αρχιτεκτονικής και των πρωτοκόλλων που χρησιμοποιούνται, καθώς και των εφαρμογών του IoT που χρησιμοποιούνται στις μέρες μας. Στη συνέχεια, μελετώνται ευέλικτα εργαλεία και πρωτόκολλα ανταλλαγής μηνυμάτων όπως το MQTT καθώς και συστήματα ενδιάμεσης αποθήκευσης όπως το Apache Kafka. Έμφαση δίνεται στα εργαλεία επεξεργασίας και μετατροπής των δεδομένων πραγματικού χρόνου όπου ελέγχθηκαν τα παραδείγματα του Apache Spark, του Kafka Streams και του Apache Flink. Τέλος, η αρχιτεκτονική που κατασκευάστηκε δοκιμάστηκε σε συνθήκες πραγματικού περιβαλλοντος και αξιολογήθηκε βάση της απόδοσης της.

1.2 Γενικό πλαίσιο

Για την επεξεργασία και οπτικοποίηση δεδομένων από αισθητήρες πρώτο βήμα είναι να βρεθεί ένας τρόπος αποδοτικής μεταφοράς και αποθήκευσης των δεδομένων. Τα δεδομένα μας εξορύσσονται μέσω IoT και αισθητήρων σε πραγματικό χρόνο, συνεπώς η συλλογή των πληροφοριών και η μεταφορά τους πρέπει να γίνεται με τρόπο ταχύ και αξιόπιστο έτσι ώστε να έχουμε μικρές καθυστερήσεις και ελάχιστες απώλειες. Επίσης, είναι απαραίτητο να επιλεχθεί ένας τρόπος μεταφοράς ο οποίος να μην χρειάζεται μεγάλα αποθέματα μνήμης και μεγάλη

υπολογιστική ισχύ, καθώς έχουμε να κάνουμε με αισθητήρες οι οποίοι έχουν μικρό μέγεθος και περιορισμένη υπολογιστική δυνατότητα. Ακόμα, για το κομμάτι της ενδιάμεσης αποθήκευσης χρειαζόμαστε ένα σύστημα το οποίο να είναι γρήγορο, αξιόπιστο και να μπορεί να εξυπηρετήσει χιλιάδες δεδομένα σε μικρό χρονικό διάστημα ενώ παράλληλα είναι και εξαιρετικά κλιμακώσιμο σε περίπτωση που χρειαστεί να προστεθούν νέοι αισθητήρες στο σύστημα. Για τις παραπάνω προδιαγραφές μελετώνται πρωτόκολλα MQTT για την μεταφορά από τους αισθητήρες και publish/subscribe μοντέλα, όπως ο Kafka Server, για την ενδιάμεση αποθήκευση.

Αφού εξασφαλιστεί η μεταφορά και η ενδιάμεση αποθήκευση των δεδομένων περνάμε στην επεξεργασία τους. Εφόσον αντιμετωπίζουμε ένα σενάριο το οποίο εκτυλίσσεται σε πραγματικό χρόνο η χρονική εξάρτηση και χρονική σειρά είναι απαραίτητο να είναι αυστηρά ορισμένη. Γι αυτό το λόγο, πρέπει προχωρώντας στην επεξεργασία να διορθώνουμε τυχόν λάθη τα οποία προέκυψαν στο κομμάτι της μεταφοράς χρησιμοποιώντας χρονοσφραγίδες και απορίπτοντας δεδομένα εκτός σειράς. Η φύση του προβλήματος εγείρει την απαραίτητη προϋπόθεση της, όσο το δυνατόν, ταχύτερης επεξεργασίας και προώθησης των αποτελεσμάτων στην έξοδο, διότι όπως αναφέραμε η είσοδος αποτελείται από μια αδιάκοπη και ταχεία ροή πληροφοριών γεγονός που μπορεί να προκαλέσει συσσώρευση δεδομένων και κατάρρευση του συστήματος σε περιπτώσεις μεγάλων καθυστερήσεων. Για το σκοπό αυτό, μελετήθηκαν παραδείγματα streaming εφαρμογών όπως το Apache Spark Streaming, το Kafka Streams και το Apache Flink, τα οποία παρέχουν εγγυήσεις στο κομμάτι της ταχείας επεξεργασίας και της υψηλής διαμεταγωγής (throughput) των μηνυμάτων.

Τέλος, θέλουμε να παρουσιάσουμε τις επεξεργασμένες μορφές των δεδομένων μας και να τις αποθηκεύσουμε, είτε για να κρατήσουμε ιστορικό είτε για να μπορέσουμε να εκτελέσουμε ερωτήματα (queries) σε αυτές και να παρατηρήσουμε συγκεκριμένες εξαρτήσεις. Όπως και σε όλα τα άλλα κομμάτια του συστήματός μας, πρέπει να βασιστούμε σε εργαλεία με μεγάλη ταχύτητα και δυνατότητα διαχείρισης μεγάλου όγκου πληροφορίας. Στο κομμάτι της αποθήκευσης μπορούμε να είμαστε περισσότερο επιεικείς στο θέμα της ταχύτητας οπότε δοκιμάστηκαν πολλές διαφορετικές βάσεις δεδομένων όπως in-memory databases, NoSQL καθώς και κλασικές SQL βάσεις. Αντίθετα, στο κομμάτι της οπτικοποίησης θέλουμε να έχουμε όσο το δυνατόν μικρότερες καθυστερήσεις και να παρουσιάζεται η τρέχουσα κατάσταση του συστήματος. Για αυτό το κομμάτι μελετήθηκε client/server επικοινωνία μέσω WebSocket με το εργαλείο flask.

1.3 Δήλωση του προβλήματος

Σε όλα τα συστήματα που επεξεργάζονται δεδομένα σε πραγματικό χρόνο αντιμετωπίζουμε τρία βασικά προβλήματα που καλούμαστε να λύσουμε. Αρχικά, πρέπει το σύστημά μας να μπορεί να διαχειριστεί μεγάλο όγκο δεδομένων καθώς και να είναι γρήγορο, να παρουσιάζει δηλαδή μικρές καθυστερήσεις, έτσι ώστε τα αποτελέσματα που παίρνουμε στην έξοδο να αποτελούν την παρούσα κατάσταση του συστήματος και να παίρνουμε μετρήσεις και αποτελέσματα που αφορούν το παρελθόν. Στη συνέχεια, είναι ζωτικής σημασίας να μπορούμε να εγγυηθούμε ότι η χρονική σειρά που έρχονται τα δεδομένα είναι σωστή και δεν λαμβάνουμε

δεδομένα σε λάθος σειρά, αφού αυτό μπορεί να επηρεάσει τα αποτελέσματα και να παρουσιάσει μια λανθασμένη εικόνα για το σύστημά μας. Τέλος, όπως και σε όλα τα συστήματα θέλουμε τα δεδομένα μας, τόσο αυτά που λαμβάνουμε στην είσοδο ως μετρήσεις από αισθητήρες, όσο και τα δεδομένα που παράγονται από τις μετατροπές του συστήματος, να είναι διασφαλισμένα ώστε να μην χαθούν σε περίπτωση που κάποια συνιστώσα του συστήματος μας αποτύχει.

1.4 Αντιμετώπιση του προβλήματος

Στην συγκεκριμένη εργασία θα παρουσιάσουμε μια αρχιτεκτονική η οποία αντιμετωπίζει τα παραπάνω ζητήματα και υλοποιεί ένα ολοκληρωμένο σύστημα εξόρυξη, επεξεργασία, αποθήκευση και την οπτικοποίηση δεδομένων από αισθητήρες σε πραγματικό χρόνο.

Για το κομμάτι της ταχύτητας και του όγκου δεδομένων χρησιμοποιήθηκαν πρωτόκολλα και εργαλεία τα οποία διακρίνονται για την ταχύτητά τους και την αντοχή τους σε μεγάλο όγκο δεδομένων. Για παράδειγμα, για την μεταφορά των δεδομένων των αισθητήρων χρησιμοποιήθηκε το πρωτόκολλο MQTT το οποίο είναι εξαιρετικά ταχύ. Επίσης, αντί για παραδοσιακές βάσεις δεδομένων για την ενδιάμεση αποθήκευση χρησιμοποιήσαμε Apache Kafka εργαλείο τύπου publish/subscribe το οποίο διαχειρίζεται μεγάλο όγκο δεδομένων με μεγάλη ταχύτητα. Ακόμα, στο κομμάτι της επεξεργασίας χρησιμοποιήθηκαν εργαλεία για επεξεργασία ροής δεδομένων (streams) όπως το Apache Spark, το Kafka Streams και το Apache Flink τα οποία διακρίνονται για την ταχύτητα και την δυνατότητά τους να αντιμετωπίζουν μεγάλο όγκο δεδομένων.

Για το κομμάτι των εγγυήσεων της σωστής χρονικής σειράς, τα εργαλεία επεξεργασίας ροής δεδομένων που χρησιμοποιήσαμε παρέχουν ειδικές συναρτήσεις για φιλτράρισμα των εκτός σειράς δεδομένων την απόρριψή τους και την αποθήκευσή τους σε ξεχωριστό σημείο. Τέλος, σε όλες τις εξόδους των επιμέρους δομικών στοιχείων που αποτελούν την αρχιτεκτονική διασφαλίζουμε τα δεδομένα αποθηκεύοντας σε βάσεις δεδομένων ώστε να μην χάνονται σε περίπτωση αποτυχίας και για να μπορεί μετά από αποτυχία του συστήματος ανάνηψη του συστήματος.

1.5 Δομή της εργασίας

Στο δεύτερο κεφάλαιο ορίζεται και περιγράφεται η έννοια του IoT (Internet of Things), πώς αυτό λειτουργεί και γιατί είναι σημαντικό και θα αποτελέσει τεχνολογία που θα ανθίσει στο μέλλον.

Στο επόμενο κεφάλαιο, αναλύονται η έννοιες του cloud computing και του edge computing, ο τρόπος λειτουργίας τους, τα πλεονεκτήματά τους και οι τεχνολογίες πίσω από αυτές τις καινοτομίες καθώς και η σύνδεσή τους με το IoT.

Στη συνέχεια περιγράφεται η διαδικασία ανταλλαγής μηνυμάτων μέσω του πρωτοκόλλου MQTT, η δομή των μηνυμάτων αυτών και τα πλεονεκτήματα της μεθόδου publish/subscribe.

Στο πέμπτο κεφάλαιο, παρουσιάζεται ο Apache Kafka, μια γνωστή πλατφόρμα λογισμικού για επεξεργασία ροών δεδομένων, και αναλύεται ο τρόπος λειτουργίας του και τα πλεονεκτήμα-

τα που παρέχει στον χρήστη.

Έπειτα, παρουσιάζεται το οπεν σουρσε κατανεμημένο framework Apache Spark, η αρχιτεκτονική του και ο τρόπος που λειτουργεί έτσι ώστε να επεξεργάζεται δεδομένα σε πραγματικό χρόνο.

Στο έβδομο κεφάλαιο, αναλύθηκε το εργαλείο Kafka Streams, οι βασικές του έννοιες, η αρχιτεκτονική του, βασικές του συναρτήσεις και ο τρόπος λειτουργίας του.

Στο επόμενο κεφάλαιο, ασχολούμαστε με το εργαλείο Kafka Flink, τον τρόπο λειτουργίας του καθώς και το πως επεξεργάζεται τα δεδομένα και ποιά είναι τα πλεονεκτήματα και τα μειονεκτήματά του.

Στο προτελευταίο κεφάλαιο παρουσιάζεται η τελική επιλεγμένη αρχιτεκτονική και αναλύεται η λειτουργικότητα της. Παρατίθενται, επίσης, αποτελέσματα προσομοιώσεων που η παρούσα αρχιτεκτονική δοκιμάζεται σε πραγματικά σενάρια χρήσης.

Η παρούσα εργασία ολοκληρώνεται με την παράθεση των συμπερασμάτων και την πρόταση μελλοντικών εφαρμογών.

Κεφάλαιο 2

Internet of Things

2.1 Εισαγωγή

Internet of things ονομάζουμε το δίκτυο που αποτελείται από υλικά αντικείμενα ή από αντικείμενα που έχουν μέσα τους εμφυτευμένους αισθητήρες, λογισμικό ή γενικά ηλεκτρονικά μέρη τα οποία σε συνδυασμό με την δυνατότητα σύνδεσης στο διαδίκτυο τα καταστούν ικανά να συλλέγουν και να ανταλλάσσουν δεδομένα.

Με τον όρο αντικείμενα, που αναφέραμε, εννοούμε μια μεγάλη ποικιλία συσκευών οι οποίες μπορεί να χρησιμοποιούνται για ιατρικούς σκοπούς (αισθητήρες για καρδιακούς παλμούς, ελέγχου γλυκόζης στο αίμα κ.α.), σε βιομηχανικές εγκαταστάσεις (για τον έλεγχο θερμοκρασιών, για δικλείδες ασφαλείας), σε έξυπνα προϊόντα (τηλεοράσεις, αυτοκίνητα , κινητά για αύξηση ασφάλειας και εξατομικευμένη εμπειρία χρήσης).

Με το IOT επιτρέπεται η επικοινωνία όλων αυτών των αντικειμένων-συσκευών μεταξύ τους καθώς και η δυνατότητα τέτοια αντικείμενα και συσκευές να διαχειρίζονται εξ αποστάσεως χρησιμοποιώντας ένα ήδη υπάρχον δίκτυο από υποδομές. Αυτή η δυνατότητα δημιουργεί πολλές ευκαιρίες για ενσωμάτωση των υπολογιστών στον κόσμο μας και μαζί της φέρνει μεγαλύτερη αποδοτικότητα, ακρίβεια καθώς και πολλά οικονομικά οφέλη.

2.2 Ορισμός

Το Διαδίκτυο των πραγμάτων (IoT) θεωρείται ότι συνδέει συσκευές με το διαδίκτυο και χρησιμοποιεί τη σύνδεση αυτή για την παροχή κάποιου είδους χρήσιμης απομακρυσμένης παρακολούθησης ή ελέγχου αυτών των συσκευών. Δεν υπάρχει ένας ενιαίος και αυστηρός ορισμός του όρου. Έχει προταθεί πληθώρα ορισμών για το IoT, μερικοί από τους οποίους είναι οι εξής :

1. Το IoT (ουσιαστικό): Η διασύνδεση, μέσω του διαδικτύου, υπολογιστικών συσκευών. Η διασύνδεση αυτή είναι ενσωματωμένη σε καθημερινά αντικείμενα, παρέχοντάς τους τη δυνατότητα να στείλουν και να λάβουν δεδομένα. (Oxford Λεξικό(2015))

2. IoT: Μια παγκόσμια υποδομή για την κοινωνία των πληροφοριών, παρέχοντας προηγ-

μένες υπηρεσίες μέσω διασυνδεδεμένων (φυσικών και εικονικών) συσκευών ("πράγματα") που βασίζονται στις υφιστάμενες και εξελισσόμενες διαλειτουργικές τεχνολογίες πληροφόρησης και επικοινωνίας. (Διεθνής Ένωση Τηλεπικοινωνιών - ITU)

3. Το IoT είναι ένα πλαίσιο στο οποίο όλα τα πράγματα έχουν εκπροσώπηση και παρουσία στο διαδίκτυο. Πιο συγκεκριμένα, το IoT στοχεύει στο να προσφέρουν νέες εφαρμογές και υπηρεσίες με σκοπό τη γεφύρωση των φυσικών και εικονικών κόσμων, στις οποίες ο τρόπος M2M αντιπροσωπεύει τη γραμμή επικοινωνίας που επιτρέπει την αλληλεπίδραση μεταξύ των πραγμάτων και εφαρμογών σε cloud επίπεδο. (IEEE Communications Magazine)

2.3 Ιστορία και εξέλιξη του IoT

Η έννοια του IOT έγινε διάσημη το 1999 μέσα από τον επιχειρηματία Kevin Ashton.O Ashton, ο οποίος είναι ένας από τους ιδρυτές του Auto-ID center στο MIT, ήταν μέρος μιας ομάδας που ανακάλυψε τον τρόπο να συνδέσει τα αντικείμενα με το Διαδίκτυο μέσω μιας ετικέτας RFID (Radio frequency Identification). [34]

Με αυτόν τον τρόπο αν κάθε αντικείμενο είχε ένα μοναδικό κωδικό αναγνώρισης σε ένα δίκτυο θα γινόταν δυνατή η διαχείριση του και η δυνατότητα "συνομιλίας" του μέσα σε ένα σύνολο με άλλα όμοια αντικείμενα. Εκτός από RFID μπορεί επίσης να χρησιμοποιήσουμε και άλλα μοντέλα για να παρέχουμε ετικέτες σε αντικείμενα όπως QR codes και barcodes.

Τη δεκαετία που ακολούθησε, όλο και περισσότερες εφαρμογές αποκτούσαν συνδεσιμότητα στο διαδίκτυο, ενώ άρχισε να χρησιμοποιείται από όλο και μεγαλύτερο μέρος εταιρικών, βιομηχανικών και καταναλωτικών προϊόντων. Σήμερα, το IoT αποτελεί έναν δημοφιλή όρο για την περιγραφή της δυνατότητας μιας ποικιλίας αντικειμένων, συσκευών, αισθητήρων και καθημερινών ειδών να συνδέονται στο διαδίκτυο. Ωστόσο, ακόμη και σήμερα, απαιτείται περισσότερη ανθρώπινη αλληλεπίδραση και παρακολούθηση μέσω εφαρμογών και διεπαφών.

Ενώ ο όρος IoT αποτελεί σχετικά πρόσφατη σύλληψη η έννοια του συνδυασμού υπολογιστών και δικτύων και η ιδέα της επικοινωνίας τους μπορεί να εντοπιστεί και στην ανακάλυψη του ηλεκτρικού τηλέγραφου (1838 από τον Σάμιουελ Μορς) [35] και του ραδιοφώνου.

Το διαδίκτυο αποτελεί από μόνο του πρωταρχικό στοιχείο του IoT, ξεκινώντας ως μέρος του DARPA (Defense Advanced Research Projects Agency) το 1962, που εξελίχθηκε στο ARPANET το 1969. Τη δεκαετία του 1980, οι πάροχοι εμπορικών υπηρεσιών άρχισαν να υποστηρίζουν τη δημόσια χρήση του ARPANET, επιτρέποντάς του να εξελιχθεί στο σύγχρονο διαδίκτυο. Οι πρώτοι δορυφόροι GPS (Global Positioning Satellites) εμφανίστηκαν το 1933, ενώ γρήγορα ακολούθησαν οι ιδιωτικοί δορυφόροι που τοποθετήθηκαν σε τροχιά. Οι δορυφόροι είναι βασικοί για την επικοινωνία στο IoT. Πρωταρχικό ρόλο είχε και η έλευση του πρωτοκόλλου IPv6 (Internet Protocol version 6), του βασικού πρωτοκόλλου επικοινωνίας πάνω στο οποίο έχει χτιστεί ολόκληρο το διαδίκτυο. [35]

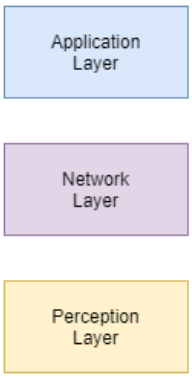
Η πρώτη "έξυπνη" συσκευή έκανε την εμφάνισή της το 1990, όταν ο John Romkey, σε συνεργασία με τον Simon Hackett, δημιούργησε μια τοστιέρα, που θα μπορούσε να ενεργο-

ποιηθεί και να απενεργοποιηθεί μέσω του διαδικτύου. Η τοστιέρα συνδέθηκε στο διαδίκτυο μέσω δικτύωσης TCP/IP. Από τις πρώτες συσκευές IoT είναι και ο αυτόματος πωλητής αναψυκτικών Coca Cola, το 1982 στο Πανεπιστήμιο Carnegie Mellon. Οι φοιτητές του πανεπιστημίου συνέδεσαν το μηχάνημα στο διαδίκτυο, προκειμένου να γνωρίζουν ανά πάσα στιγμή αν υπάρχουν αναψυκτικά, καθώς και το πότε τοποθετήθηκαν, ώστε να γνωρίζουν εάν έχουν ψυχθεί αρκετά [4]. Σήμερα, το IoT υιοθετείται και αναπτύσσεται με ταχείς ρυθμούς και αυτό δεν προβλέπεται να αλλάξει στα χρόνια που ακολουθούν. Καταλυτικός παράγοντας αποτελεί η συνεχώς εξελισσόμενη τεχνολογία, με όλο και περισσότερες συσκευές να είναι συνδεδεμένες στο διαδίκτυο[34].

2.4 Αρχιτεκτονική του IoT

Το IoT δεν έχει κάποιο συγκεκριμένο καθολικό consensus όσον αφορά την αρχιτεκτονική του λόγου της μεγάλης ποικιλίας των συσκευών και τον διαφορετικών αναγκών χρηστών που εξυπηρετεί.
Γενική αναπαράσταση IoT με παραδείγματα [20]:
Device Layer: Περιέχει έναν ή παραπάνω διαφορετικών ειδών αισθητήρες μικρούς σε μέγεθος, χαμηλούς σε κόστος με μικρή κατανάλωση ενέργειας.
Communication Layer: Περιέχει Ρεστ πρωτόκολλα και άλλα applications level πρωτοκολλα (MQTT / HTTP)
Bus Layer: Σημαντικό επίπεδο, περιέχει HTTP Server και συχνά MQTT Broker. Συνδυάζει και συγκεντρώνει τα δεδομένα μέσω Gateway καθώς και "μεταμορφώνει" τα δεδομένα έτσι ώστε να μπορούν να χρησιμοποιηθούν από διαφορετικά πρωτόκολλα.
Event Processing and Analytics: Προεπεξεργασία με εφαρμογή cloud και edge computing.
Application Layer: Web based Engine (Web Portal) για επικοινωνία με άλλα API (API Management), επικοινωνεί και με συσκευές εκτός του δικτύου (Dashboard).

2.4.1 Αρχιτεκτονική 3 επιπέδων (Three-layer architecture)

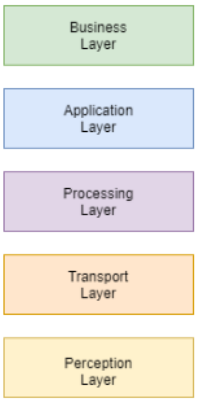


Σχήμα 2.1: Αρχιτεκτονική 3 επιπέδων

Η πιο βασική μορφή αρχιτεκτονικής είναι 3 επιπέδων [20][41].
(i) Perception Layer: Επίπεδο αισθητήρων: συγκεντρώνονται πληροφορίες για το περιβάλλον καθώς και τα αντικείμενα που βρίσκονται και αλληλεπιδρούν σε αυτό.
(ii) Network Layer: Υπεύθυνο για την σύνδεση των συσκευών που χρησιμοποιούν το IoT και γενικά των συσκευών δικτύου. Σε αυτό το επίπεδο γίνεται ακόμα, η διαβίβαση και η προεπεξεργασία των δεδομένων που συλλέγονται από το παρακάτω στρώμα.

(iii) Application Layer: Σε αυτό το στρώμα βρίσκεται το σύνολο των εφαρμογών στις οποίες χρησιμοποιείται το IoT. Είναι υπεύθυνο για την διανομή των πληροφοριών, μέσω υπηρεσιών, στους χρήστες που τις διαχειρίζονται και χρησιμοποιούν.

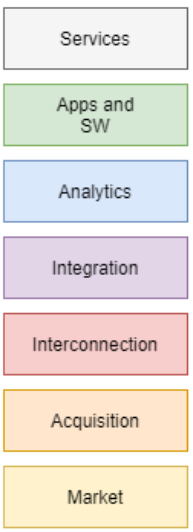
2.4.2 Αρχιτεκτονική 5 επιπέδων (Five-layer architecture)



Σχήμα 2.2: Αρχιτεκτονική 5 επιπέδων

Για μεγαλύτερη λεπτομέρεια η αρχιτεκτονική του Σχήματος 2.1 μπορεί να χωριστεί επιμέρους και να προκύψει αυτή των 5 στρωμάτων [20][41].
(i) Perception Layer: Όπως περιγράφεται παραπάνω στην αρχιτεκτονική 3 στρωμάτων.
(ii) Transport Layer: Υπεύθυνο για την μεταφορά δεδομένων από τους αισθητήρες ανάμεσα στα στρώματα μέσω του δικτύου (wireless, 3G, 4G, 5G, LAN, Bluetooth, RFID, NFC).
(iii) Processing Layer: Αποθηκεύει, αναλύει και επεξεργάζεται των τεράστιο όγκο δεδομένων που παράγονται στα παρακάτω στρώματα, χρησιμοποιώντας τεχνικές όπως cloud computing και Big Data Analysis.
(iv) Application Layer: Όπως περιγράφεται παραπάνω στην αρχιτεκτονική 3 στρωμάτων.
(v) Business Layer: Διαχείριση του IoT οικοδομήματος από ένα σύνολο διαφορετικών οπτικών και εξασφάλιση ασφάλειας και ιδιωτικότητας στον χρήστη. Στενά συνδεδεμένο με την εμπορική χρήση του IoT ως τεχνολογία εφαρμοσμένη σε παραγωγικές μονάδες με σκοπό το κέρδος.

2.4.3 Αρχιτεκτονική 7 επιπέδων (Seven-layer architecture)



Σχήμα 2.3: Αρχιτεκτονική 7 επιπέδων

Είναι η πλέον χρησιμοποιούμενη αρχιτεκτονική για να περιγράψει τη δομή ενός συστήματος IoT. [41].
(i) Market: Μπορεί να είναι ένα έξυπνο σπίτι ή έξυπνη υγεία κλπ.
(ii) Acquisition: Αποτελείται από σένσορες που συλλέγουν δεδομένα από το περιβάλλον, όπως σένσορες θερμοκρασίας, υγρασίας, κάμερες κλπ.
(iii) Interconnection: Επιτρέπει στα δεδομένα που προέρχονται από τους σένσορες,

να μεταφερθούν, συνήθως σε ένα data center, στο cloud κλπ. Αποτελούν μεγάλα δεδομένα.

(iv) Integration: Τα δεδομένα συγκεντρώνονται και προστίθενται σε άλλα, γνωστά σύνολα δεδομένων.

(v) Analytics: Τα συνδυασμένα αυτά δεδομένα στη συνέχεια, αναλύονται, χρησιμοποιώντας τεχνικές μηχανικής μάθησης και εξόρυξης γνώσης από δεδομένα.

(vi) Apps and SW: Για να υλοποιηθούν τέτοιες κατανεμημένες εφαρμογές απαιτείται το επίπεδο αυτό, όπως επίσης και λογισμικό όπως SDN (software defined networking), SOA (services oriented architecture) κ.α.

(vii) Services: Υπεύθυνο για την παροχή υπηρεσιών προς το χρήστη, όπως ενεργειακή διαχείριση, εκπαίδευση, μεταφορές κ.α.

Επιπλέον των 7 αυτών επιπέδων, απαιτείται η ασφάλεια και η διαχείριση κάθε επιπέδου (security και management applications) [46]

2.5 Πρωτόκολλα IoT

Λαμβάνοντας υπόψη την αρχιτεκτονική 7 επιπέδων, θα αναλυθεί το επίπεδο Interconnection, το οποίο, όπως φαίνεται και στην εικόνα που ακολουθεί, μπορεί να θεωρηθεί ότι από μόνο του αποτελείται από πλήθος επιπέδων [46]. Το επίπεδο datalink συνδέει δύο στοιχεία του IoT, τα οποία μπορεί να είναι δύο σένσορες, ή ένας σένσορας και μία συσκευή gateway, που συνδέει ένα σύνολο αισθητήρων με το διαδίκτυο. Συνήθως, απαιτούνται πολλαπλοί αισθητήρες που επικοινωνούν και συλλέγουν πληροφορίες, προτού φτάσουν στο διαδίκτυο.

Τα διάφορα τμήματα του συστήματος πρέπει να βρουν έναν τρόπο να επικοινωνούν, είτε πρόκειται για ενσύρματη (wired), είτε για ασύρματη (wireless) σύνδεση. Το πρωτόκολλο επικοινωνίας είναι μια δέσμη κανόνων στους οποίους στηρίζεται η επικοινωνία των συσκευών σε ένα δίκτυο. Είναι απαραίτητα στις εφαρμογές IoT για τη σύνδεση των διαφόρων τμημάτων του συστήματος, τα οποία μπορούν να ελεγχθούν με λογισμικό βασισμένο στο cloud. Ειδικά πρωτόκολλα έχουν σχεδιαστεί για τη δρομολόγηση μεταξύ των αισθητήρων και είναι μέρος του routing layer. Τα πρωτόκολλα του session layer protocols επιτρέπουν την επικοινωνία μεταξύ πλήθους αντικειμένων του IoT υποσυστήματος επικοινωνίας. Τέλος, πλήθος πρωτοκόλλων security και management έχουν αναπτυχθεί, όπως φαίνεται στο παρακάτω σχήμα [46].

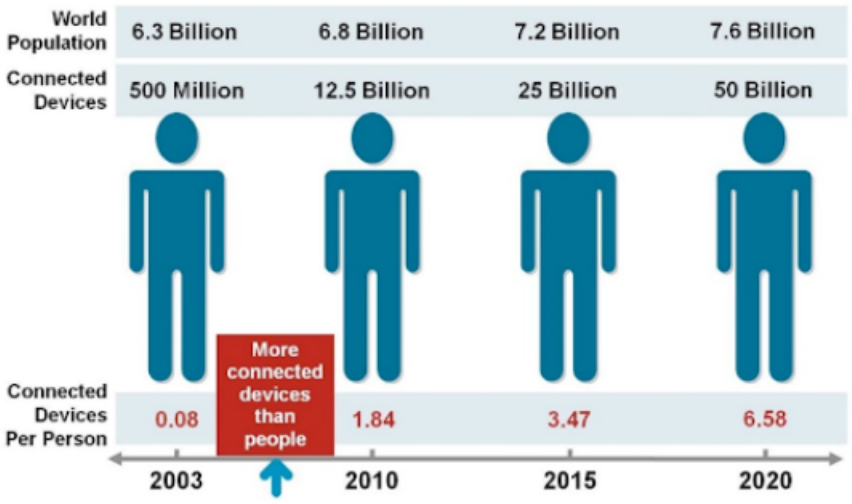
Όπως θα δούμε και στην συνέχεια στην παρούσα εργασία μας ενδιαφέρει ιδιαίτερα το Session πρωτόκολλο MQTT το οποίο θα αναλύσουμε εκτενώς σε επόμενο κεφάλαιο.

Session		MQTT, SMQTT, CoRE, DDS, AMQP , XMPP, CoAP, …	**Security**	**Management**
Network	**Encapsulation**	6LowPAN, 6TiSCH, 6Lo, Thread, …	TCG, Oath 2.0, SMACK, SASL, ISASecure, ace, DTLS, Dice, …	IEEE 1905, IEEE 1451, …
	Routing	RPL, CORPL, CARP, …		
Datalink		WiFi, **Bluetooth Low Energy**, **Z-Wave**, **ZigBee Smart**, **DECT/ULE**, 3G/LTE, NFC, **Weightless**, **HomePlug GP**, **802.11ah**, **802.15.4e**, **G.9959**, **WirelessHART**, **DASH7**, ANT+, **LTE-A**, **LoRaWAN**, …		

Σχήμα 2.4: Πρωτόκολλα IoT

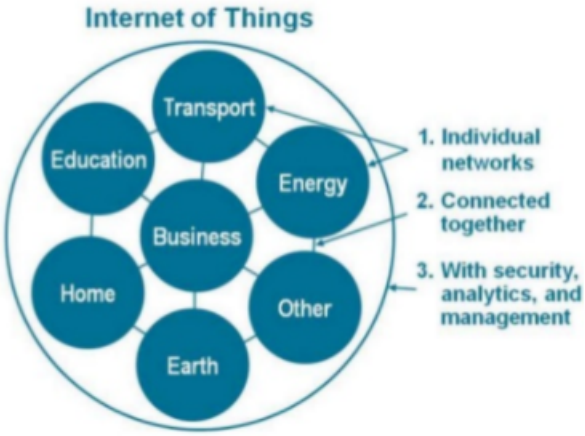
2.6 Παρόν, Μέλλον & Εφαρμογές του IoT

2.6.1 Παρόν & Μέλλον



Σχήμα 2.5: Χρονική εξέλιξη του IoT

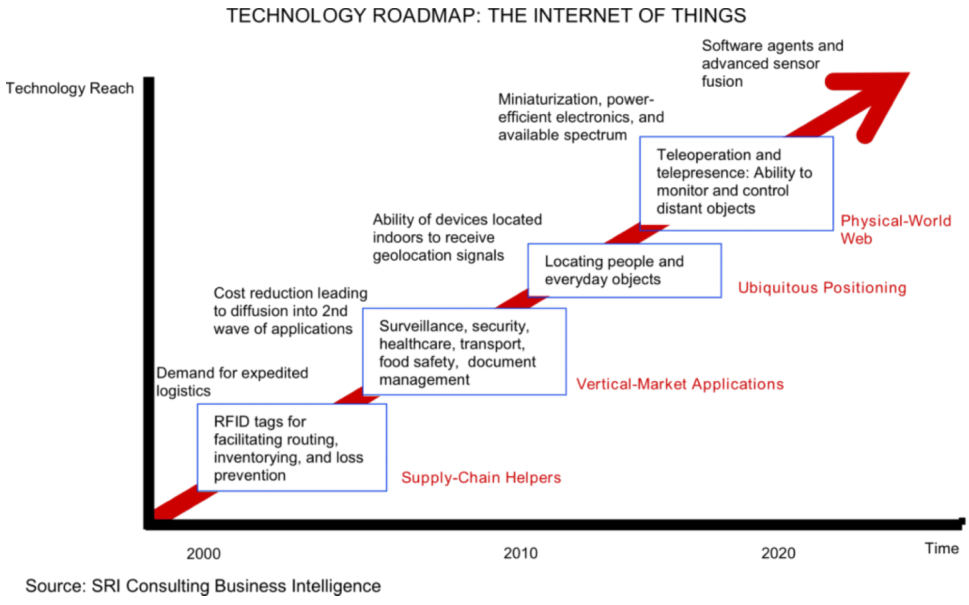
Όπως είναι λοιπόν λογικό στην σημερινή εποχή λόγω του τεράστιου αριθμού των συσκευών το IoT δεν αποτελείται από ένα καθόλα ενιαίο δίκτυο. Παρά μοιάζει περισσότερο στην παρακάτω εικόνα.



Σχήμα 2.6: Πραγματική απεικόνιση του IoT

Χωρίς όμως αυτό να προκαλεί οποιαδήποτε αλλαγή στις δυνατότητες που αναφέραμε μέχρι στιγμής.

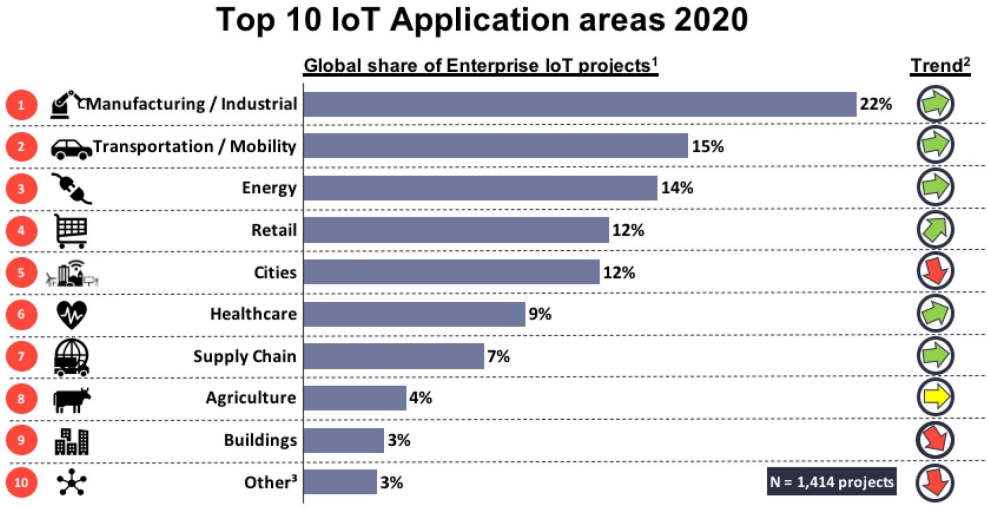
Όπως βλέπουμε το IoT εξελίσσεται ραγδαία και αν και δεν μπορούμε να ξέρουμε τίποτα για το μέλλον, υπάρχει μεγάλη πιθανότητα το IoT να βρίσκεται παντού γύρω μας έτσι ώστε να καλύψει τις όλο και αυξανόμενες ανάγκες για live δεδομένα, επεξεργασία δεδομένων και χειρισμό εξ αποστάσεως. Προβλέπεται ότι στα επόμενα 5 χρόνια θα υπάρξει κατακόρυφη ανάπτυξη σε αυτά που ονομάζονται Industrial και Consumer IoT με ανάπτυξη "έξυπνων" αυτοκινήτων, "έξυπνων" πόλεων και γενικά ανάπτυξη της τεχνητής νοημοσύνης σε όλους τους τομείς. [39]



Σχήμα 2.7: Τεχνολογικός χάρτης του IoT

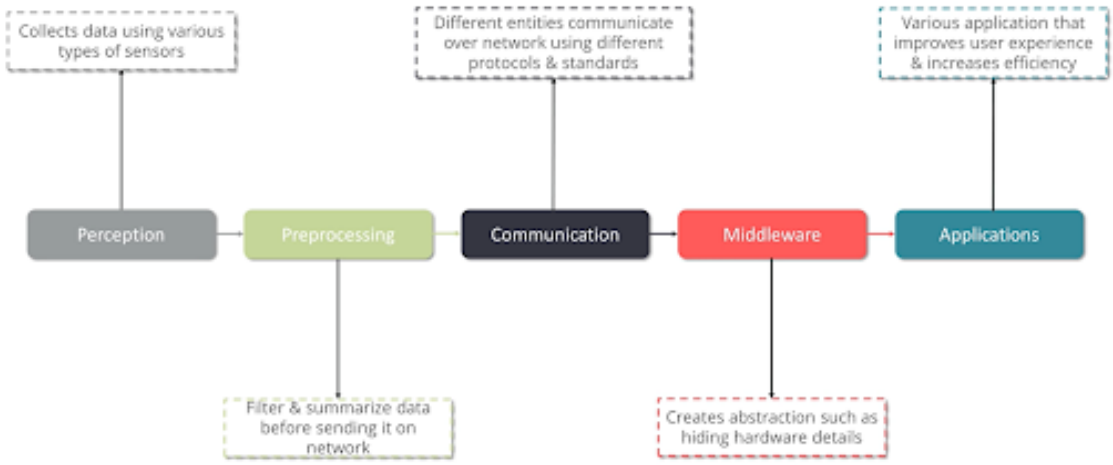
2.6.2 Εφαρμογές

Εφαρμογές του IoT συναντόνται σε τομείς όπως: [40]
Wearable: Τομέας με μεγάλη ανάπτυξη λόγω της αυξανόμενης ζήτησης της αγοράς. Είδη προσελκύει πολλές εταιρείες κολοσσούς και startups.
Υγεία: Αυξανόμενη ζήτηση για "έξυπνα" προϊόντα υγείας. Ήδη αρκετά προϊόντα αυτού του τομέα βρίσκονται στην αγορά με υπόσχεση για βελτίωση στο μέλλον. Έξυπνα αναπηρικά αμαξίδια, υπενθύμιση για φαρμακευτική αγωγή και γενικά παρακολούθηση υγείας από εφαρμογές αποτελούν τα πλέον διαδεδομένα παραδείγματα.
Γεωργία: Έξυπνοι τρόποι για καλλιέργεια της γης και εκμετάλλευση της με έλεγχο υγρασίας και θρεπτικών συστατικών. Παρακολούθηση κτηνοτροφικών μονάδων με αισθητήρες στα ζώα και γεωγραφικό εντοπισμό τους.
Έξυπνες πόλεις: Έργο που εκπονείται από την κυβέρνηση της Ινδίας με σκοπό την δημιουργία αυτοεξυπηρετούμενης έξυπνης πόλης και σπιτιών.
Αυτοκινητοβιομηχανία: Μεγάλη αύξηση στην ζήτηση αυτοκινήτων δίχως οδηγό με επιπλέον συστήματα ασφαλείας γεωγραφικό εντοπισμό και δυνατότητα επικοινωνίας με άλλες συσκευές. Σε δοκιμαστικό ακόμα στάδιο από την Google.
Έξυπνο πλέγμα: Προσπάθεια για έλεγχο ενέργειας και νερού του κάθε σπιτιού και ακινήτου μέσω αισθητήρων και του Internet.
Βιομηχανικό Internet: Ήδη εν μέρη εφαρμοζόμενο με την αντικατάσταση όλο και περισσότερου έμβιου εργατικού δυναμικού με μηχανές για μικρότερη πιθανότητα λάθους και μικρότερη επικινδυνότητα και νομική ευθύνη της εταιρείας.



Σχήμα 2.8: Top 10 IoT applications in 2020. Πηγή: `https://iot-analytics.com/top-10-iot-applications-in-2020/`

2.7 Τεχνολογικά εμπόδια βαθμίδων IoT [20]



Σχήμα 2.9: Βαθμίδες IoT

Perception: Στο συγκεκριμένο επίπεδο δεν υπάρχουν πολλοί ανασταλτικοί παράγοντες που προκαλούν προβλήματα. Παρόλα αυτά υπάρχουν συγκεκριμένα όρια που τοποθετούνται στο μέγεθος και το κόστος των αισθητήρων που επιδέχονται βελτίωση.
Preprocessing: Στο συγκεκριμένο επίπεδο τα εμπόδια προκαλούνται από τα όρια που τοποθετεί η επεξεργασία όλων των δεδομένων στο cloud. Τα οποία είναι:

- Mobility: Οι περισσότερες συσκευές είναι φορητές με αποτέλεσμα να αλλάζουν συχνά δίκτυα πράγμα που προκαλεί προβλήματα στην επικοινωνία.
- Reliable and real-time actuation: Live εφαρμογές χρειάζονται "ζωντανά" δεδομένα και είναι ευαίσθητες σε καθυστερήσεις μεταφοράς.
- Scalability: Η ύπαρξη πολλών συσκευών επιβαρύνει τις καθυστερήσεις στην μεταφορά και την ροή των πληροφοριών.

Communication: Τα βασικά εμπόδια στην βαθμίδα της επικοινωνίας είναι:

- Addressing and Identification: Κάθε συσκευή πρέπει να φέρει μοναδικό αναγνωριστικό, το οποίο είναι η διεύθυνση της.
- Low Power Communication: Πρέπει η επικοινωνία των συσκευών να γίνεται με τις μικρότερες δυνατές ενεργειακές απαιτήσεις.
- Απλά αποδοτικά πρωτόκολλα μεταφοράς (routing) με μικρές απαιτήσεις σε μνήμη.
- Γρήγορη επικοινωνία χωρίς παρεμβολές και απώλειες.

Middleware: Διαλέγοντας τα εργαλεία και τα ενδιάμεσα λογισμικά που θα χρησιμοποιήσουμε πρέπει να έχουμε στο μυαλό μας τα παρακάτω:

- Interoperability and programming Abstraction: Κάθε εφαρμογή χρησιμοποιεί διαφορετικό τρόπο για να λαμβάνει και να μεταδίδει τα δεδομένα συνεπώς θα πρέπει να υπάρχει μια κάποιο αφηρημένο σχέδιο για το σύνολο τον εργαλείων που χρησιμοποιεί η εφαρμογή.
- Device Discovery and Management: Αν έχουμε πολλές συσκευές θα πρέπει η κάθε μία να είναι ξεχωριστά προσβάσιμη και διαχειριζόμενη.
- Scalability: Βοηθάει η ύπαρξη εργαλείων τα οποία μας επιτρέπουν να κρατάμε σταθερό τον κορμό της εφαρμογής ενώ αυτή μεγαλώνει σε απαιτήσεις και πόρους.
- Big Data Analytics: Βοηθούν αν περιλαμβάνονται λόγω του μεγάλου όγκου δεδομένων.
- Security and Privacy
- Cloud Services
- Context Detection

Application: Τα εμπόδια και τα όρια τοποθετούνται ανάλογα με το τι είδους εφαρμογή θέλουμε να υλοποιήσουμε.

2.8 Κριτική IoT

Η ευρεία διάδοση του IoT έχει όμως προκαλέσει και έντονες κριτικές καθώς φαίνεται οι δυνατότητες που παρέχει να υπόκεινται κάθε φορά στην κρίση του χρήστη πολλές φορές εγείροντας ερωτηματική σχετικά με:

1. Την ιδιωτικότητα
2. Την Ασφάλεια
3. Την Αυτονομία και τον Έλεγχο
4. Την καθοδήγηση της κοινής γνώμης
5. Την Πολιτική εκμετάλλευση
6. Το περιβάλλον
7. Και τον γενικότερο επηρεασμό των ανθρώπων πάνω σε ηθικά διλήμματα.

Παρόλα αυτά καμία τεχνολογία δεν είναι καλή ή κακή από μόνη της ο τρόπος χρήσης της είναι που της δίνει θετικό ή αρνητικό πρόσημο.

Κεφάλαιο 3

Cloud Computing & Edge Computing

3.1 Cloud Computing

3.1.1 Εισαγωγή

Το υπολογιστικό νέφος (cloud computing) είναι η διάθεση υπολογιστικών υπηρεσιών μέσω διαδικτύου. Αυτές οι υπηρεσίες αποτελούνται πρακτικά από ένα σύνολο εργαλείων, εφαρμογών και πόρων που παρέχονται από κεντρικά συστήματα που βρίσκονται απομακρυσμένα από τον τελικό χρήστη. Μέσω αυτών, ο χρήστης έχει δυνατότητες αποθήκευσης δεδομένων (data storage), χρήσης υπολογιστικών πόρων-servers, διαχείριση βάσεων δεδομένων (database management), εκτέλεσης εφαρμογών καθώς και πιο ειδικών δυνατοτήτων όπως το blockchain και η τεχνητή νοημοσύνη (AI) [22].

Γνωστές εφαρμογές που χρησιμοποιούν cloud computing είναι: Google Drive, Apple iCloud, Dropbox και Slack.

Σχήμα 3.1: Cloud εφαρμογές

3.1.2 Ιστορία και εξέλιξη του Cloud Computing

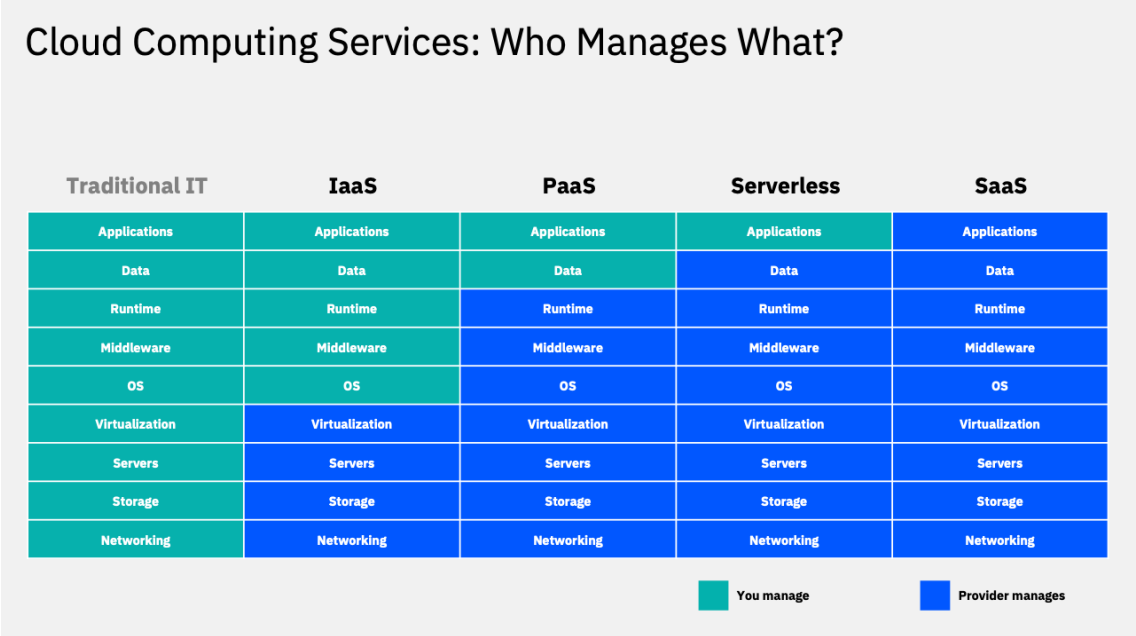
Το cloud computing ως όρος είναι σχετικά πρόσφατος κάνοντας την εμφάνιση του στις αρχές του 2000, η φιλοσοφία όμως της υπηρεσίας που παρέχει (computing-as-a-service) προϋπήρχε από τις δεκαετίες 1950-1960 όταν υπηρεσίες πληροφορικής ενοικίαζαν σε εταιρείες υπολογιστικό χρόνο σε κεντρικούς υπολογιστές (mainframe).

Παρόλα αυτά η παραπάνω υπηρεσία ξεπεράστηκε προσωρινά από την δημιουργία των data centers και την απομυθοποίηση του υπολογιστής (PC) ως είδος πολυτελείας. Μόνο και μόνο για να επανεμφανιστεί στις αρχές του 1990 χάρη στην τεχνολογία των Εικονικών Ιδιωτικών Δικτύων (Virtual Private Networks) και του Internet που είχε κάνει την εμφάνισή του λίγα χρόνια πριν.

Μέχρι το 2000, κολοσσοί της πληροφορικής όπως η Amazon, η Microsoft, η IBM και η Google, ασχολήθηκαν με την ανάπτυξη και την παροχή υπηρεσιών Υπολογιστικού Νέφους και συνεχίζουν και σήμερα με κάποια παραδείγματα να είναι το Oracle Cloud από την IBM, το Amazon Web Services από την Amazon και η πλατφόρμα Windows Azure από την Microsoft [1].

3.1.3 Μοντέλο ανάπτυξης

Υπάρχουν τρία βασικές κατηγορίες μοντέλων υπολογιστικού νέφους [8, 37, 29, 44, 23]:
Infrastructure-as-a-Service (IaaS): Πρόκειται για τις απλές-βασικές συσκευές με υλική υπόσταση (raw υπολογιστές) όπως είναι οι εικονικοί υπολογιστές, οι διακομιστές, οι συσκευές αποθήκευσης, η μεταφορά μέσω δικτύου. Επιτρέπει σε εταιρείες να νοικιάζουν και να χτίσουν το δικό τους εικονικό data center χωρίς να πρέπει αυτό να έχει φυσική υπόσταση στο χώρο της εταιρείας αποφεύγοντας έτσι κόστος συντήρησης και εξοικονομώντας χώρο. Αυτή η πρακτική ακολουθείτε κυρίως όταν χρειάζεται να υπάρχει καθολική εποπτεία πάνω σε όλα τα κομμάτια μιας εφαρμογής πράγμα όμως που προβλέπει την ύπαρξη τεχνικά καταρτισμένου προσωπικού στην εταιρεία για την υποστήριξη.
Software-as-a-Service (SaaS): Πρόκειται για την πλέον διαδεδομένη χρήση του cloud computing. Αποτελείται από εφαρμογές (λογισμικό) το οποίο χρησιμοποιεί ο πελάτης μέσω του διαδικτύου έναντι κάποιας μηνιαίας ή ετήσιας συνδρομής. Αξίζει να σημειωθεί ότι το λειτουργικό σύστημα καθώς και η αρχιτεκτονική της εφαρμογής δεν επηρεάζει τον τελικό χρήστη. Γνωστό παράδειγμα αυτης της μορφή νέφους είναι το Microsoft Office 365.
Platform-as-a-Service (PaaS): Παρέχει μια ολοκληρωμένη πλατφόρμα για ανάπτυξη, εκτέλεση και διαχείριση εφαρμογών χωρίς την πολυπλοκότητα, το κόστος και την έλλειψη ευελιξίας που ακολουθούν την δημιουργία και την διατήρηση μιας τέτοιας πλατφόρμας τοπικά. Η εταιρείες που προσφέρουν τέτοιες υπηρεσίες παρέχουν servers, αποθηκευτικό χώρο, κατάλληλα δίκτυα, βάσεις δεδομένων καθώς και διαφορετικά λειτουργικά συστήματα. Οι χρήστες μπορούν να χρησιμοποιήσουν τις εν λόγω υπηρεσίες δωρεάν ή πληρώνοντας ένα ποσό ανάλογα με την χρήση και τον σκοπό της χρήσης των πόρων.
Function-as-a-Service(FaaS): Η υπηρεσία είναι ένα υποσύνολο αυτού που ονομάζουμε serverless computing, προσφέρει στους χρήστες την δυνατότητα να εκτελούν συγκεκριμένα κομμάτια από τον κώδικα (συναρτήσεις) ως απάντηση σε συγκεκριμένα γεγονότα. Όλα τα προαπαιτούμενα για την εκτέλεση του κώδικα (λογισμικό, virtual machine, web server, hardware κ.τ.λ.) εκτός από τον κώδικα παρέχονται από την cloud service υπηρεσία σε πραγματικό χρόνο δίνοντας έτσι μεγάλη ελευθερία στον χρήστη.



Σχήμα 3.2: Αρμοδιότητες στα Cloud Computing Services

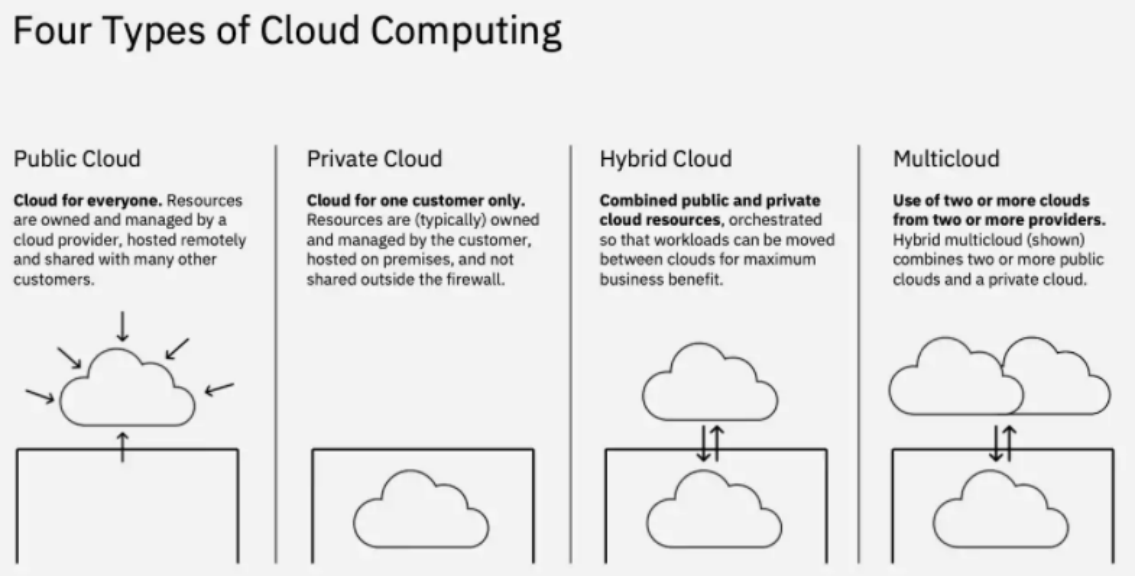
3.1.4 Μορφές υπολογιστικού νέφους

Υπάρχουν 3 βασικά μοντέλα ανάπτυξης του νέφους [37, 29].
Public Cloud: Ανήκει και χρησιμοποιείται από ένα τρίτο φορέα παροχής υπηρεσιών και προσφέρει (servers, υπολογιστικούς πόρους και χώρο αποθήκευσης) μέσω του διαδικτύου. Όλη αυτή η δομή ανήκει και συντηρείται από την εταιρεία.
Private Cloud: Οι υποδομές χρησιμοποιούνται αποκλειστικά από την εταιρεία, όλες οι ενέργειες που αφορούν την αποθήκευση και την διατήρηση των δεδομένων βρίσκονται τοπικά στο data center της εταιρείας. Συνήθως προτιμάται από εταιρείες που απαιτούν την μυστικότητα και ασφάλεια των δεδομένων καθώς και τον πλήρη έλεγχο αυτών, που όμως χρειάζονται πρόσβαση σε εφαρμογές του νέφους.
Hybrid Cloud: Είναι ένας συνδυασμός δημόσιου (public) και ιδιωτικού (private) cloud έτσι ώστε μέσω κατάλληλων εργαλείων και τεχνολογιών οι εφαρμογές και τα δεδομένα να μοιράζονται τους. Με αυτόν τον τρόπο παρέχεται ευκολία και ταχύτητα σε κομμάτια όπως η ανάπτυξη και η βελτίωση υπαρχουσών υποδομών καθώς παράλληλα διατηρείται η ασφάλεια και η μυστικότητα που χρειάζεται η κάθε εταιρεία.

Υπάρχει επίσης ένα μοντέλο που αξίζει να αναφέρουμε εφόσον έχει μεγάλη απήχηση σε εταιρικά περιβάλλοντα αυτό το μοντέλο είναι το **Multicloud** [29]

Το Multicloud δεν αποτελεί ξεχωριστό μοντέλο διότι πρακτικά είναι ένας συνδυασμός των βασικών χωρίς αυτό να σημαίνει ότι δεν είναι το ίδιο σημαντικό. Η διαφορά του είναι

ότι χρησιμοποιεί 2 ή περισσότερα cloud διαφορετικών προμηθευτών. Όπως είναι φανερό αυτό δεν είναι ικανό για να ορίσει ένα νέο τύπο μοντέλου είναι όμω πολύ βοηθητικό για εταιρείες ακόμα και για χρήστες να μπορούν να συνδυάζουν και να διαλέγουν από τις υπηρεσίες πολλών διαφορετικών προμηθευτών αυτές που τους είναι πιο χρήσιμες με βάση την δουλειά. Για παράδειγμα, μπορεί μια εταιρεία να χρησιμοποιεί το cloud ενός προμηθευτή για τις SaaS υπηρεσίες του email και μια SaaS εφαρμογή για την επεξεργασίας εικόνας από έναν άλλο προμηθευτή. Μπορεί επίσης η εταιρεία να έχει και εσωτερικό δικό της cloud (private) με δικές τις (homegrown) εφαρμογές που να λειτουργούν και να αλληλεπιδρούν με τις άλλες δύο όλες μαζί. Το τελευταίο παράδειγμα ονομάζεται Hybrid multicloud και είναι ουσιαστικά το πλέον διαδεδομένο μοντέλο.



Σχήμα 3.3: Μοντέλα ανάπτυξης Cloud

3.1.5 Πλεονεκτήματα, Περιορισμοί & Χρήσεις

Πλεονεκτήματα [37]

Κόστος: Το υπολογιστικό νέφος εξαλείφει την οικονομική επιβάρυνση της αγοράς κατάλληλου hardware και λογισμικού. Επίσης το κόστος των υποδομών που χρειάζονται για να εγκατασταθούν και να εκτελούνται οι διάφορες εφαρμογές δεν είναι πλέον πρόβλημα της εταιρείας που χρησιμοποιεί το cloud. Σε πολλές περιπτώσεις μάλιστα μειώνεται σημαντικά και η ανάγκη σε ενέργεια που χρειάζεται για να λειτουργήσει ένα τοπικό data center καθώς και το προσωπικό που χρειάζεται για την αντιμετώπιση προβλημάτων που προκύπτουν και την συντήρηση του.
Ταχύτητα: Οι περισσότερες εφαρμογές και υπηρεσίες στο cloud είναι πολύ γρηγορότερες από τις αντίστοιχες που δημιουργούνται τοπικά αυτό γίνεται χάρη στους ανώτερους υπολογιστικούς πόρους όπως και στο ότι αναβαθμίζονται καθημερινά και δημιουργούνται από διεθνείς εταιρείες με πολυπρόσωπο εξειδικευμένο προσωπικό.

Κλιμακωσιμότητα: Το νέφος παρέχει, και συνεπώς χρεώνει τον χρήστη για τα απολύτως απαραίτητα εργαλεία και πόρους που χρειάζεται η εφαρμογή του (αποθηκευτικό χώρο, IT πόρους). Επίσης μπορεί αυτά τα εργαλεία και οι πόροι να αλλάξουν ανάλογα με την βούληση του χρήστη σε πολύ σύντομο χρονικό διάστημα, η κλιμακωσιμότητα δηλαδη γίνεται πολύ πιο εύκολη στο νέφος από ότι τοπικά.
Παραγωγικότητα: Μπορεί η κάθε εταιρεία να επικεντρωθεί καλύτερα και πιο αποδοτικά στο δικό της παραγωγικό μοντέλο, προϊόν και στόχο χωρίς να την απασχολούν βασικά προβλήματα όπως η συντήρηση servers και η ανάγκη για συνεχή εκσυγχρονισμός των πόρων.
Απόδοση: Οι περισσότερες υπηρεσίες του νέφους εκτελούνται σε ένα παγκόσμιο δίκτυο από data center τα οποία αναβαθμίζονται τακτικά από με τα υλικά και λογισμικό τελευταίας τεχνολογιας με αποτέλεσμα να προσφέρουν την καλύτερη δυνατή υπηρεσία στους χρήστες τους.
Αξιοπιστία: Οι υπηρεσίες του νέφους παρέχουν πολλές δικλείδες ασφαλείας με data-backup και ανάνηψη από σφάλματα "ασφαλίζοντας" πρακτικά τα δεδομένα του κάθε χρήστη.
Ασφάλεια: Πολλές εταιρείες που παρέχουν υπηρεσίες cloud προσφέρουν και τελευταίας τεχνολογίας εργαλεία ασφάλειας που διασφαλίζουν τα δεδομένα του χρήστη και του δίνουν την δυνατότητα καλύτερης εποπτείας πάνω σε αυτά θωρακίζοντας τις εφαρμογές του και τις υποδομές του από πολλών διαφορετικών ειδών επικείμενες απειλές.

Περιορισμοί [22, 32]
Τεχνικά προβλήματα: Υπάρχουν στιγμές που το σύστημα μπορεί να έχει μερικές πρόσκαιρες δυσλειτουργίες είτε λόγω φυσικών καταστροφών είτε λόγω ανθρώπινου λάθους στον κώδικα. Επιπλέον μπορεί τα τεχνικά προβλήματα να προκύπτουν από την πλευρά του χρήστη, όπως για παράδειγμα να μην υπάρχει μια αξιόπιστη σύνδεση στο διαδίκτυο.
Ασφάλεια στο σύννεφο: Ως χρήστης πρέπει να βεβαιώνεται κανείς ότι επιλέγει τον πλέον αξιόπιστο πάροχο υπηρεσιών για την ασφάλεια των πληροφοριών του. Πολύ σημαντικό θέμα ιδιαίτερα αν έχουμε να κάνουμε με δεδομένα προσωπικού απορρήτου ή οικονομικά δεδομένα.
Επιρρέπεια σε επιθέσεις: Η αποθήκευση πληροφοριών στο σύννεφο ενδέχεται να κάνει τα δεδομένα των χρηστών ευάλωτα σε εξωτερικές επιθέσεις χάκερ και διαδικτυακές απειλές ακόμα.
Έλλειψη υποστήριξης: Σε ορισμένους παρόχους υπηρεσιών δεν είναι ακόμα ικανοποιητικό επίπεδο εξυπηρέτησης για εφαρμογές ιστού.

Χρήσεις [37, 29]

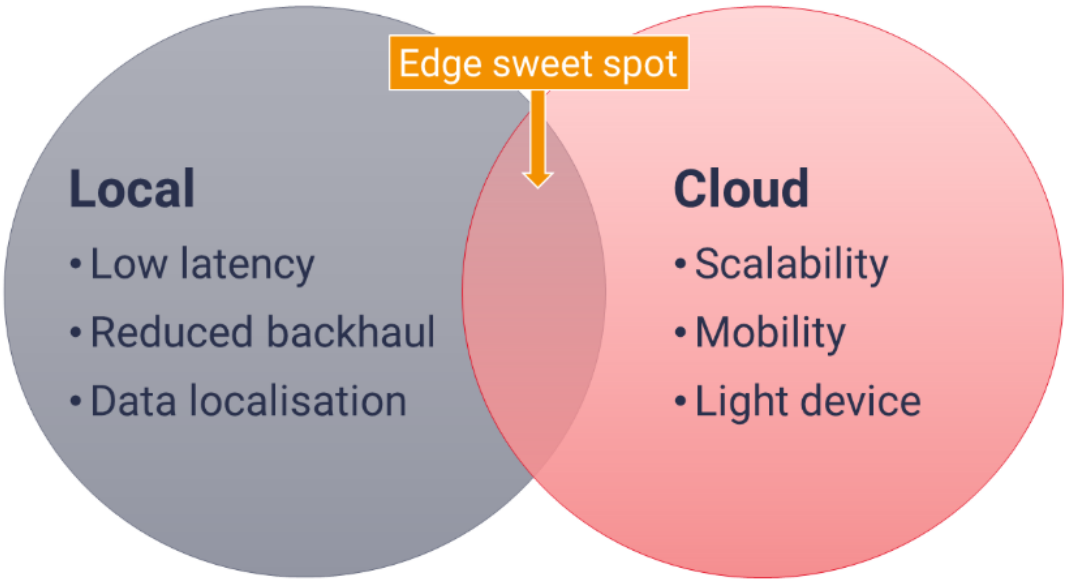
Το cloud computing χρησιμοποιείται πλέον παντού γύρω μας είτε για την δημιουργία μιας εφαρμογής, είτε για streaming video και ήχου, είτε ως δικλείδα ασφαλείας για ανάκτηση δεδομένων είτε για τον έλεγχο εφαρμογών και νέων λογισμικών. Ακόμα χρησιμοποιείται σε μεγάλο βαθμό για την ανάπτυξη καινούργιων τεχνολογιών όπως είναι εφαρμογές τεχνητής νοημοσύνης, machine learning και deep learning καθώς και στην εξέλιξη της τεχνολογίας που ασχολείται με τα Big Data. Επίσης ένα μεγάλο κομμάτι του είναι ο έλεγχος, η επικοινωνία και όλες οι εφαρμογές που έχουν να κάνουν με το Internet of Things στο προηγούμενο κεφάλαιο.

3.2 Edge Computing

3.2.1 Εισαγωγή

Το edge computing είναι μια φιλοσοφία δικτύωσης που επικεντρώνεται στο να φέρνει τους υπολογισμούς και την αποθήκευση των πόρων όσο κοντά γίνεται στην πηγή που συλλέγονται τα δεδομένα και όχι σε μια κεντρική τοποθεσία που βρίσκεται μακριά από τον χρήστη. Με απλά λόγια αυτό σημαίνει μικρότερη εκτέλεση εργασιών στο νέφος (cloud) και μεταφορά τους σε τοπικά μηχανήματα όπως οι υπολογιστές χρηστών, συσκευές IoT και edge servers. Αυτό πραγματοποιείται προκειμένου τα δεδομένα και πρωτίστως τα real-time δεδομένα να μην αντιμετωπίζουν θέματα καθυστέρησης μεταφοράς (latency) που επηρεάζουν την απόδοση της εκάστοτε εφαρμογής καθώς και μην απαιτείται μεγάλο εύρος ζώνης (bandwidth) για την αποστολή δεδομένων από και προς τον χρήστη [16].

Που βρίσκεται όμως αυτό το edge ;



Σχήμα 3.4: Περιοχή Edge

Για συσκευές που χρησιμοποιούν το Internet, το edge του δικτύου είναι το σημείο το οποίο η συσκευή ή το τοπικό δίκτυο συνδέονται στο Internet. Στο σημείο αυτό δεν μπορούν να μπουν αυστηρά όρια καθώς διαφέρει από περίπτωση σε περίπτωση, το σημαντικό όμως εδώ είναι ότι το edge βρίσκεται γεωγραφικά κοντά στην συσκευή, όχι όπως servers του νέφους (cloud) οι οποίοι μπορούν να να βρίσκονται πολύ μακριά από αυτήν [45].

3.2.2 Πλεονεκτήματα & Περιορισμοί

Πλεονεκτήματα [16, 42]

Μειωμένη καθυστέρηση μεταφοράς: Κάθε φορά που μια συσκευή θέλει να επικοινωνήσει με έναν μακρινό server δημιουργείται μία καθυστέρηση. Χρησιμοποιώντας edge computing τεχνικές καταφέρνουμε αυτήν την καθυστέρηση να την μειώσουμε σημαντικά. Για παράδειγμα, σε μια εταιρεία για την επικοινωνία δύο τμημάτων στον ίδιο χώρο δεν χρειάζεται να μεταβάλλεται κάποια cloud υπηρεσία. Αν κάποιος θέλει να στείλει ένα μήνυμα αντι αυτό να προωθείτε σε έναν εξωτερικό server και έπειτα πίσω στο παραλήπτη μπορεί αυτό να γίνεται κατευθείαν μέσα στο δίκτυο της εταιρείας από το router της εταιρείας. Αυτές οι καθυστερήσεις παρατηρούνται γενικά σε web εφαρμογές και διαδικασίες που εκτελούνται σε εξωτερικούς servers και επηρεάζονται από το bandwidth του δικτύου και την τοποθεσία του server. Αυτές οι καθυστερήσεις μπορούν να μειωθούν σημαντικά με την μεταφορά περισσότερων "εργασιών" στο edge.

Μείωση στο εύρος ζώνης και στους απαραίτητους πόρους του server: Με την εξέλιξη της τεχνολογίας όλο και περισσότερες συσκευές στο κόσμο χρησιμοποιούν το IoT. Συνεπώς, πρέπει οι υπολογισμοί και τα δεδομένα να εκτελούνται και να αποθηκεύονται τοπικά έτσι ώστε να μην επιβαρύνεται πολύ το cloud και να μπορεί να εξυπηρετεί τα πραγματικά απαραίτητα αιτήματα χωρίς το κόστος της υπηρεσίας να αγγίζει απαγορευτικές τιμές.

Επιπλέον λειτουργίες: Με το edge μπορούμε επίσης να αναπτύξουμε λειτουργίες οι οποίες δεν ήταν υλοποιήσιμες μέχρι στιγμής. Όπως για παράδειγμα, η real-time επεξεργασία, αποθήκευση και ανάλυση δεδομένων.

Περιορισμοί [16]

Ανάγκη για περισσότερο και καλύτερο Hardware: Με την εξέλιξη του edge computing μεγαλώνει και η ανάγκη για περισσότερο και πιο εξειδικευμένο τοπικό hardware για να μπορούμε να καλύπτουμε τις ανάγκες των εφαρμογών. Πρόβλημα που εν μέρει δεν είναι τόσο σημαντικό, αφού με την εξέλιξη της τεχνολογίας πλέον το hardware είναι φθηνότερο και επιτρέπει την δημιουργία οικονομικά προσιτών smart συσκευών.

Ιδιωτικότητα και Ασφάλεια: Το βασικό πρόβλημα της τεχνολογίας βρίσκεται στην ασφάλεια. Όλο και περισσότερες συσκευες αποκτούν πρόσβαση στο δίκτυο με αποτέλεσμα να παρατηρείται πιο συχνά το φαινόμενο επιθέσεων και προσπάθειας υποκλοπών. Ακόμα και αν έχουν γίνει σημαντικές βελτιώσεις από τους προμηθευτές στην αναχαίτιση των απειλών που προκύπτουν από εξωτερικούς χρήστες ο διάλογος στις μέρες μας εκτυλίσσεται γύρω από τις ελευθερίες και πληροφορίες που παραδίδει ο χρήστης εκούσια στον πάροχο αυτών των υπηρεσιών. Περνάμε, λοιπόν, σε μία εποχή που η ιδιωτικότητα του κάθε χρήστη εμπίπτει στην καλή θέληση και εχεμύθεια του παρόχου. Παρόλου που υπάρχει σχετική προσπάθεια νομοθετικά να λυθούν τέτοια προβλήματα το edge μας μεταφέρει με γρήγορα, μη αναστρέψιμα βήματα στο μέλλον.

3.2.3 Σύγκριση Cloud Computing και Edge Computing

Σημειώνεται ότι το edge και το cloud computing δεν δημιουργήθηκαν για να αντικαταστήσουν το ένα το άλλο, το καθένα έχει τα δυνατά του σημεία και τα σημεία στα οποία αντιμετωπίζει δυσκολίες το θέμα είναι να διαλέξει ο κάθε χρήστης ποιό ταιριάζει καλύτερα στις ανάγκες του και να το εφαρμόσει ή ακόμα καλύτερα να βρει έναν τρόπο να συνδυάσει τα δυνατά στοιχεία της κάθε τεχνολογίας έτσι ώστε να βρει την καλύτερη απόδοση με βάση το κόστος. Δουλεύοντας μαζί edge και cloud μπορούν να δημιουργήσουν δομές με πολλή γρήγορη αποκρισιμότητα και μεγάλες δυνατότητες αποθήκευσης και επεξεργασίας. Σαν βασική διαφορά θα λέγαμε ότι το edge χρησιμοποιείται για χρονικά ευαίσθητες (time sensitive) εφαρμογές ενώ το cloud χρησιμοποιείται για να καλύψει ανάγκες στην ασφάλεια και στα Big Data [31].

Κεφάλαιο 4

Πρωτόκολλο MQTT

4.1 Εισαγωγή

Στο διαδίκτυο οι συσκευές αλληλεπιδρούν μεταξύ τους μέσω διαφόρων διεπαφών και πρωτοκόλλων επικοινωνίας (IoT). Για την επικοινωνία μεταξύ τους, οι συσκευές μπορούν να χρησιμοποιούν διάφορα βιομηχανικά πρωτόκολλα. Για αυτό το σκοπό, το MQTT είναι ένα από τα πιο δημοφιλή.

MQTT ή Queue Telemetry Transport είναι ένα ελαφρύ συμπαγές πρωτόκολλο ανταλλαγής μηνυμάτων για την μεταφορά δεδομένων σε απομακρυσμένες τοποθεσίες όπου απαιτείται ¨αποτύπωμα μικρού κώδικα' ή το εύρος ζώνης του δικτύου είναι περιορισμένο. Αυτά τα πλεονεκτήματα επιτρέπουν την εφαρμογή αυτού του πρωτοκόλλου στα συστήματα M2M (Machine to Machine) και IoT (Internet of Things).

Σχήμα 4.1: MQTT

Το MQTT σε τίτλους: [19]

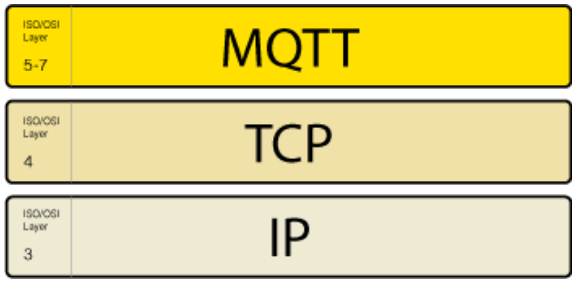
- Πρωτόκολλο ανταλλαγής μηνυμάτων του IoT
- Ελάχιστη επιβάρυνση (Minimal overhead)
- Publish / Subscribe
- Εύκολο

- Δυαδικό
- Αγνωστικό (Data Agnostic): Δεν γνωρίζει ή δεν νοιάζεται με ποιον τρόπο τα δεδομένα που λαμβάνει αποστέλλονται σε αυτό
- Σχεδιασμένο για αξιόπιστη για επικοινωνία πάνω σε αναξιόπιστα κανάλια

Βασικά χαρακτηριστικά του πρωτοκόλλου MQTT

- Ασύγχρονο πρωτόκολλο
- Συμπαγή μηνύματα
- Λειτουργία σε συνθήκες ασταθούς σύνδεσης της γραμμής μετάδοσης των δεδομένων
- Υποστήριξη διαφόρων επιπέδων ποιότητας υπηρεσιών (QoS)
- Εύκολη ενσωμάτωση νέων συσκευών

Στο επίπεδο της εφαρμογής, το πρωτόκολλο MQTT λειτουργεί πάνω από το πρωτόκολλο TCP / IP και χρησιμοποιεί την θύρα 1883 (ή 8883 εάν συνδέεται μέσω SSL) ως προεπιλογή [30].



Σχήμα 4.2: Στρώμα λειτουργίας MQTT

4.2 Ανταλλαγή μηνυμάτων στο MQTT

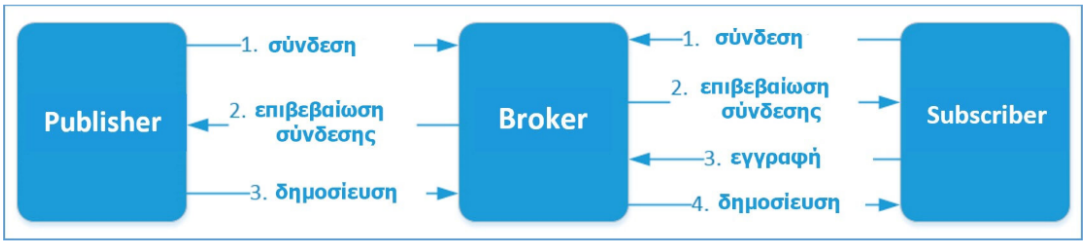
Στο πρωτόκολλο MQTT, η ανταλλαγή των μηνυμάτων πραγματοποιείται μεταξύ του Client, που μπορεί να είναι Publisher ή Subscriber των μηνυμάτων και του Broker των μηνυμάτων (π.χ. Mosquitto MQTT) [30]

Ο MQTT broker είναι πρακτικά ένα λογισμικό το οποίο εκτελείται τοπικά ή στο cloud. Ο broker λειτουργεί σαν ένα ταχυδρομείο, αφού κάνει την διαχείριση όλων των μηνυμάτων μεταξύ Publishers και Subscribers. Η διαχείριση αυτή πραγματοποιείται χάρη στην ύπαρξη των Topics. Ο Publisher γράφει το μήνυμα κάτω από το συγκεκριμένο Topic και όποιος επιθυμεί να δέχεται μηνύματα για που το αφορούν καλείται να κάνει Subscribe σε αυτό. Πολλοί Clients μπορεί να λαμβάνουν μηνύματα από έναν Publisher (ονε το μανψ σαπαβιλιτψ) και ένας subscriber μπορεί να λαμβάνει μηνύματα από πολλούς Publishers (many to one).

Κάθε Client μπορεί να είναι ταυτόχρονα Publisher και Subscriber, πράγμα που βοηθά στην διαχείρισει και τον έλεγχο των συσκευών και την κοινοποίηση δεδομένων.

Οι συσκευές MQTT χρησιμοποιούν ορισμένους τύπους μηνυμάτων για την επικοινωνία τους με τον Broker. Εδώ είναι οι βασικοί τύποι:

- Connect - δημιουργία σύνδεσης με το Broker μηνυμάτων.
- Disconnect - διακοπή σύνδεσης με το Broker μηνυμάτων.
- Publish - δημοσίευση δεδομένων σχετικά με κάποιο θέμα (Topic) μέσα στο Broker μηνυμάτων.
- Subscribe - εγγραφή σε ένα θέμα (Topic) στο Broker μηνυμάτων.
- Unsubscribe - διαγραφή του θέματος (Topic).

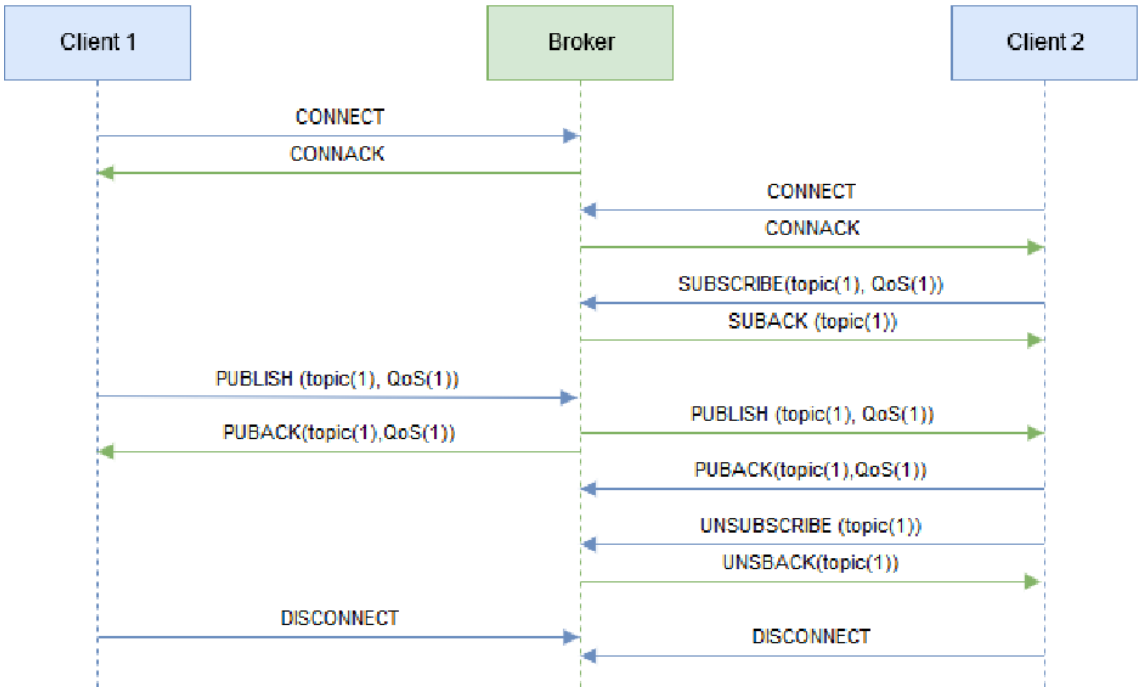


Σχήμα 4.3: Ανταλλαγή μηνυμάτων στο MQTT

Πλεονεκτήματα της μεθόδου Publish/Subscribe [19]

- Αποσύζευξη χώρου (Space Decoupling)
- Αποσύζευξη χρόνου (Time Decoupling)
- Αποσύζευξη συντονισμού (Synchronization Decoupling)
- Φιλτράρισμα μηνυμάτων με βάση τα θέματα (Topics)
- Επικοινωνία που παραδίδεται από έναν αποστολέα στους παραλήπτες. Μπορεί να υπάρχει επιβεβαίωση αποστολής, αλλά δεν υπάρχει εγγύηση ότι το μήνυμα λήφθηκε και κατανοήθηκε (Push communication).
- Δυναμικά θέματα (Dynamic Topics)

Πιο αναλυτικά ένα message flow διάγραμμα που περιγράφει την παραπάνω διαδικασία με τα αντίστοιχα πακέτα που ανταλλάσσονται ανάμεσα στον Client και τον Broker είναι το παρακάτω (τα πακέτα του MQTT θα αναλυθούν εκτενώς παρακάτω). Ο Client μπορεί να είναι είτε Publisher είτε Subscriber είτε και τα δύο.



Σχήμα 4.4: Πακέτα μηνυμάτων ανάμεσα σε Client και Broker

4.3 Ποιότητα εξυπηρέτησης στο πρωτόκολλο MQTT (QoS)

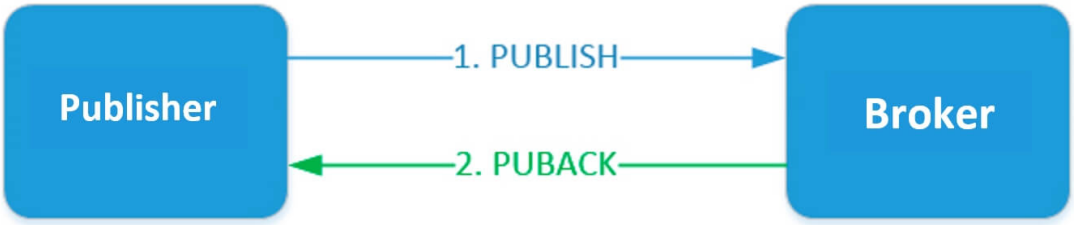
Το MQTT υποστηρίζει τρία επίπεδα ποιότητας εξυπηρέτησης Quality of Service (QoS) κατά την αποστολή των μηνυμάτων [30].

At most once (QoS = 0)
Ο Publisher στέλνει ένα μήνυμα στον Broker μία φορά και δεν ακολουθεί καμία επιπλέον ενέργεια (fire and forget).

Σχήμα 4.5: MQTT with QoS = 0

At least once (QoS = 1)
Αυτό το επίπεδο εγγυάται την παράδοση του μηνύματος στον Broker, ωστόσο είναι δυνατή η επανάληψη των μηνυμάτων από τον Publisher. Μόλις ληφθεί ένα αντίγραφο, ο Broker στέλνει αυτό το μήνυμα στους Subscribers ξανά και προωθεί την επιβεβαίωση της παραλαβής του μηνύματος στον Publisher. Εάν ο Publisher δεν λάβει το μήνυμα PUBACK από τον Broker,

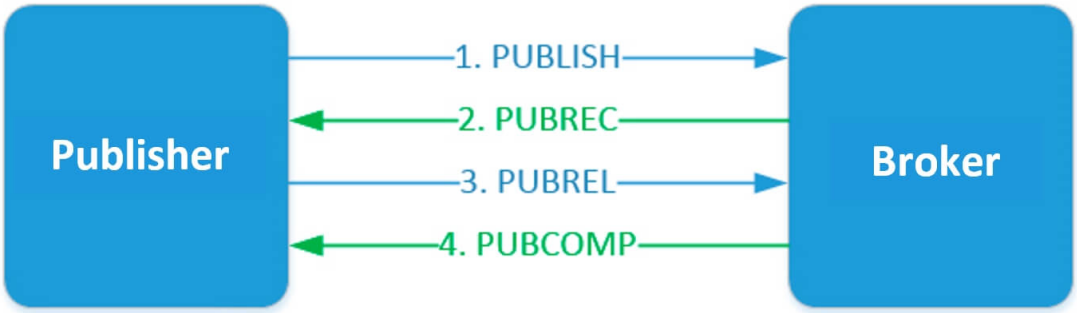
θα προσπαθήσει να επαναποστέλλει αυτό το πακέτο (acknowledged delivery).



Σχήμα 4.6: MQTT with QoS = 1

Exactly once (QoS = 2)

Σε αυτό το επίπεδο, η παράδοση του μηνύματος στον Subscriber είναι εγγυημένη, χωρίς την επαναποστολή αντιγράφων. Ο Publisher στέλνει ένα μήνυμα στον Broker με συγκεκριμένο Packet ID. Ο Publisher αποθηκεύει το μήνυμα που δεν έχει αναγνωριστεί εκτός αν λάβει απάντηση PUBREC από τον Broker. Ο Broker απαντά με το μήνυμα PUBREC που περιέχει το ίδιο Packet ID. Αφού λάβετε αυτό το μήνυμα, ο Publisher στέλνει το PUBREL με το ίδιο Packet ID. Ο Broker πρέπει να αποθηκεύσει το αντίγραφο του μηνύματος μέχρι να πάρει PUBREL. Μόλις ο Broker λάβει PUBREL, διαγράφει το αντίγραφο του μηνύματος και στέλνει στο Publisher το μήνυμα PUBCOMP για την ολοκλήρωση της συναλλαγής.



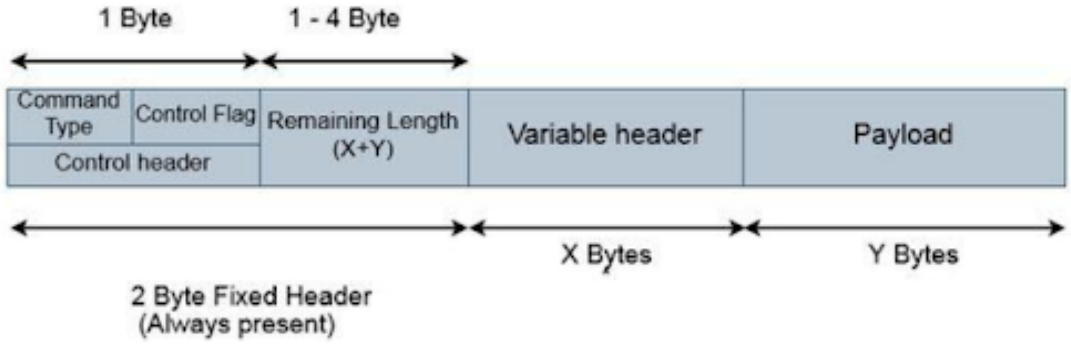
Σχήμα 4.7: MQTT with QoS = 2

4.4 Δομή του μηνύματος

Το μήνυμα MQTT αποτελείται από 3 βασικά κομμάτια:

1. Σταθερή κεφαλίδα (fixed header)
2. Μεταβλητή κεφαλίδα (variable header)
3. Δεδομένα, ωφέλιμο φορτίο (payload)

Η σταθερή επικεφαλίδα μήκους 2-byte συμπεριλαμβάνεται σε κάθε μήνυμα ενώ οι άλλες δύο δεν είναι πάντα παρούσες [3].



Σχήμα 4.8: Δομή του MQTT μηνύματος

4.4.1 Σταθερή κεφαλίδα (fixed header)

Η Σταθερή κεφαλίδα χωρίζεται επιμέρους σε [3]:

Command type
Από τα 2-bytes της σταθερής επικεφαλίδας το πρώτο byte ονομάζεται control field (control header). Αυτό με την σειρά του χωρίζεται σε δύο 4αδες από bits. Η πρώτη MSB τετράδα είναι το command type field και συμβολίζει τον τύπο μηνύματος, για παράδειγμα: CONNECT → 0001, SUBSCRIBE → 1000, PUBLISH → 0011, κλπ. και οι τιμές που παίρνει συμβολίζουν τα παρακάτω:

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

Σχήμα 4.9: Πίνακας τιμών Command type

Control Flag

Τα επόμενα 4 bits της σταθερής επικεφαλίδας χρησιμοποιούνται μόνο για την εντολή publish και για όλες τις άλλες εντολές έχουν συγκεκριμένη τιμή.

Control Packet	Fixed header flags	Bit 3	Bit 2	Bit 1	Bit 0
CONNECT	Reserved	0	0	0	0
CONNACK	Reserved	0	0	0	0
PUBLISH	Used in MQTT 3.1.1	DUP[1]	QoS[2]	QoS[2]	RETAIN[3]
PUBACK	Reserved	0	0	0	0
PUBREC	Reserved	0	0	0	0
PUBREL	Reserved	0	0	1	0
PUBCOMP	Reserved	0	0	0	0
SUBSCRIBE	Reserved	0	0	1	0
SUBACK	Reserved	0	0	0	0
UNSUBSCRIBE	Reserved	0	0	1	0
UNSUBACK	Reserved	0	0	0	0
PINGREQ	Reserved	0	0	0	0
PINGRESP	Reserved	0	0	0	0
DISCONNECT	Reserved	0	0	0	0

Σχήμα 4.10: Πίνακας τιμών Control Flag

DUP – Το αντίγραφο ορίζεται όταν ένας client ή Broker MQTT πραγματοποιεί την επαναποστολή του πακέτου. Εάν έχει οριστεί flag, η μεταβλητή κεφαλίδα πρέπει να περιέχει Message ID (αναγνωριστικό μηνύματος).
QoS – Ποιότητα Υπηρεσιών (0,1,2)
RETAIN – κατά τη δημοσίευση των δεδομένων με ενδείκτη Retain, ο Broker θα το αποθη-

κεύσει. Όταν δημιουργηθεί μια νέα συνδρομή σε αυτό το θέμα, ο Broker θα στείλει αμέσως ένα μήνυμα με αυτό το flag. Χρησιμοποιείται μόνο σε μηνύματα τύπου PUBLISH.

Remaining Length

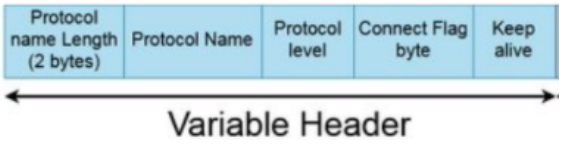
Το δεύτερο byte της σταθερής επικεφαλίδας περιέχει το remaining length το οποίο υπολογίζεται ως μήκος variable header + μήκος payload. Το μήκος αυτό μπορεί να είναι εώς 4 βψτες καθένα από τα οποία περιέχει 7 βιτς για το μήκος και 1 bit (το MSB) είναι το continuation flag. Αν το continuation flag bit είναι 1 τότε σημαίνει ότι το επόμενο byte είναι και αυτό μέρος του remaining length, αν το continuation flag bit είναι 0 τότε σημαίνει ότι αυτό το byte είναι το τελευταίο του remaining length.

4.4.2 Μεταβλητή κεφαλίδα (variable header)

Η μεταβλητή επικεφαλίδα δεν είναι παρούσα σε όλα τα MQTT πακέτα. Κάποιες εντολές ή μηνύματα την χρησιμοποιούν για να μεταφέρουν επιπλέον πληροφορίες ή flags σε αυτή τη θέση. [36] Θα αναλύσουμε την επικεφαλίδα του Connect πακέτου, παρακάτω όμως θα αναλύσουμε όλων των διαφορετικών πακέτων.

Θα αναλύσουμε την επικεφαλίδα του Connect πακέτου, παρακάτω όμως θα αναλύσουμε όλων των διαφορετικών πακέτων. Η variable header του Connect πακέτου αποτελείται από 4 στοιχεία:

1. Protocol name length μαζί με Protocol name
2. Protocol level
3. Connect flags
4. Keep Alive



Σχήμα 4.11: Variable Header

1. Protocol name length και Protocol Name

Η τιμή του υποδηλώνει το μήκος του ονόματος του πρωτοκόλλου που χρησιμοποιείται και το όνομά του. Το MQTT έχει μήκος 4 συνεπώς.

	description	7	6	5	4	3	2	1	0
Protocol Name									
Byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
Byte2	Length LSB (4)	0	0	0	0	0	1	0	0
Byte3	M	0	1	0	0	1	1	0	1
Byte4	Q	0	1	0	1	0	0	0	1
Byte5	T	0	1	0	1	0	1	0	0
Byte6	T	0	1	0	1	0	1	0	0

Σχήμα 4.12: Protocol Name

2. Protocol level

Για το MQTT το protocol level έχει την τιμή 4 οποιαδήποτε άλλη τιμή σε αυτή τη θέση θα οδηγήσει σε αποσύνδεση.

Protocol Level									
Byte 7	Level(4)	0	0	0	0	0	1	0	0

Σχήμα 4.13: Protocol level

3. Connect Flags

Οι διάφορες τιμές το κάθε ενδείκτη φανερώνουν την ύπαρξη ή την απουσία συγκεκριμένων πληροφοριών στο ωφέλιμο φορτίο (payload). [30]

Bit	7	6	5	4	3	2	1	0
	User name flag	Password flag	Will retain	Will qos		Will flag	clean session	reserve d
Byte 8	x	x	x	x	x	x	x	X

Σχήμα 4.14: Connect Flags

User name – αν ο ενδείκτης έχει οριστεί, θα πρέπει να υπάρχει ένα όνομα χρήστη στο ωφέλιμο φορτίο (χρησιμοποιείται για λόγους επαλήθευσης του Client)
Password – αν ο ενδείκτης έχει οριστεί, πρέπει να υπάρχει ένας κωδικός πρόσβασης στο ωφέλιμο φορτίο (χρησιμοποιείται για λόγους επαλήθευσης του Client).
Will Retain - εάν ο ενδείκτης έχει οριστεί σε 1, ο Broker αποθηκεύει ένα μήνυμα Ωιλλ.
Will QoS - Ποιότητα εξυπηρέτησης για το μήνυμα Will. Εάν έχει οριστεί η σημαία Will, θα πρέπει να είναι παρόντες οι Will QoS και Will.
Will Flag - εάν ο ενδείκτης έχει οριστεί, τότε ο Πελάτης αποσυνδέει τον Broker χωρίς να αποστείλει την εντολή DISCONNECT (σε περίπτωση απρόβλεπτου τερματισμού λειτουργίας, σφάλματος κλπ.), ο Broker θα ενημερώσει όλους τους συνδεδεμένους Clients μέσω του αποκαλούμενου Will Message.
Clean Session - εάν αν ο ενδείκτης έχει οριστεί σε 0, ο Broker αποθηκεύει μια συνεδρία, όλες τις συνδρομές του Client και, με την επόμενη σύνδεση ο ενδείκτης, θα στείλει όλα τα μηνύματα από το QoS = 1 και QoS = 2 που ελήφθησαν από τον Broker ενώ αποσυνδέθηκε ένας Client. Συνεπώς, εάν ο Client έχει οριστεί σε 1, τότε από την επόμενη σύνδεση, ο Client θα πρέπει να εγγραφεί ξανά σε όλα τα θέματα.

4. Keep Alive
Η τιμή του υποδηλώνει την μέγιστη διάρκεια που είναι επιτρεπτό να παρέλθει από την στιγμή που ο Client τελειώνει την μετάδοση ενός Control πακέτου και την στιγμή που ξεκινάει να στέλνει το επόμενο.

Bit	7	6	5	4	3	2	1	0
Byte 9	Keep Alive MSB							
Byte 10	Keep Alive LSB							

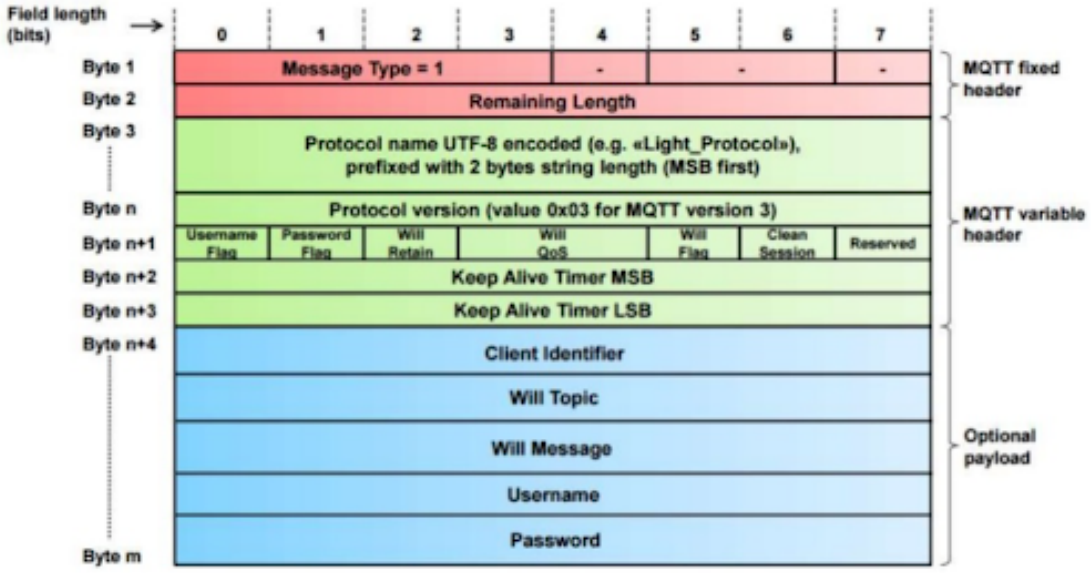
Σχήμα 4.15: Keep Alive

4.4.3 Δεδομένα, ωφέλιμο φορτίο (payload)

Στο τέλος το πακέτο μπορεί να περιέχει κάποιο payload είναι διαφορετικό ανάλογα με τον τύπο του πακέτου, συνήθως εδώ μεταφέρονται τα δεδομένα που θέλουμε να στείλουμε. Παρακάτω στα παραδείγματα θα δούμε πως μοιάζει το Payload για κάθε είδος πακέτου [3].

4.5 Παραδείγματα πακέτων [28]

4.5.1 CONNECT (Client request to connect to server)



Σχήμα 4.16: Πακέτο CONNECT

Τα πεδία fixed header και variable header έχουν αναλυθεί παραπάνω στο συγκεκριμένο πακέτο το payload αποτελείται από:

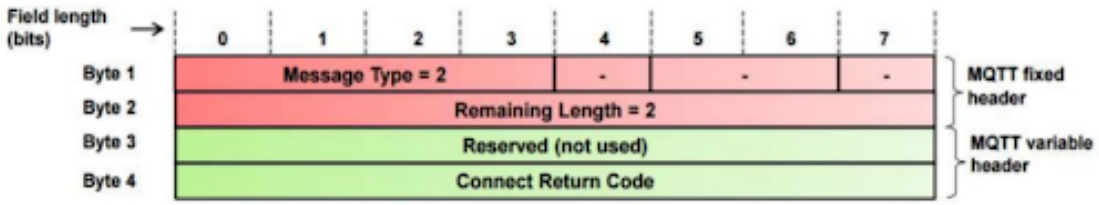
Client Identifier - Είναι μεταξύ σε 1 - 23 χαρακτήρες και είναι μοναδικό για κάθε Client, με αυτό γίνεται η αναγνώριση του Client από τον server.

Will Topic - Το Will topic στο οποίο θα δημοσιευθεί το will message (αν το will flag στο variable header είναι ενεργό).

Will Message - Το Will message το οποίο θα δημοσιευθεί (αν το will flag στο variable header είναι ενεργό).

Username και Password - Username και Password (αν τα αντίστοιχα flags στο variable header είναι ενεργά).

4.5.2 CONNACK (Connect Acknowledgement)



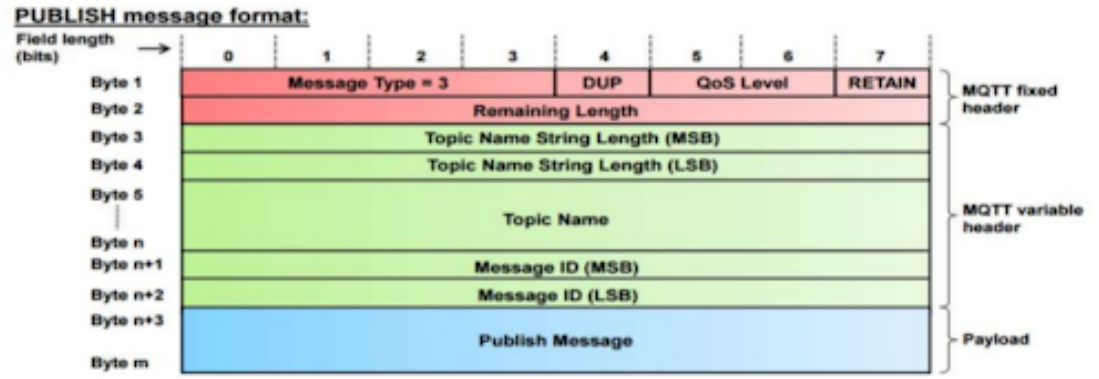
Σχήμα 4.17: Πακέτο CONNACK

Το πεδίο fixed header έχει αναλυθεί παραπάνω στο συγκεκριμένο πακέτο όμως το variable header αποτελείται από τα Connect Return Codes:

Return Code	Return Code Response
0	Connection Accepted
1	Connection Refused, unacceptable protocol version
2	Connection Refused, identifier rejected
3	Connection Refused, server unavailable
4	Connection Refused, bad credentials
5	Connection Refused, not authorized

Σχήμα 4.18: Return Codes

4.5.3 PUBLISH (Publish Message)



Σχήμα 4.19: Πακέτο PUBLISH

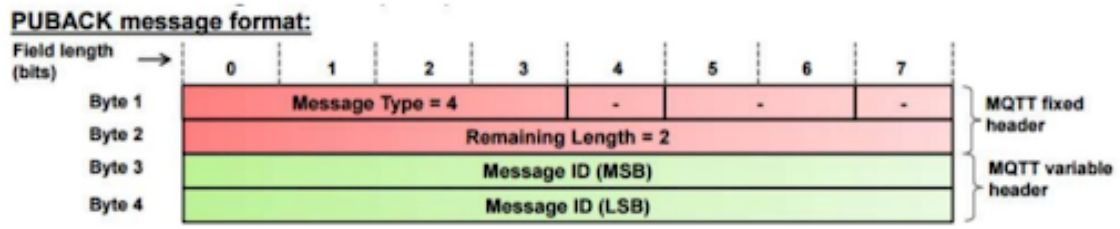
Το πεδίο fixed header έχει αναλυθεί παραπάνω στο συγκεκριμένο πακέτο όμως το variable header αποτελείται από:

Topic Name και Topic String Length - περιέχει το topic Name στο οποίο θα δημοσιευθεί το μήνυμα. Τα πρώτα 2 bytes περιέχουν το μήκος αυτού του topic name.

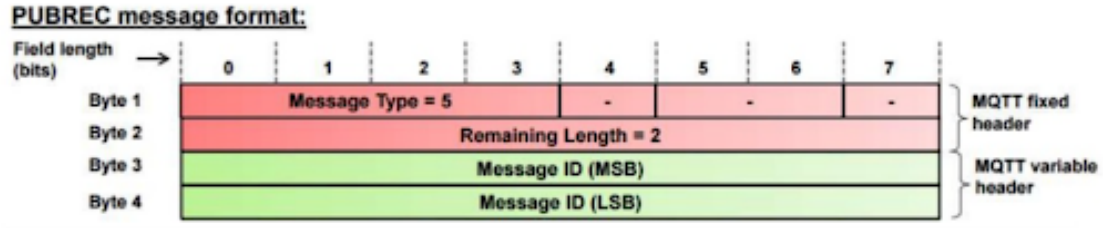
Message ID - Είναι παρόν μόνο όταν έχουμε QoS = 1 (at least once delivery), QoS = 2 (exactly once delivery).

Και το payload αποτελείται από:
Publish Message - Το μήνυμα ως ένας πίνακας από bytes, η δομή του μηνύματος παρουσιάζει διαφορές ανάλογα με την εφαρμογή.

4.5.4 PUBACK και PUBREC (Publish Acknowledgement και Publish Received)



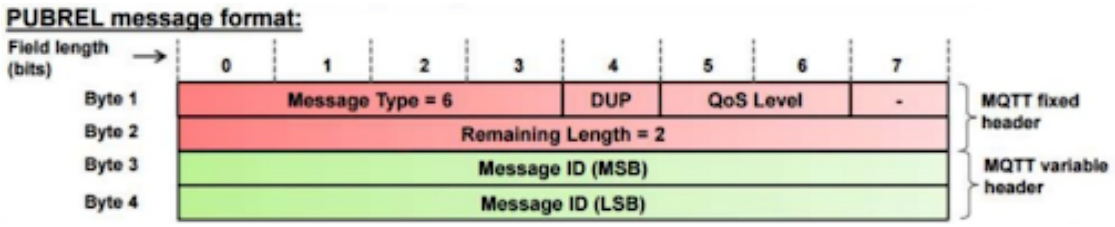
Σχήμα 4.20: Πακέτο PUBACK



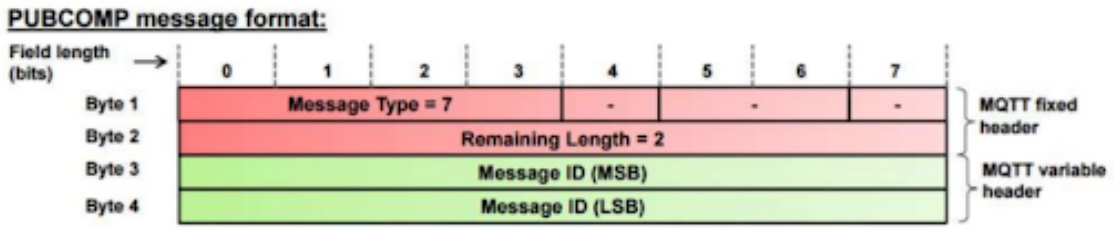
Σχήμα 4.21: Πακέτο PUBREC

Και στα δύο πακέτα τα πεδία fixed header έχουν αναλυθεί παραπάνω όμως τα variable headers αποτελούνται από:
Message ID - Περιέχει το message ID του πακέτου που χρειάζεται αναγνώριση (acknowledgement).
Το payload απουσιάζει.

4.5.5 PUBREL και PUBCOMP (Publish Release και Publish Complete)



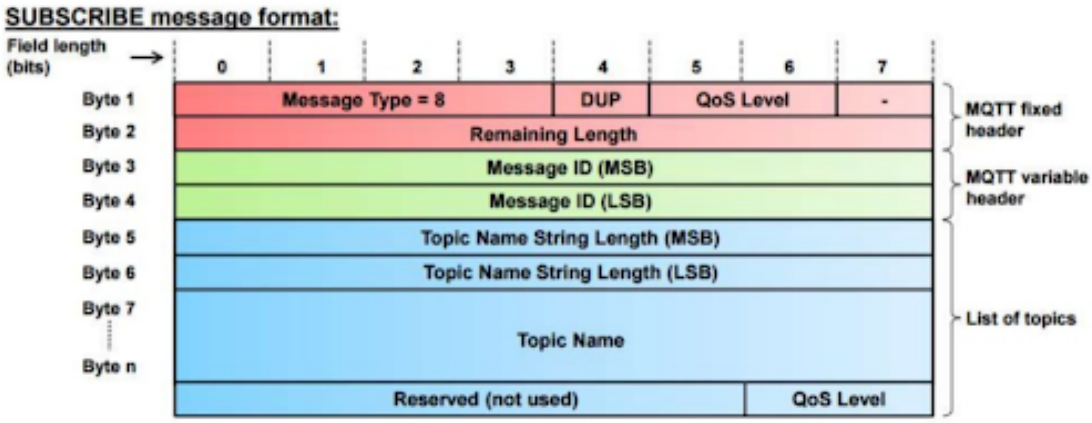
Σχήμα 4.22: Πακέτο PUBREL



Σχήμα 4.23: Πακέτο PUBCOMP

Και στα δύο πακέτα τα πεδία fixed header έχουν αναλυθεί παραπάνω όμως τα variable headers αποτελούνται από:
Message ID - Περιέχει το message ID του πακέτου που χρειάζεται αναγνώριση (acknowledgement).
Το payload απουσιάζει.

4.5.6 SUBSCRIBE (Client Subscribe Request)

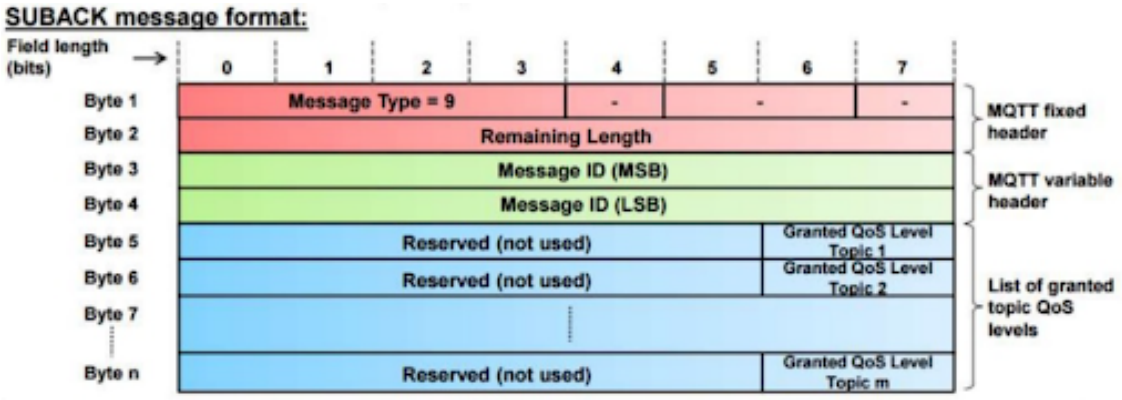


Σχήμα 4.24: Πακέτο SUBSCRIBE

Το πεδίο fixed header έχει αναλυθεί παραπάνω στο συγκεκριμένο πακέτο όμως το variable header αποτελείται από:
Message ID - Είναι παρόν μόνο όταν έχουμε QoS = 1 (at least once delivery), QoS = 2 (exactly once delivery). Χρησιμοποιείται για την αναγνώριση (acknowledgement) του subscribe μηνύματος.
Και το payload αποτελείται από:
Topic Name και Topic String Length - περιέχει το topic Name στο οποίο ο Client θέλει να κάνει subscribe. Τα πρώτα 2 bytes περιέχουν το μήκος αυτού του topic name.
QoS Level- Την ποιότητα εξυπηρέτησης με την οποία ο Client θέλει να ενημερώνεται από κάθε topic.

Στο συγκεκριμένο σημείο (payload) μπορεί να βρίσκονται πολλά topic names με τα αντίστοιχα QoS levels.

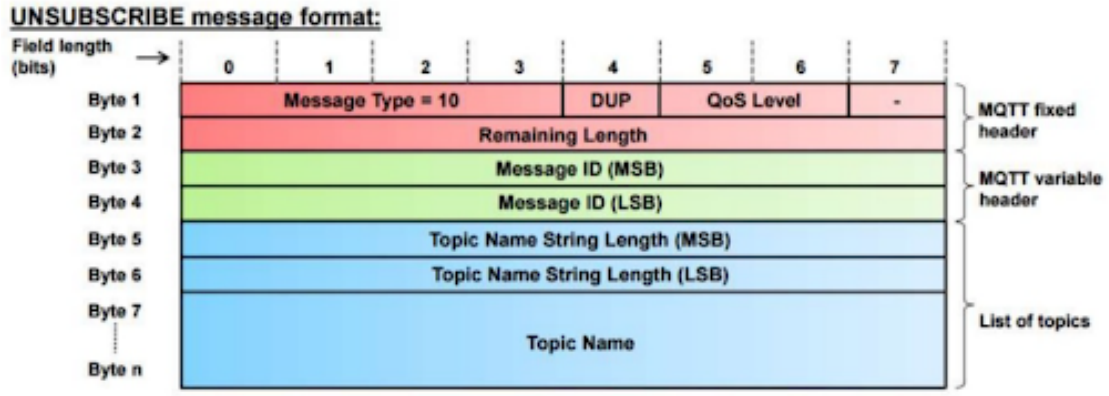
4.5.7 SUBACK (Subscribe Acknowledgement)



Σχήμα 4.25: Πακέτο SUBACK

Το πεδίο fixed header έχει αναλυθεί παραπάνω στο συγκεκριμένο πακέτο όμως το variable header αποτελείται από:
Message ID - Περιέχει το message ID του πακέτου που χρειάζεται αναγνώριση (acknowledgement).
Και το παψλοαδ αποτελείται από:
Granted QoS Level for Topic - περιέχει μια λίστα από τα granted QoS levels για τα topics που ζήτησε να κάνει subscribe ο client με το πακέτο SUBSCRIBE που έστειλε.
Στο συγκεκριμένο σημείο (payload)μπορεί να βρίσκονται πολλά granted topic QoS levels ανάλογα με το σε πόσα topics ζήτησε να κάνει subscribe ο client.

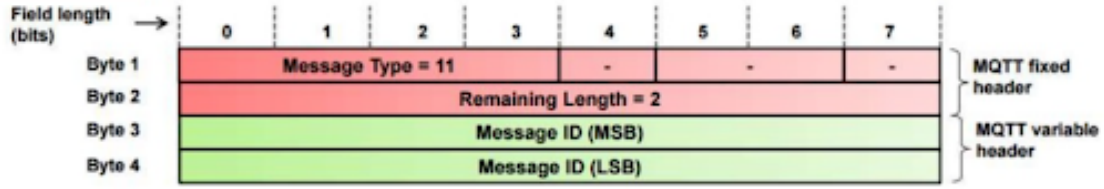
4.5.8 UNSUBSCRIBE (Client Unsubscribe Request)



Σχήμα 4.26: Πακέτο UNSUBSCRIBE

Το πεδίο fixed header έχει αναλυθεί παραπάνω στο συγκεκριμένο πακέτο όμως το variable header αποτελείται από:
Message ID - Χρησιμοποιείται για την αναγνώριση (acknowledgement) του συβσςριβε μηνύματος. Τα unsubscribe μηνύματα έχουν QoS=1 (at least once delivery).
Και το payload αποτελείται από:
Topic Name και Topic String Length - περιέχει το topic Name από το οποίο ο Client θέλει να κάνει unsubscribe. Τα πρώτα 2 bytes περιέχουν το μήκος αυτού του topic name. Στο συγκεκριμένο σημείο (payload) μπορεί να βρίσκονται πολλά topic names.

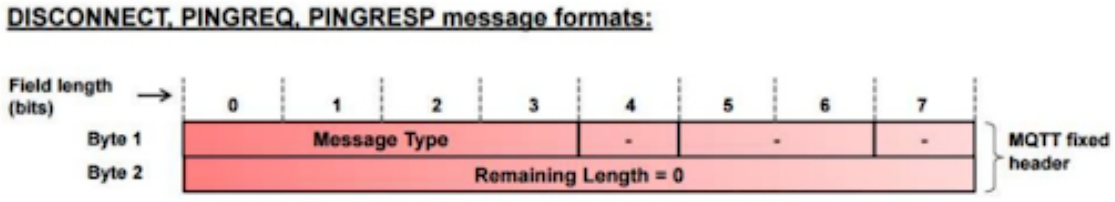
4.5.9 UNSUBACK(Unsubscribe Acknowledgement)



Σχήμα 4.27: Πακέτο UNSUBACK

Το πεδίο fixed header έχει αναλυθεί παραπάνω όμως το variable header αποτελείται από:
Message ID - Περιέχει το message ID του πακέτου που χρειάζεται αναγνώριση (acknowledgement).
Το payload απουσιάζει.

4.5.10 PINGREC, PINGREST και DISCONNECT (PING request, PING response, Client is Disconnecting)



Σχήμα 4.28: Πακέτο PINGREC, PINGREST, DISCONNECT

Περιέχει μόνο το πεδίο fixed header που έχει αναλυθεί παραπάνω.

Κεφάλαιο 5

Apache Kafka

5.1 Εισαγωγή

Το Apache Kafka είναι ένα open-source σύστημα ανταλλαγής μηνυμάτων το οποίο βασίζεται στην τεχνική publish-subscribe. Αρχικά αναπτύχθηκε από το LinkedIn και στην συνέχεια έγινε open-source από το Apache στις αρχές του 2011 [6]. Πήρε το όνομά του από τον σπουδαίο λογοτέχνη του 20ού αιώνα, Franz Kafka.

Σχήμα 5.1: Apache Kafka Logo

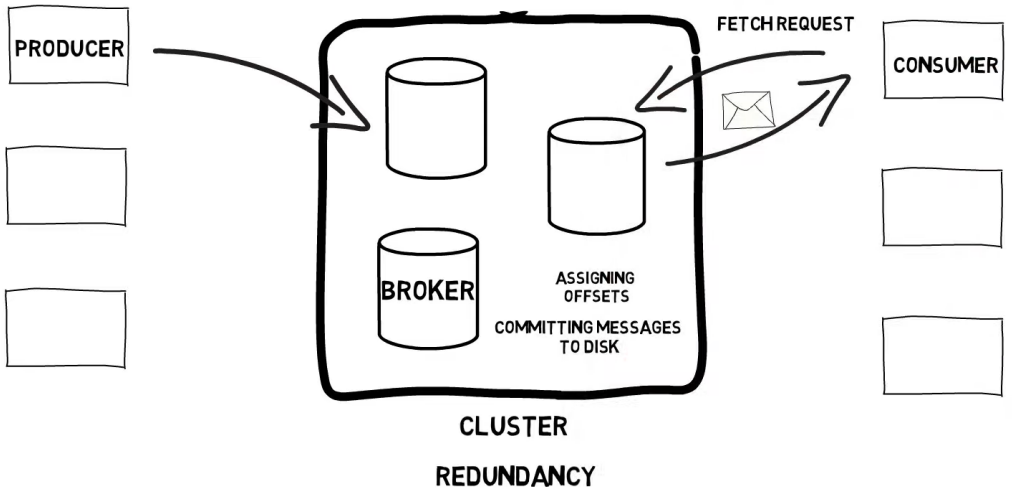
Σύμφωνα με αυτήν την τεχνική ο αποστολέας (publisher) δεν στέλνει δεδομένα κατευθείαν στον παραλήπτη (subscriber), αλλά τα χωρίζει με ετικέτες ανάλογα με το θέμα τους και τα κοινοποιεί δίχως να ξέρει αν υπάρχουν ενδιαφερόμενοι (subscribers) οι οποίοι θα τα παραλάβουν. Με τον ίδιο τρόπο ένας ενδιαφερόμενος παραλήπτης (subscriber) κάνει subscribe για ένα θέμα μηνυμάτων που θα ήθελε να λαμβάνει δίχως να γνωρίζει αν σε αυτό το θέμα υπάρχει κάποιος ο οποίος αποστέλλει δεδομένα. Από την περιγραφή του συστήματος προκύπτει ότι υπάρχει ένα κεντρικό σύστημα (broker) στο οποίο αποστέλλονται και αρχειοθετούνται τα μηνύματα ανάλογα με το θέμα (topic). Αυτό το σύστημα είναι υπεύθυνο να παραλαμβάνει τα μηνύματα διαφορετικών θεμάτων και να τα διαβιβάζει στους ενδιαφερόμενους χωρίς να συνδέει κάθε αποστολέα με τον παραλήπτη ξεχωριστά.

Πολλές φορές το Apache Kafka περιγράφεται και ως ένα κατανεμημένο αρχείο καταγραφής γεγονότων (distributed event log) το οποίο περιέχει αμετάβλητα μηνύματα τα οποία μάλιστα αποθηκεύονται και προσωρινά στον δίσκο με βάση την πολιτική που ακολουθεί ο broker το οποίο είναι και η βασική διαφορά του Kafka με άλλα παρόμοια συστήματα ανταλλαγής μηνυμάτων [6].

Παρουσιάζεται συχνά σαν υβρίδιο ενός συστήματος ανταλλαγής μηνυμάτων και μιας βάσης δεδομένων και σκοπός του είναι ο χειρισμός real-time ροής δεδομένων και data pipelines αξιόπιστα και με μεγάλη διεκπεραιωτική ταχύτητα. Καθώς παράλληλα προσφέρει ένα κοινό χώρο για αποθήκευση των δεδομένων και διαχείριση γεγονότων μειώνοντας το πρόβλημα της ενσωμάτωσης και ενοποίησης, εφόσον ότι χρειάζεται η εφαρμογή βρίσκεται σε κοινό σημείο. Χρησιμοποιείται ακόμα για δημιουργία ETL (Extract, Transform, Load), CDC (Change Data Capture) και Big Data Ingest συστημάτων.

5.2 Αρχιτεκτονική

Η αρχιτεκτονική του Apache Kafka παρουσιάζεται σχηματικά στην παρκάτω εικόνα:



Σχήμα 5.2: Αρχιτεκτονική Apache Kafka

Βασικά κομμάτια της αρχιτεκτονικής του Apache Kafka [6]:
Message: Το μήνυμα, για τον Kafka είναι ένα συγκεκριμένο στοιχείο δεδομένων το οποίο αποτελείται από έναν πίνακα με byte που είναι η τιμή του μηνύματος και ένα optional key το οποίο βοηθάει στην οργάνωση των μηνυμάτων στα partitions.
Producer: Ο αποστολέας των μηνυμάτων. Συνήθως είναι άλλες εφαρμογές που παράγουν δεδομένα και τα αποστέλλουν στους servers. Για παράδειγμα μια εφαρμογή καταγραφης της θερμοκρασίας σε ένα κτίριο ή ένας αισθητήρας κίνησης σε ένα σπίτι.
Consumer: Ο παραλήπτης. Συνήθως είναι άλλες εφαρμογές που δέχονται δεδομένα και είναι υπεύθυνες για την αποθήκευση τους, την επεξεργασία τους ή απλά την μεταβίβασή τους. Για παράδειγμα, μια εφαρμογή πυρόσβεσης σε ένα κτίριο που λαμβάνει δεδομένα για την θερμοκρασία ή μια απλή βάση δεδομένων.
Cluster: Λειτουργεί ως ένα κεντρικός κόμβος είναι το περιβάλλον που βρίσκονται οι Kafka servers (brokers). Συνήθως ένα cluster περιέχει τουλάχιστον 3 brokers για να υπάρχουν

αντίγραφα ασφαλείας και πλεόνασμα πληροφορίας.
Broker: Ένας Kafka server μέσα στο cluster. Είναι υπεύθυνος για την ανάθεση σειράς στα μηνύματα, για την εισαγωγή (commit) των μηνυμάτων στο δίσκο καθώς και την απάντηση στα ερωτήματα (fetch requests) των consumers διακινώντας τα μηνύματα.

Ανάλυση του consumer

Κάθε consumer είναι υπεύθυνος να θυμάται το offset του τελευταίου μηνύματος που διάβασε από ένα συγκεκριμένο Topic έτσι ώστε να μην υπάρχει απώλεια μηνυμάτων. Συχνά οι consumers δεν είναι μοναδικά στοιχεία αλλά ανήκουν σε συγκεκριμένα consumer groups τα οποία είναι υπεύθυνα για την "κατανάλωση" των μηνυμάτων ενός Topic. Λειτουργούν ομαδικά και κάθε partition του topic διαβάζεται αυστηρά από έναν και μόνο consumer του group. Παρόλα αυτά ένας consumer μπορεί να είναι υπεύθυνος να διαβάζει από παραπάνω από ένα partition. Εκτός από ταχύτητα και κλιμακωσιμότητα η ύπαρξη consumer groups βοηθά στην διαχείριση λαθών και αποτυχίας κάποιου consumer. Καθώς σε περίπτωση προβλήματος οι ευθύνες για τα partitions διαμοιράζονατι ξανά στο group και δεν χάνονται μηνύματα. Κάθε μήνυμα διαβάζεται ακριβώς μια φορά από έναν consumer και consumer group συνεπώς στην περίπτωση που θέλουμε να διαβάσουμε παραπάνω φορές τα ίδια δεδομένα για διαφορετικές εφαρμογές πρέπει οι consumer να βρίσκονται σε διαφορετικά groups [6].

Ανάλυση των brokers

Οι brokers είναι προγραμματισμένοι να λειτουργούν μέσα σε ένα cluster και διαχειρίζονται χιλιάδες partitions και εκατομμύρια μηνύματα το δευτερόλεπτο. Συνεπώς μέσα σε ένα cluster πρέπει να υπάρχει κάποια μορφή οργάνωσης και ιεραρχίας. Γι αυτό το λόγο ένας broker μέσα στο cluster ενεργεί ως ο διαχειριστής του cluster (cluster controller) και είναι υπεύθυνος για την ανάθεση των partitions στους brokers και την παρακολούθηση για τυχόν λάθη και αποτυχίες των brokers. Κάθε partition ανήκει σε έναν συγκεκριμένο broker μέσα στο cluster ο οποίος ονομάζεται partition leader για το συγκεκριμένο partition. Παρόλα αυτά υπάρχουν και σε άλλους brokers αντίγραφα του partition (replicas) προκειμένου να υπάρχει πλεόνασμα πληροφορίας και αντίγραφα ασφαλείας έτσι ώστε με κάθε αποτυχία ή αποσύνδεση του partition leader κάποιος άλλος broker στο cluster να είναι έτοιμος να πάρει την θέση του. Σε αυτήν την περίπτωση πρέπει όλες οι εφαρμογές (producers-consumers) του παλιού leader να συνδεθούν με τον νέο, γι αυτό το λόγο πρέπει ο καινούργιος leader να επιλεχθεί προσεκτικά για να παραμένει το φορτίο του cluster όσο το δυνατόν καλύτερα διαμοιρασμένο [6].

5.3 Apache Zookeeper

Βασικό κομμάτι της λειτουργίας του cluster είναι και ο Zookeeper. Ο Zookeeper είναι ένα top-level λογισμικό υλοποιημένο από την Apache, δρα ως ένα κέντρο ελέγχου για και διαχείρισης δεδομένων ονοματολογίας και ρυθμίσεων το οποίο παρέχει ευελιξία και συγχρονισμό του cluster μέσα σε κατανεμημένα συστήματα. Είναι υπεύθυνο για τον συνεχή έλεγχο της κατάσταση του συστήματος και κόμβων του cluster καθώς και την εποπτεία των topics

και των partitions. Ακόμα δίνει την δυνατότητα σε πολλούς clients να εκτελούν ταυτόχρονα read και write ενέργειες. Γενικά σημειώνεται ότι ο Kafka Server δεν μπορεί να εκτελεστεί χωρίς την εγκατάσταση του Zookeeper.

Λειτουργία του Zookeeper

Τα δεδομένα στον Zookeeper χωρίζονται σε διαφορετικές ομάδες κόμβων με σκοπό να υπάρχει μεγάλη διαθεσιμότητα, συνέπεια και συνοχή στα δεδομένα. Επιπλέον με τον χωρισμό των δεδομένων σε κόμβους δημιουργούμε ένα σύστημα το οποίο μπορεί ευκολότερα και γρηγορότερα να διαχειριστεί λάθη και προβλήματα. Αν για παράδειγμα, ένας master κόμβος (κόμβος που είναι υπεύθυνος για όλη την ομάδα) αντιμετωπίσει πρόβλημα και δεν είναι διαθέσιμος, αντικαθίσταται μέσω ψηφίσματος με έναν άλλον στην ομάδα με αποτέλεσμα ο τελικός χρήστης συνδεδεμένος στον καινούργιο κόμβο να μπορεί να συνεχίσει τα αιτήματα του στο σύστημα [21].

Λόγοι που ο Zookeeper είναι αναγκαίος: [21]
Ανάδειξη controller (Controller election): Ο controller είναι υπεύθυνος να διατηρεί τις σχέσεις follower-leader ανάμεσα στα partitions του cluster. Σε περίπτωση σφάλματος ο προβληματικός κόμβος ενημερώνει όλα τα αντίγραφα ασφαλείας (replicas) για την αποχώρησή του και αυτά με την σειρά τους προχωρούν στην εκλογή ενός καινούργιου controller για το σύστημα. Κάθε δεδομένη στιγμή υπάρχει μοναδικός controller τον οποίο γνωρίζουν όλοι και για τον οποίο έχουν συμφωνήσει όλοι.
Ρύθμιση παραμέτρων των topics (Configuration Of Topics): Οι ρυθμίσεις που αφορούν τα topics όπως ονόματα υπαρχόντων topics, αριθμός και τοποθεσία αντιγράφων ασφαλείας (replicas), αριθμός partitions, λίστα ειδικών ρυθμίσεων του topic καθώς και ποιός κόμβος είναι ο leader για κάθε topic αποθηκεύονται στον Zookeeper.
Πληροφορίες Πρόσβασης (Access control lists): Οι πληροφορίες για το ποιοί χρήστες έχουν πρόσβαση σε συγκεκριμένα topics αποθηκεύεται στον Zookeeper.
Πληροφορίες για το cluster (Membership of the cluster): Στον Zookeeper ακόμα αποθηκεύεται η λίστα με τους ενεργούς brokers του cluster κάθε δεδομένη στιγμή.

5.4 Χαρακτηριστικά Kafka

Retention (Αποθήκευση)

Ο Kafka server προσφέρει την δυνατότητα αποθήκευσης των μηνυμάτων σε ένα topic είτε για συγκεκριμένο χρονικό διάστημα (7 μέρες by default) είτε μέχρι τα μηνύματα να φτάσουν ένα συγκεκριμένο αριθμό σε μέγεθος (για παράδειγμα 1 GB). Μετά την ικανοποίηση κάποιου από τα παραπάνω τα μηνύματα διαγράφονται με βάση την παλαιότητα έτσι ώστε να υπάρχουν πάντα δεδομένα στο topic. Ανάλογα με το topic οι ρυθμίσεις αποθήκευσης μπορούν να μεταβάλλονται για παράδειγμα σε πολύ ενεργά topics μπορεί το χρονικό όριο αποθήκευσης να είναι κάποιες ώρες [6].

Reliability Guarantees (Εγγυήσεις Αξιοπιστίας) [6]

- Χρονική σειρά των μηνυμάτων σε ένα partition. Αν το μήνυμα Α γράφεται πριν το μήνυμα Β από τον ίδιο producer στο ίδιο partition τότε το μήνυμα Β θα έχει μεγαλύτερο offset από το μήνυμα Α. Δηλαδή ένας consumer θα λάβει πρώτα το μηνυμα Α και μετά το Β.

- Τα μηνύματα λέμε ότι είναι committed όταν γράφονται στον partition leader και συγχρονίζονται όλα τα αντίγραφα (replicas) του partition με το στιγμιότυπο του leader. Ο Kafka εγγυάται ότι τα committed μηνύματα δεν θα χαθούν εφόσον υπάρχει τουλάχιστον μια replica που τα περιέχει και η πολιτική αποθήκευσης δεν έχει ενεργοποιήσει την διαγραφή τους.

- Οι consumers μπορούν να διαβάζουν μόνο committed μηνύματα. Ο Kafka παρέχει at-least-once message delivery semantics, συνεπώς το committed μήνυμα δεν θα χαθεί αλλά δεν αποφεύγεται η ανάγνωση διπλότυπων μηνυμάτων. Για exactly-once semantics βασιζόμαστε είτε σε εξωτερικά συστήματα είτε στα Kafka Streams.

Trade-Offs [6]
Όπως σε κάθε κατανεμημένο σύστημα έτσι και στον Kafka καλούμαστε να βρούμε την χρυσή τομή ανάμεσα στην δυνατότητα τον εγγυήσεων που παρέχονται. Σε αυτήν την περίπτωση, ο χρήστης καλείται να επιλέξει ανάμεσα σε αξιοπιστία (reliability) και συνέπεια (consistency) εναντίον διαθεσιμότητα (availability), υψηλής διεκπεραιωτικότητας (high throughput) και χαμηλής καθυστέρησης (low latency). Σε όλες τις περιπτώσεις το σύστημα εγγυάται σε μεγάλο βαθμό όλα τα παραπάνω, σε περίπτωση όμως που θέλουμε αυξημένες εγγυήσεις για ένα κομμάτι θα πρέπει να θυσιάσουμε μια άλλη πτυχή του συστήματός μας. Για παράδειγμα, αν θέλουμε χαμηλή καθυστέρηση στην ανάγνωση μηνυμάτων από ένα topic το οποίο είναι αντίγραφο ενός topic του cluster, πρέπει να είμαστε διατεθειμένοι να δεχτούμε την πιθανότητα το μήνυμα που θα πάρουμε να είναι λάθος λόγω έλλειψης χρόνου ανανέωσης της τιμής του μηνύματος με βάση την τιμή του αυθεντικού μηνύματος στον leader του topic.

5.5 Πλεονεκτήματα & Μειονεκτήματα

Πλεονεκτήματα

1. Αντιμετωπίζει την πολυπλοκότητα της ενοποίησης και ενσωμάτωσης (integration)

2. Καλό για συστήματα που ασχολούνται με ETL/CDC/Big Data Ingestion

3. Τοπική Αποθήκευση δεδομένων

4. Δυνατότητα πολλαπλών producers/consumers

5. Εξαιρετικά Κλιμακώσιμο
6. Αντοχή σε σφάλματα
7. Ικανοποιητικά χαμηλές καθυστερήσεις
8. Εξαιρετικά διαμορφώσιμο στις ανάγκες του κάθε χρήστη
9. Μπορεί να διαχειριστεί αποδοτικά real-time pipelines δεδομένων

Μειονεκτήματα

1. Χρειάζεται αρκετό χρόνο για να κατανοηθεί πλήρως ο τρόπος λειτουργίας και να φτάσει κάποιος σε ικανοποιητικό επίπεδο γνώσης.
2. Του λείπει ένα πλήρες σετ από εργαλεία διαχείρισης και παρακολούθησης. Πράγμα που απωθεί πιθανούς πελάτες.
3. Καμιά φορά όταν αντιμετωπίζει μεγάλο φόρτο εργασίας συμπεριφέρεται άτσαλα και παρατηρούνται καθυστερήσεις.

Κεφάλαιο 6

Apache Spark & Spark Streaming

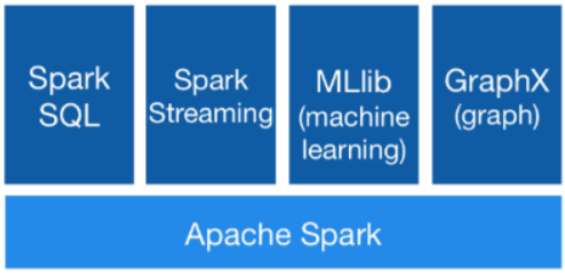
6.1 Apache Spark

Το Apache Spark είναι ένα open source κατανεμημένο framework, το οποίο λειτουργεί με ομάδες clusters, με σκοπό υπολογισμούς γενικών και διαφόρων τύπων πάνω σε δεδομένα. Το Spark δημιουργήθηκε στο πανεπιστήμιο Berkeley της California και δωρίστηκε στην Apache Software Foundation που ανέλαβε την συντήρησή του από τότε. Το Spark κέρδισε την αγορά διότι παρέχει μια διεπαφή που επιτρέπει την παραγωγή ολόκληρων ομάδων (clusters) με δυνατότητες παραλληλοποίησης και μεγάλη ανοχή σε σφάλματα. Το Apache Spark αποτελείται από ένα σύνολο εργαλείων, όπως απεικονίζεται παρακάτω. Τα επιμέρους αυτά τμήματα μπορούν να συνεργάζονται μεταξύ τους, παρέχοντας πληθώρα δυνατοτήτων στην επεξεργασία και ανάλυση δεδομένων.

Σχήμα 6.1: Apache Spark Logo

Πιο αναλυτικά, το Apache Spark απαρτίζεται από τα ακόλουθα [47]

- **Spark Core:** Αποτελεί την υποκείμενη μηχανή της πλατφόρμας Spark, πάνω στην οποία έχουν χτιστεί όλες οι άλλες λειτουργικότητες. Παρέχει In-Memory computing και την αλληλεπίδραση με εξωτερικά συστήματα αποθήκευσης.
- **Spark SQL:** Αποτελεί το τμήμα του Spark που παρέχει υποστήριξη για δομημένα και ημι-δομημένα δεδομένα, μέσω του SchemaRD.



Σχήμα 6.2: Spark Core

- **Spark Streaming:** Επιτρέπει την επεξεργασία και ανάλυση ροών δεδομένων σε πραγματικό χρόνο. Απορροφά δεδομένα σε mini-batches και εκτελεί μετασχηματισμούς RDD (Resilient Distributed Datasets) σε αυτά. Τα RDD είναι πρακτικά μια read-only συλλογή από κατανεμημένα δεδομένα τα οποία είναι αμετάβλητα και είναι χωρισμένα σε partitions τα οποία μπορούν να διαχειρίζονται από διαφορετικά clusters [48].
- **MLlib (Machine Learning):** Αποτελεί κατανεμημένο framework και παρέχει αλγορίθμους μηχανικής μάθησης.
- **GraphX:** Αποτελεί κατανεμημένο framework επεξεργασίας γράφων και εκτέλεσης παράλληλων υπολογισμών σε αυτούς.

Στην παρούσα εργασία δίνουμε ιδιαίτερη βάση στο Spark Streaming το οποίο αναλύεται και παρακάτω.

6.2 Spark Streaming

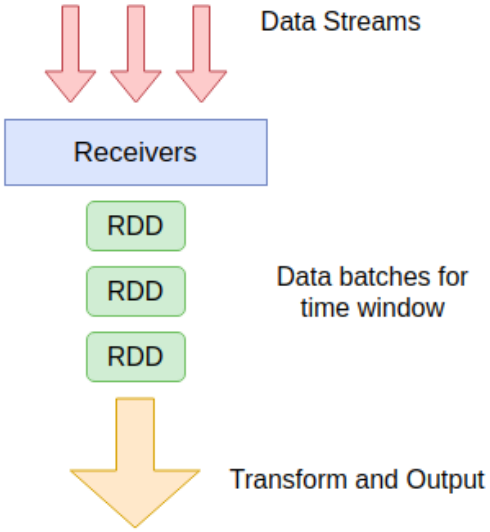
Τα δεδομένα στις μέρες μας είναι ένα μέγεθος που συνεχώς αυξάνεται και γίνεται πιο πολύπλοκο. Συνεπώς, η συχνή επεξεργασία των δεδομένων μέσω του Spark φαίνεται πως δεν είναι καλή πρακτική καθώς μας κοστίζει χρόνο και εισάγει μεγάλες καθυστερήσεις στην παρακολούθηση ενός πραγματικού συστήματος. Συνεπώς μας βολεύει να επεξεργαζόμαστε μόνο τα καινούργια δεδομένα που εισάγονται και να ενημερώνουμε τη βάση δεδομένων ειδικά αν μιλάμε για εφαρμογές που αντιδρούν σε μετρήσεις αισθητήρων ή γενικά ερεθισμάτων από το περιβάλλον.

Σχήμα 6.3: Apache Spark Streaming Logo

Για παράδειγμα, σε ένα έξυπνο σπίτι που χρησιμοποιεί εφαρμογές IoT έχουμε αισθητήρες για την θερμοκρασία, θα μας βόλευε να επεξεργαζόμαστε τις αλλαγές στις μετρήσεις και να αντιδρούμε σε αυτές, ανοίγοντας την θέρμανση ή τον κλιματισμό, με μικρή καθυστέρηση και όχι να περιμένουμε τις μετρήσεις μιας ολόκληρης μέρας ή και ώρας μέχρι να μπορούμε να αντιδράσουμε. Το Spark Streaming χρησιμοποιεί την γρήγορη δρομολόγηση του Spark Core έτσι ώστε να επεξεργάζεται τα δεδομένα σε μικρά κομμάτια (mini batches) παρέχοντας μεγάλη ευελιξία.

6.2.1 High-Level

Όπως φαίνεται και στο παρακάτω σχήμα έχουμε μία συνεχή ροή δεδομένων η οποία μπορεί να προέρχεται από αισθητήρες, από κάποιο log ή ακόμα να προέρχεται από μια βάση δεδομένων [43, 27].



Σχήμα 6.4: Spark Streaming High Level Functionality

Έπειτα στην ιεραρχία έχουμε κάποιους δέκτες (receivers) οι οποίοι δέχονται αυτά τα δεδομένα και τα εισάγουν στο Spark Streaming. Όπως αναφέραμε τα δεδομένα μπορεί να είναι πολλών ειδών, συνεπώς, έχουμε και μια ποικιλία από διαφορετικούς receivers που μπορούμε να έχουμε σε αυτό το επίπεδο ανάλογα με τον τύπο των δεδομένων που διαχειρίζεται η εφαρμογή.

Οι receivers είναι υπεύθυνοι να χωρίζουν τα δεδομένα σε μικρά πακέτα πληροφορίας τα RDDs τα οποία περιέχουν πληροφορία για συγκεκριμένες χρονικές στιγμές. Γενικά το πόσο μικρή πληροφορία περιέχεται σε κάθε RDD μπορεί να προσαρμοστεί από τον χρήστη μέσω των ρυθμίσεων έστι ώστε να διευκολύνει την μετέπειτα επεξεργασία. Ανάλογα με τη λειτουργία της εφαρμογής και το είδος των δεδομένων, μπορεί να θέλουμε μικρότερα ή μεγαλύτερα χρονικά πλαίσια για επεξεργασία.

Το αποτέλεσμα αυτού του μετασχηματισμού και της επεξεργασίας που υφίστανται τα δε-

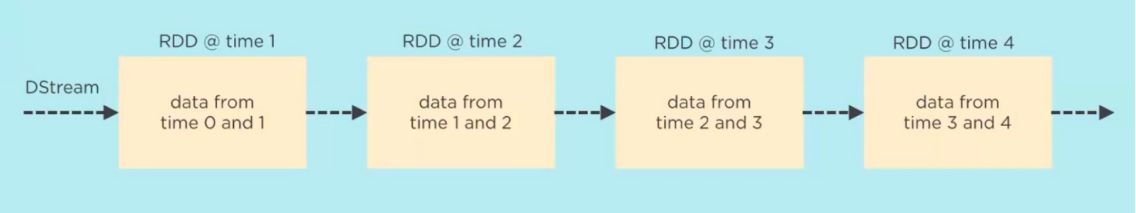
δομένα εξέρχεται από το σύστημα είναι έτοιμα για χρήση. Μπορούν να παρουσιάζονται σε μια συσκευή εξόδου με την μορφή διαγραμμάτων, είτε να είναι είσοδος σε άλλες εφαρμογές και συστήματα.

Τέτοια συστήματα μπορεί να είναι εξωτερικές βάσεις δεδομένων που κρατούν ιστορικό, βάσεις δεδομένων που ανανεώνονται με βάση τις καινούργιες πληροφορίες. Ακόμα, μπορεί τα μηνύματα εξόδου να μην αποθηκεύονται απλά, αλλά να χρησιμοποιούνται για λήψη επιτόπου αποφάσεων, είτε από ανθρώπους, είτε το πιο συνηθισμένο, από αυτοματοποιημένα συστήματα τα οποία είναι υπεύθυνα να ελέγχουν και να αντιδρούν με συγκεκριμένες δράσεις σε ορισμένα γεγονότα.

Για παράδειγμα, στα πλαίσια του IoT οι θερμοστάτης ενός έξυπνου σπιτιού μπορεί να είναι προγραμματισμένοι στο να παρακολουθούν την θερμοκρασία στα δωμάτια και να αποστέλλουν πληροφορίες σε τακτά χρονικά διαστήματα. Αυτές οι πληροφορίες χωρίζονται σε RDDs και επεξεργάζονται. Τα αποτελέσματα αποστέλλονται στο σύστημα κεντρικού ελέγχου του σπιτιού το οποίο παίρνει αποφάσεις να ανοίξει την θέρμανση ή τον κλιματισμό ανάλογα με τα δεδομένα που έχει λάβει σε πραγματικό χρόνο συνδυάζοντάς τα.

6.2.2 Dstream Processing

Ένα ακόμα πλεονέκτημα του Spark Streaming, είναι ότι αντί για τα επιμέρους RDDs το Spark Streaming δίνει την δυνατότητα στον χρήστη αν θέλει να επεξεργάζεται και να εφαρμόζει συναρτήσεις πάνω στο Dstream το οποίο πρακτικά αποτελείται από πολλά RDDs.



Σχήμα 6.5: Dstream Processing

Αυτό δίνει την δυνατότητα στο σύστημά μας να γίνει κατανεμημένο και να μπορεί να χωρίσει το Dstream όπως θέλει στο cluster μας γνωρίζοντας όμως τι επεξεργασίες πρέπει να υποστεί χωρίς εμείς να χρειαστεί να κάνουμε τον χωρισμό γινεται αυτόματα και βέλτιστα από τον λειτουργικό του Spark Streaming [43].

Ορισμένες μετατροπές που μπορούμε να εφαρμόσουμε είναι [43]:

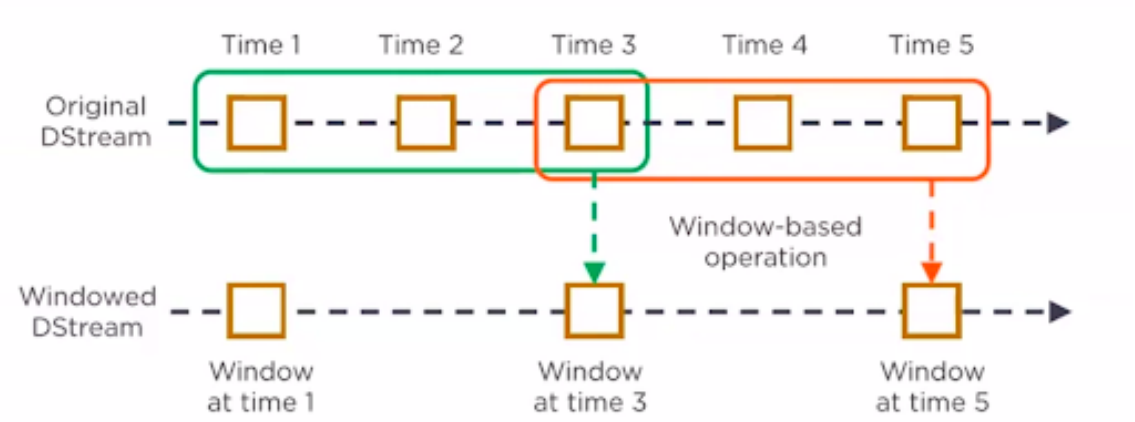
- **map(func):** Επιστρέφει ένα νέο DStream περνώντας κάθε στοιχείο από το DStream εισόδου από την συνάρτηση func.
- **flatmap(func):** Παρόμοιο με το μαπ όμως κάθε στοιχείο μπορεί να αντιστοιχίζεται με 0 ή περισσότερες εξόδους.
- **filter(func):** Επιστρέφει ένα νέο DStream διαλέγοντας τα στοιχεία για τα οποία η συνάρτηση func γυρνάει true.

- **union(otherStream):** Επιστρέφει ένα νέο DStream που περιέχει την ένωση των στοιχείων της εισόδου και του otherStream.

- **join(otherStream, [numTasks]):** Όταν καλείται για δύο DStreams *(X, Y)* και *(X, W)* επιστρέφει ένα νέο DStream *(X, (Y, W))* ομαδοποιώντας όλα τα στοιχεία με κλειδί *X*.

6.2.3 Windowed Processing

Εκτός από την επεξεργασία μεμονωμένων RDDs καθώς και ολόκληρου του Dstream το Spark Streaming δίνει την επίσης την δυνατότητα να εφαρμόζουμε αλλαγές σε ένα "παράθυρο" δεδομένων του Dstream. Το "παράθυρο" είναι ουσιαστικά ένα χρονικό πλαίσιο, το οποίο μπορούμε να ορίσουμε έτσι ώστε να καλύπτει καλύτερα τις ανάγκες που έχει η εφαρμογή μας. Με αυτόν τον τρόπο έχουμε ενοποιημένη επεξεργασία για μια χρονική σειρά δεδομένων μετατοπίζοντας ένα παράθυρο πάνω στη ροή των δεδομένων μας [43].



Σχήμα 6.6: Windowed Processing

Όπως βλέπουμε και από το παράδειγμα, βασικά χαρακτηριστικά αυτού του μοντέλου του Spark Streaming είναι το μέγεθος του παραθύρου και το βήμα μετατόπισης του παραθύρου. Στην συγκεκριμένη περίπτωση, το μήκος παραθύρου (διάρκεια του παραθύρου) είναι 3 και κάθε φορά το παράθυρο μετατοπίζεται κατά 2, τα συγκεκριμένα νούμερα πρέπει να είναι ακέραια πολλαπλάσια του batch size του Dstream. Παρατηρούμε ότι στο τέλος του μήκους παραθύρου παίρνουμε την επεξεργασμένη μορφή των RDDs που ανήκουν στο συγκεκριμένο παράθυρο.

6.2.4 Caching/Persistence

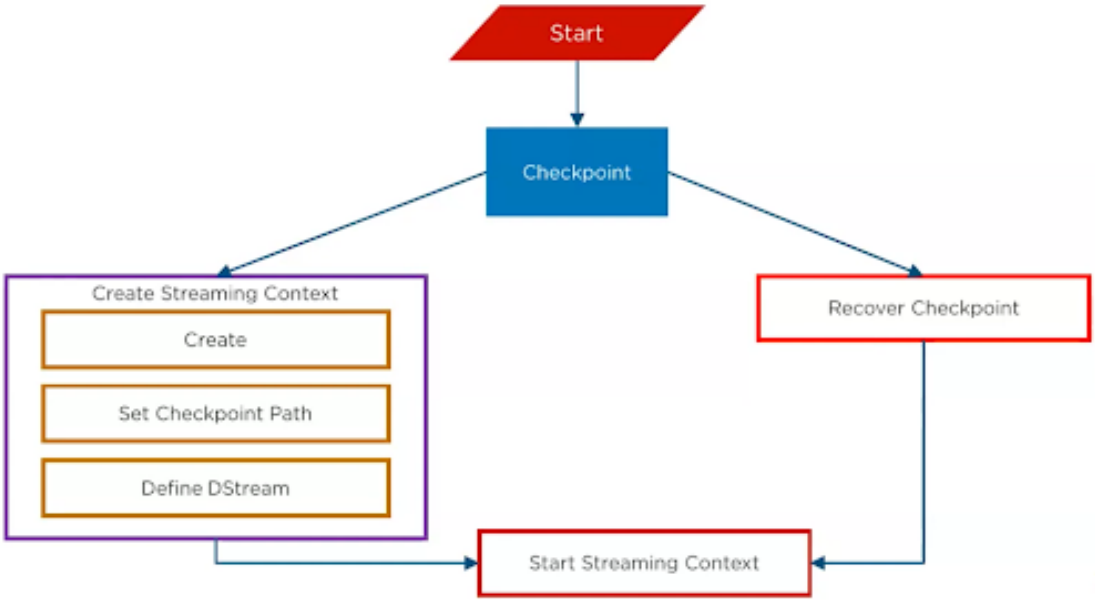
Ένα ακόμα χαρακτηριστικό του Spark Streaming είναι ότι επιτρέπει στους χρήστες του να αποθηκεύουν στην μνήμη τα δεδομένα του Dstream γεγονός που μπορεί να είναι χρήσιμο για δεδομένα που θέλουμε να υπολογιστούν και να χρησιμοποιηθούν πολλές φορές. Για δεδομένα που προέρχονται από το δίκτυο όπως για παράδειγμα Kafka servers η default συμπεριφορά είναι να αποθηκεύονται σε 2 nodes (replicas) προκειμένου να έχουμε ανοχή σε σφάλματα [43].

6.2.5 Checkpointing στο Spark Streaming

Όπως σε όλες τις εφαρμογές φροντίζουμε να έχουμε ανοχή σε σφάλματα και να προλαβαίνουμε τα λάθη πριν γίνουν μη αναστρέψιμα ή προκαλέσουν λάθος αποτελέσματα στις εφαρμογές. Μια πραγματική εφαρμογή συχνά εκτελείται συνεχόμενα χωρίς διακοπή, είναι λοιπόν πολύ πιθανό κάποια στιγμή να προκύψει κάποιο πρόβλημα είτε μεταφοράς δεδομένων είτε κάποιος κόμβος του συστήματος να αποτύχει με αποτέλεσμα λόγω της μεγάλης συνδεσιμότητας του συστήματος να έχουμε ασυνέπειες δεδομένων.

Όπως αναφέραμε στην περιγραφή του Kafka Cluster έχουμε έναν κόμβο στο σύστημα που επιβλέπει την σωστή λειτουργία του συστήματος, καθώς και κάποια nodes που έχουν πρωτότυπη πληροφορία για κάποια topics αλλά και αντίγραφα. Στην περίπτωση που ο επιβλέπων κόμβος αποτύχει το σύστημα χάνει τον έλεγχο όλου του cluster και τις λειτουργίες που πρέπει να γίνουν σε αυτό, και ως αποτέλεσμα, δε μπορεί πλέον να εξυπηρετεί τους χρήστες του. Αν ακόμα, κάποιος κόμβος που ελέγχει πρωτότυπο περιεχόμενο αποσυνδεθεί από το σύστημα περνάμε σε μία κατάσταση που τα δεδομένα μπορεί να μην είναι επικαιροποιημένα στα αντίγραφα ή να είναι μόνο σε κάποια συνεπώς για ίδιες ερωτήσεις ο χρήστης μπορεί να πάρει διαφορετικες απαντήσεις. Αυτό είναι σοβαρό πρόβλημα των κατανεμημένων συστημάτων και λύνεται σε μεγάλο βαθμό από την αποθήκευση χρονικών στιγμών που το σύστημα είναι "υγιές" με σκοπό να μπορούμε να ανατρέξουμε σε αυτές αν χρειαστεί. Αυτές οι στιγμές ονομάζονται checkpoints και είναι άλλος ένας τρόπος να θωρακίσουμε την εφαρμογή μας από σφάλματα [43].

Λειτουργία



Σχήμα 6.7: Checkpointing Flow Chart

Το Checkpointing είναι μια διαδικασία που ουσιαστικά χωρίζει όλη την λειτουργία των εργασιών πάνω στα δεδομένα σε κομμάτια. Αν υποθέσουμε ότι κάθε ενέργεια (συνάρτηση) σε RDD είναι ένας κόμβος ενός ακυκλικού κατευθυνόμενου γράφου και η ακμές δείχνουν την κατεύθυνση, που είναι πρακτικά η χρονική σειρά, το Checkpointing προσπαθεί να βρει τις καλύτερες τομές στον γράφος για να εισάγει σημεία "εισόδου" σε περίπτωση σφάλματος όπως φαίνεται στο παρακάτω διάγραμμα [43].

Σχήμα 6.8: Checkpointing Schema

Κατόπιν, αποθηκεύει την κατάσταση της εφαρμογής στο Hadoop HDFS που αποτελεί αξιόπιστο αποθηκευτικό χώρο και αν χρειαστεί από εκεί αντλεί τις πληροφορίες για το restart των drivers στην περίπτωση σφάλματος. Όπως φαίνεται και στο παραπάνω διάγραμμα (σχήμα 6.8) η διαδικασία του restart δεν είναι απαραίτητα απλή. Για παράδειγμα, στο παραπάνω restart (σχήμα 6.8), θα πρέπει να αναιρεθεί το αποτέλεσμα του δεύτερου μαπ στα δεδομένα.

Για τέτοιου είδους περιπτώσεις έχουμε δύο τύπους δεδομένων για τα οποία μπορούμε να κάνουμε checkpoint [43].

1. **Metadata Checkpointing:** Metadata είναι τα δεδομένα που αφορούν άλλα δεδομένα. Περιέχουν δεδομένα που έχουν να κάνουν με ρυθμίσεις που αφορούν την δημιουργία streaming operations, ενέργειες πάνω σε DStream και ημιτελών τεμαχίων (batches)δεδομένων. Όλα αυτά αποθηκεύονται στο HDFS.

2. **Data Checkpointing:** Αφορά την αποθήκευση μεμονωμένων RDDs. Στην περίπτωση που έχουμε RDDs που εξαρτώνται από RDDs προηγούμενων τεμαχίων δεδομένων είναι φανερό πώς η πολυπλοκότητα αυξάνεται όσο περνάει ο χρόνος, για να αποφύγουμε λοιπόν μεγάλους χρόνους ανάκαμψης τα ενδιάμεσα RDDs αποθηκεύονται περιοδικά σε βάσεις δεδομένων. Αυτός είναι και ο τρόπος που παρουσιάζεται στο διάγραμμα (σχήμα 6.8) που διαχωρίζουμε την αλυσίδα ενεργειών με την εισαγωγή checkpoints.

6.2.6 Shared Variables στο Spark Streaming

Όλες οι συναρτήσεις που παρέχει το εργαλείο, τις οποίες αναφέραμε και παραπάνω, εκτελούνται σε ένα κόμβο (node) του cluster. Για τους λόγους που έχουμε εξηγήσει, τα δεδομένα δεν βρίσκονται μόνο σε ένα αντίγραφο έτσι κάθε διεργασία (εκτελεστής) παίρνει ένα αντίγραφο των μεταβλητών που χρειάζεται για την εκτέλεσή της, όμως δεν ενημερώνει το κεντρικό πρόγραμμα (driver program) με τις αλλαγές που έκανε ούτε και είναι ενήμερη για τυχόν αλλαγές που έχουν κάνει ενδεχομένως άλλα προγράμματα στα δεδομένα που χρησιμοποίησε.

Αν εισάγουμε κοινές read/write μεταβλητές τις οποίες να τις μοιράζονται όλες οι διεργασίες θα προκύψει ένα μη αποδοτικό σύστημα, γι αυτό λοιπόν το Spark Streaming εισάγει δύο τύπους κοινών (shared) μεταβλητών, τους **accumulators** και τις **broadcast variables** [43].

1. **Accumulators:** Οι Accumulators είναι μεταβλητές που χρησιμοποιούνται για τη συγκέντρωση πληροφοριών μεταξύ των εκτελεστών. Είναι εφαρμόσιμοι σε οποιαδήποτε διεργασία είναι:

   (α′) προσεταιριστική → $f(\, f(x,\, y),\, z\,) = f(\, f(x,\, z),\, y\,) = f(\, f(y,\, z),\, x\,)$

   (β′) αντιμεταθετική → $f(x,\, y) = f(y,\, x)$

   Χρησιμοποιούνται κυρίως για αθροίσματα, για διαδικασίες map/reduce και για συναρτήσεις εύρεσης μέγιστου, όχι όμως για average αφού δεν ικανοποιούν τις παραπάνω δύο αρχές.

2. **Broadcast Variables:** Η μέθοδος αυτή επιτρέπει στον προγραμματιστή να διατηρεί μια read only μεταβλητή στην μνήμη cache κάθε κόμβου (node) αντί να την στέλνει όπου χρειάζεται με ειδικές διεργασίες. Αυτή η μέθοδος χρησιμοποιείται συνήθως όταν θέλουμε να κάνουμε join σε κάποια RDDs. Αν τα RDDs που διαχειριζόμαστε έχουν μορφή (key, value) τότε το σύστημα φροντίζει αυτά με το ίδιο key να αποθηκεύονται στο ίδιο κόμβο, όσο αυτό είναι εφικτό.

   Στην περίπτωση όμως που αυτό δεν γίνεται, πρέπει να σταλούν στον κόμβο που εκτελείται η συνάρτηση οι πληροφορίες που χρειάζεται από όλους τους άλλους κόμβους, πράγμα που είναι μη αποδοτικό και χρονοβόρο. Σε αυτήν την περίπτωση μπορούμε να δημιουργήσουμε μια broadcast μεταβλητή σε κάθε κόμβο που έχει να στείλει πληροφορία στον κόμβο-εκτελεστή και να στείλουμε αυτήν. Με αυτόν τον τρόπο αποθηκεύονται η πληροφορίες τοπικά στον κόμβο-εκτελεστή και οι διεργασίες γίνονται πιο εύκολα και γρήγορα αφού πλέον ο κόμβος που εκτελεί τις μετατροπές έχει τα δεδομένα σε τοπικά αντίγραφα και δεν χρειάζεται να τα ζητήσει ξανά.

Κεφάλαιο 7

Kafka Streams

7.1 Εισαγωγή

Το Kafka Streams είναι μια βιβλιοθήκη που χρησιμοποιείται για την ανάλυση και επεξεργασία δεδομένων που αποθηκεύονται στο Kafka. Οι έξοδοι του εργαλείου αποθηκεύονται σε Kafka clusters ή στέλνονται με την σειρά τους σε εξωτερικά συστήματα. Φροντίζει για την εγκυρότητα που πρέπει να διέπει κάθε εφαρμογή επεξεργασίας ροής δεδομένων, δηλαδή τον διαχωρισμό ανάμεσα στον χρόνο που συμβαίνουν τα γεγονότα, τον χρόνο που επεξεργάζονται, την δυνατότητα επεξεργασίας χρονικού "παραθύρου" δεδομένων και την απλή διαχείριση της κατάστασης της εφαρμογής [18].

Χρησιμοποιεί πολλά στοιχεία του Kafka όπως την κλιμάκωση με δημιουργία partition στα topics και για αυτό τον λόγο είναι σχετικά "ελαφρύ" και μπορεί να ενταχθεί σε μια εφαρμογή.

Το Kafka Streams επεξεργάζεται γεγονότα με πολύ μικρή καθυστέρηση έχοντας δυνατότητες για κατανεμημένες εντολές join και άλλων αθροιστικών ενεργειών καθώς και δυνατότητα εφαρμογής των παραπάνω σε χρονικά παράθυρα δεδομένων. Ακόμα, χρησιμοποιεί κατανεμημένη επεξεργασία δεδομένων με μεγάλη ανοχή στα σφάλματα και γρήγορους αλγορίθμους ανάκαμψης. Βασικό πλεονέκτημα ότι για να τρέξει η εφαρμογή δεν χρειάζεται deploy πράγμα που μπορεί διαλέγοντας το Apache Spark να προσθέτει πολυπλοκότητα στην εφαρμογή.

Χρησιμοποιείται κυρίως για την συγγραφή εφαρμογών σε γλώσσες Java και Scala εκμεταλλευόμενο τα προτερήματα του Kafka στο server side κομμάτι της εφαρμογής [18].

Σχήμα 7.1: Kafka Streams Logo

7.2 Τοπολογία της Επεξεργασίας Ροής (Stream Processing Topology)

Στο Kafka Streams η πιο βασική έννοια είναι το Stream. Stream ονομάζουμε μια απεριόριστη, συνεχώς ανανεούμενη ροή δεδομένων, η οποία έχει χρονική σειρά, μπορεί να επαναληφθεί, είναι ανεκτική στα σφάλματα και δεν αλλάζει τιμές. Τα δεδομένα (data records) αυτά αναπαριστώνται με ένα συνδυασμο key-value (κλειδί - τιμή) [18].

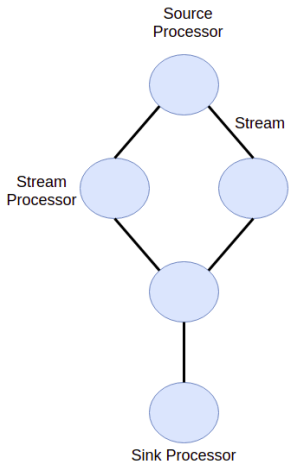
Η τοπολογία κάθε Stream processing εφαρμογής μπορεί εύκολα να οπτικοποιηθεί ως ένας γράφος. Οι κόμβοι αυτού του γράφου είναι οι επεξεργαστές των δεδομένων (stream processor) προγραμματισμένοι να εκτελούν συγκεκριμένες λειτουργίες στα δεδομένα που εισέρχονται από τους κόμβους του ανώτερου στρώματος και να προωθούν τα αποτελέσματά τους στους κατώτερους κόμβους. Ενώ οι ακμές είναι το stream που μεταφέρεται από τον έναν κόμβο στον άλλο. Κάθε κόμβος μπορεί να έχει ένα ή περισσότερα stream εισόδου και εξόδου [18].

Στην τοπολογία που αναφέραμε υπάρχουν δύο ειδικοί τύποι stream processors:

1. **Source Processor:** Ο συγκεκριμένος Processor είναι ιδιαίτερος διότι δεν έχει κάποιον άλλο σε υψηλότερο επίπεδο. Σκοπός του είναι να καταναλώνει δεδομένα από ένα ή περισσότερα Kafka topics και να τα προωθεί στους stream processor των επόμενων βαθμίδων παρέχοντας μια είσοδο για όλη την τοπολογία.

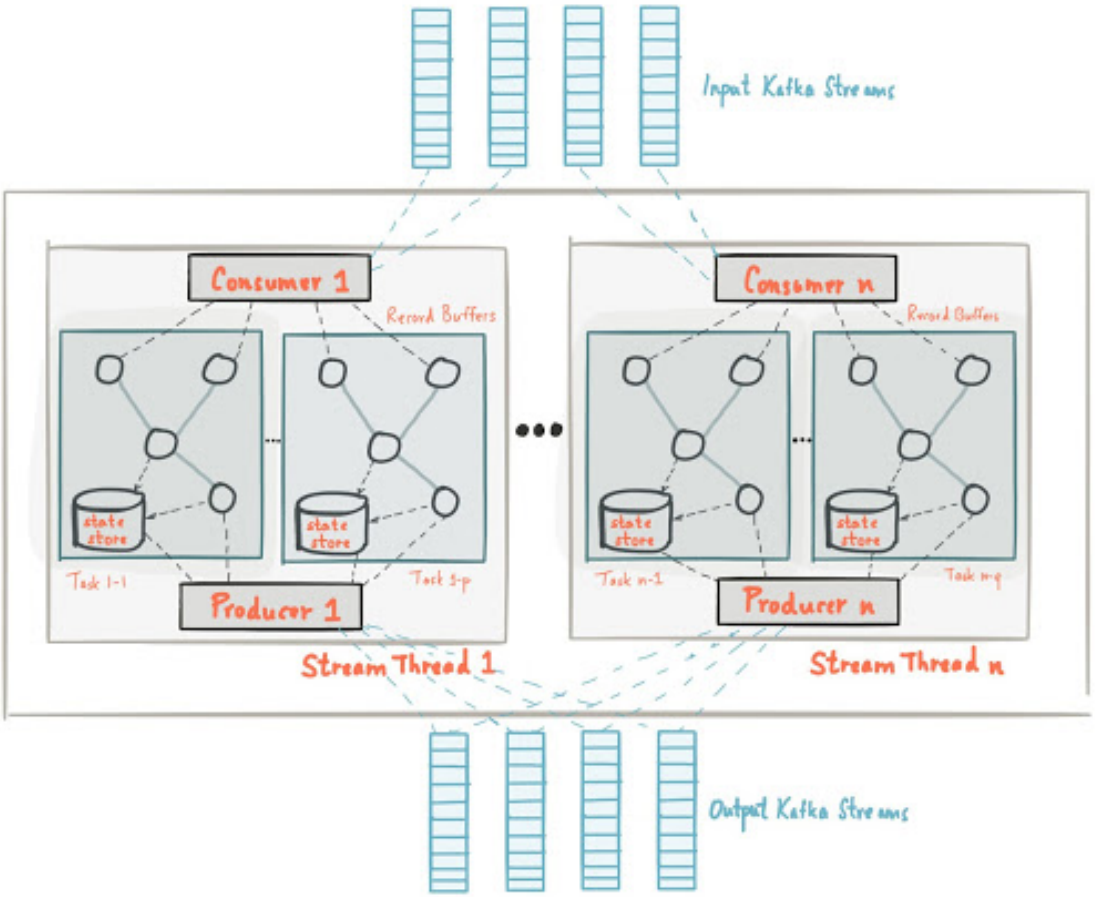
2. **Sink Processor:** Ο συγκεκριμένος Processor είναι ιδιαίτερος διότι δεν έχει κάποιον άλλο σε χαμηλότερο επίπεδο. Σκοπός του είναι να στέλνει τα δεδομένα που λαμβάνει από τους stream processor των ανώτερων επιπέδων σε ένα ή περισσότερα Kafka topics για να αποθηκευτούν.

Η παρακάτω τοπολογία αποτελεί ένα απλό παράδειγμα αυτών που αναφέραμε. Σημειώνεται ότι σε μια τοπολογία δεν υπάρχει περιορισμός στον αριθμό των Source και Sink Processors.



Σχήμα 7.2: Stream Processing Topology

7.3 Αρχιτεκτονική



Σχήμα 7.3: Kafka Streams Architecture

7.3.1 Stream Partitions και Tasks [13]

Το Kafka Streams χρησιμοποιεί τις έννοιες partitions και tasks ως δομικά στοιχεία του βασισμένο στον Kafka και το topic partition που γίνεται εκεί με σκοπό τον πιο αποδοτικό παραλληλισμό.

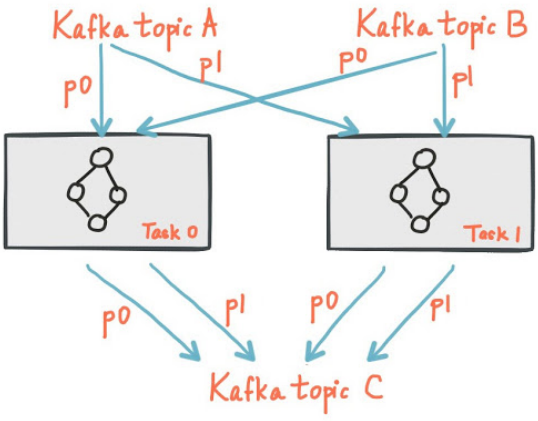
Η στενή σχέση μεταξύ Kafka Streams και Kafka φαίνεται ακόμα και από τα παρακάτω:

- Κάθε stream partition είναι μια σειρά από data records που αντιστοιχίζονται σε ένα Kafka topic partition.
- Κάθε data record στο stream αντιστοιχίζεται σε ένα kafka message του topic.
- Τα κλειδιά των δεδομένων μπορούν να χρησιμοποιηθούν για να γίνει partitioning τόσο στον Kafka όσο και στο Kafka Streams.

Η κλιμάκωση της τοπολογίας της εφαρμογής γίνεται χωρίζοντάς την σε πολλαπλά Tasks. Το Kafka Streams δημιουργεί έναν συγκεκριμένο αριθμό από Tasks με βάση τον αριθμό των partitions που υπάρχουν στο topic που εξετάζει η εφαρμογή. Αυτή η ανάθεση partition σε task δεν αλλάζει ώστε να το κάθε task να κλιμακώνεται μαζί με την εφαρμογή και να είναι υπεύθυνο για το partition που του έχει ανατεθεί.

Αυτή η δυνατότητα δίνει μεγάλη ευελιξία και ταχύτητα στην εφαρμογή αφού πρακτικά μπορεί να διαμοιράσει την δουλειά σε διαφορετικούς επεξεργαστές (άλλα μηχανήματα, cloud, Virtual Machines). Παρόλα αυτά δεν μπορούμε να βάλουμε όσα διαφορετικά application instances θέλουμε αφού ο αριθμός τους εξαρτάται από τον αριθμό των partitions στο συγκεκριμένο input topic. Στην περίπτωση που χρησιμοποιούμε περισσότερα application instances κάποια είναι καταδικασμένα να μένουν ανενεργά (idle), όμως στην περίπτωση που χρησιμοποιούμε κάποιου σφάλματος αν δηλαδή ένα application instance "πέσει" η εφαρμογή συνεχίζει να εκτελείται κανονικά αφού κάποιο άλλο (idle) θα ξεκινήσει αυτόματα από το σημείο που σταμάτησε όταν παρουσιάστηκε το πρόβλημα.

Το παρακάτω διάγραμμα δείχνει δύο tasks στα οποία έχει ανατεθεί ένα partition από κάθε input stream.

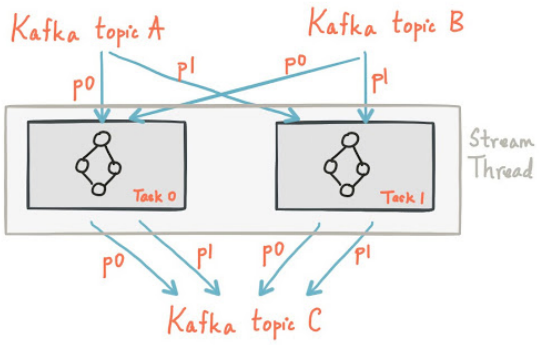


Σχήμα 7.4: Kafka Streams Tasks

7.3.2 Threading Model

Όπως αναφέραμε παραπάνω τα topic partitions ανατίθενται σε task που με την σειρά τους ανατίθενται σε νήματα. Τα νήματα αυτά (threads) είναι κοινά για όλα τα instances και για την βελτιστοποίηση της απόδοσης και τις αποδοτικότητας στον χειρισμό τους χρησιμοποιούν την κλάση StreamsPartitionAssignor την οποία ο χρήστης δεν μπορεί να μετασχηματίσει. Αυτό όμως που μπορεί ο χρήστης να αλλάξει είναι ο αριθμός των νημάτων που μπορεί να χρησιμοποιήσει η εφαρμογή για να παραλληλοποίηση την επεξεργασία σε ένα instance [13].

Το παρακάτω διάγραμμα δείχνει ένα stream thread να εκτελεί δύο stream tasks.

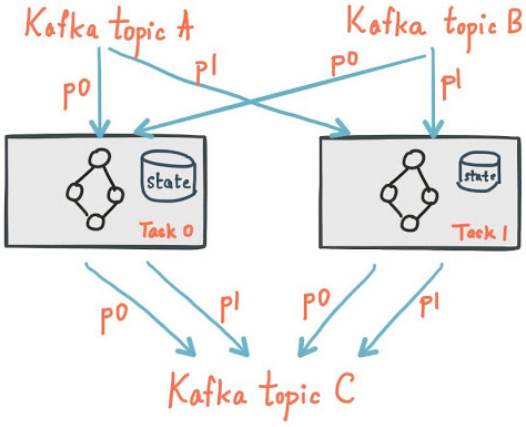


Σχήμα 7.5: Thread executing Tasks

7.3.3 Local State Stores

Το Kafka Streams παρέχει state store που χρησιμοποιούνται από την εφαρμογή για να αποθηκεύουν και να κάνουν queries πάνω στα data. Δημιουργούνται αυτόματα όταν χρησιμοποιούνται stateful operations και widowing σε ένα stream. Κάθε εφαρμογή μπορεί να έχει πολλά διαφορετικά state store που είναι προσπελάσιμα μέσω συγκεκριμένων APIs [13].

Το παρακάτω διάγραμμα δείχνει δύο stream tasks καθένα από τα οποία έχει το δικό του state store.



Σχήμα 7.6: Tasks with State Stores

7.3.4 Fault Tolerance [13]

Οι εφαρμογές Kafka Streams έχουν μεγάλη αντοχή σε σφάλματα γεγονός που προέρχεται και από την μεγάλη σχέση τους με το Kafka το οποίο έχει partitions και replicas για να προστατεύει τα δεδομένα του. Επιπλέον, εκτός από την ασφάλεια των μη επεξεργασμένων δεδομένων, δηλαδή των μηνυμάτων του Kafka broker, το Kafka Stream παρέχει μεγάλη ανοσία από σφάλματα και στα αποτελέσματα των local state stores.

Για κάθε state store διατηρεί ιστορικό αλλαγών (changelog) σε ένα replicated Kafka topic που παρακολουθεί τις αλλαγές που γίνονται από τις εφαρμογές. Με αυτόν τον τρόπο, σε περίπτωση σφάλματος μπορεί να γίνει σωστός καθαρισμός των δεδομένων και να αναλάβει ένα άλλο instance να συνεχίσει την επεξεργασία χωρις ο τελικός χρήστης να παρατηρήσει διαφορές ή να χρειαστεί να ακολουθήσει συγκεκριμένες οδηγίες για ανάκτηση και επανάληψη της εφαρμογής.

Σημαντικό είναι επίσης να αναφέρουμε ότι προκειμένου το changelog να μην μεγαλώνει ασταμάτητα ακολουθεί την τακτική Log Compaction που με αποτέλεσμα να κάνει συχνές εκκαθαρίσεις στα άχρηστα δεδομένα μειώνοντας το απαιτούμενο αποθηκευτικό χώρο ενώ παράλληλα διασφαλίζει ότι θα έχει τουλάχιστον την τελευταία γνωστή τιμή για κάθε κλειδί μηνύματος που βρίσκεται στο log.

7.4 Βασικές έννοιες του Kafka Streams

7.4.1 Χρόνος (Time)

Όπως σε κάθε εφαρμογή που επεξεργάζεται streams με real time δεδομένα έτσι και στο Kafka Streams η έννοια του χρόνου είναι πολύ σημαντική. Η αντίληψη του χρόνου όταν ασχολούμαστε με δεδομένα σε μορφή Stream έχει την μορφή [17]:

1. **Event-time:** Η χρονική στιγμή την οποία προκύπτει το γεγονός (event) ή το αρχείο δεδομένων (data record), δηλαδή η ακριβής στιγμή που παράγεται από την πηγή. Συνήθως ο τρόπος να έχουμε αυτή την πληροφορία είναι μέσω timestamps το οποίο ενσωματώνεται μέσα στο data record.

2. **Processing-time:** Η χρονική στιγμή την οποία το γεγονός (event) ή το αρχείο δεδομένων (data record) επεξεργάζεται από την εφαρμογή. Το processing-time, ανάλογα με την λειτουργία της εφαρμογής, μπορεί να απέχει δευτερόλεπτα, ώρες ή ακόμα και μέρες από το αρχικό event-time.

3. **Ingestion-time:** Η χρονική στιγμή την οποία το γεγονός (event) ή το αρχείο δεδομένων (data record) αποθηκεύεται σε ένα topic partition του Kafka broker. Το Ingestion-time είναι παρόμοιο με το event time αφού πάλι έχουμε την ενσωμάτωση timestamp στο δεδομένο. Η βασική διαφορά με το timestamp προκύπτει όταν το αντικείμενο προστεθεί στον Kafka και όχι όταν δημιουργήθηκε πραγματικά.

Στις καινούργιες εκδόσεις του Kafka σε κάθε δεδομένο ανατίθεται ένα timestamp το οποίο ανάλογα με τα configurations μπορεί να είναι είτε το event-time είτε το ingestion-time. Έπειτα στο κομμάτι του Kafka Streams, υπάρχει ένα default timestamp extractor μέσω του οποίου μπορεί να γίνει ανάκτηση του δοσμένου timestamp. Επιπλέον, η βιβλιοθήκη Kafka Streams δίνει την δυνατότητα στον χρήστη να κατασκευάσει εξατομικευμένο timestamp extractor με σκοπό να ανακτά το timestamp από τα δεδομένα με όποιον τρόπο χρειάζεται η λειτουργία της εφαρμογής. Αυτό είναι πολύ χρήσιμο, διότι αποτελεί κοινή πρακτική σε πολλές εφαρμογές

το timestamp να γράφεται από την πηγή στα δεδομένα του μηνύματος που μεταφέρεται στο stream ή ακόμα να τοποθετούνται πάνω από ένα timestamp και να διαλέγεται διαφορετικό ανάλογα με την λειτουργία της διεργασίας που θέλουμε να εκτελέσουμε [17].

Γενικά όπως περιγράψαμε και παραπάνω, η βιβλιοθηκη Kafka Streams δίνει πολύ μεγάλη ελευθερία στον χρήστη να διαχειρίζεται τον χρόνο όπως επιθυμεί. Πρέπει όμως πάντα ο χρήστης να έχει στο μυαλό του ότι οι εφαρμογές που ασχολούνται με streams και ειδικά με real time events βασίζονται σε μεγάλο ποσοστό στον χρόνο και συνεπώς πρέπει να τον χειριζόμαστε με προσοχή. Σαν γενική οδηγία το τι αντίληψη χρόνου θα διαλέξουμε έχει να κάνει με την δουλειά που κάνει η εφαρμογή μας, όμως πρέπει να έχουμε πάντα στο μυαλό μας ότι έχουμε στα χέρια μας ένα οικοδόμημα (pipeline) με διαφορετικά κομμάτια και για να έχουμε σωστά αποτελέσματα και αναμενόμενη συμπεριφορά πρέπει να προσέχουμε σε όλα τα επιμέρους κομμάτια να έχουμε σωστή αντίληψη του χρόνου και να έχουμε ένα κοινό σημείο αναφοράς, αυτό σημαίνει ότι αν έχουμε διαφορές ώρας και διαφορές στο ημερολόγιο πρέπει να τις λάβουμε υπόψιν μας και να κάνουμε σωστό συγχρονισμό [17].

7.4.2 Streams και Tables [17]

Προτού αναφερθούμε σε πιο πολύπλοκες έννοιες θα παρουσιάσουμε την έννοια του πίνακα (table). Ουσιαστικά οι έννοιες stream και table παρουσιάζουν διττότητα αφού όπως θα δούμε μπορούμε να εκφράσουμε ένα stream ως πίνακα και το ανάποδο.

- **Stream ως Table:** Ένα stream μπορεί να θεωρηθεί ως ένα αρχείο καταγραφής των αλλαγών (changelog) που συμβαίνουν σε ένα πίνακα. Συνεπώς αναπαράγοντας τα δεδομένα από το αρχείο καταγραφής μπορούμε να κατασκευάσουμε τον πίνακα για την συγκεκριμένη χρονική στιγμή που θέλουμε.
- **Table ως Stream:** Ο πίνακας μπορεί να θεωρηθεί ένα στιγμιότυπο του stream για την συγκεκριμένη χρονική στιγμή, ο οποίος για κάθε ζευγάρι κλειδί-τιμή περιέχει την τελευταία τους τιμή. Συνεπώς, μπορούμε επαναλαμβάνοντας για κάθε key-value ζευγάρι να μετατρέψουμε τον πίνακα σε ένα stream.

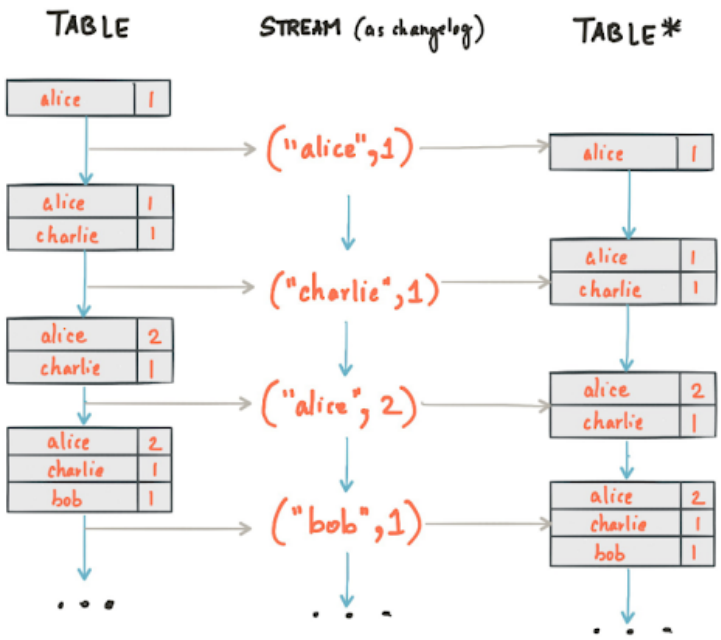
Οι μετατροπές φαίνονται και στο σχήμα 7.7 που από πίνακα περνάμε σε stream και μετά πίσω σε πίνακα.

KStream (record stream)

Αποτελείται από δεδομένα τα οποία έχουν την μορφή (key, value), κάθε στοιχείο στο KStream αποτελεί ένα ξεχωριστό κομμάτι πληροφορίας το οποίο προστίθεται στη ήδη υπάρχουσα ροή.

KTable (changelog stream)

Αποτελείται από τα δεδομένα του αρχείου καταγραφής των αλλαγών στο οποίο κάθε δεδομένο αποτελεί μια ενημέρωση. Πιο συγκεκριμένα, η πληροφορία μέσα σε κάθε στοιχείο καταγραφής αποτελεί την νέα τιμή της καταγραφής με το ίδιο κλειδί (σε περίπτωση που το κλειδί συναντάται πρώτη φορά το δεδομένο απλώς εισάγεται στον πίνακα).



Σχήμα 7.7: From Table to Stream and back again

Η διαφορές μέσα από ένα παράδειγμα:
Στην εφαρμογή μας, έρχονται μέσω του stream οι ακόλουθες 2 καταγραφές: ("Sensor 1", 1), ("Sensor 1", 3)

Στην περίπτωση του KStream αν είχαμε μία εφαρμογή η οποία αθροίζει τα δεδομένα τότε στο αποτέλεσμά μας θα είχαμε ("Sensor 1", 4). Στην περίπτωση του KTable θα είχαμε ("Sensor 1", 3). Αυτό συμβαίνει διότι στο KStream, υπάρχουν και οι δύο τιμές συνεπώς η εφαρμογή μας τις χρησιμοποιεί και τις δύο ενώ στην δεύτερη περίπτωση η δεύτερη τιμή γράφεται πάνω στην πρώτη διότι έχουν το ίδιο κλειδί.

7.5 Μετατροπές και Συναρτήσεις

7.5.1 Stateless Transformations

Δεν απαιτούν κάποιο state για την επεξεργασία και δεν απαιτούν κάποιο local state store. Κάποιες βασικές stateless μετατροπές είναι [15]:

Transformation	Description
Branch • KStream → KStream[]	Χωρίζει το KStream σε διαφορετικά υπο-KStream με βάση δοσμένους κανόνες.
Filter • KStream → KStream • KTable → KTable	Φιλτράρει το KStream ή το KTable απορρίπτοντας κάποιες τιμές με βάση δοσμένους κανόνες.
GroupByKey • KStream → KGroupedStream	Ομαδοποιεί τα δεδομένα με βάση τα κλειδιά τους. Είναι απαραίτητο για συγκεκριμένες μετατροπές που θα δούμε παρακάτω.
Map • KStream → KStream	Δέχεται μια καταγραφή και παράγει μια άλλη, χρησιμοποιείται κυρίως για την αλλαγή κλειδιου και τιμής καθώς και του τύπου των δεδομένων.
Merge • KStream → KStream	Συγχωνεύει δεδομένα από δύο streams σε ένα.

Σχήμα 7.8: Stateless Transformations

7.5.2 Stateful Transformations

Οι συγκεκριμένες μετατροπές βασίζονται στην κατάσταση (state) στην οποία βρίσκεται η εφαρμογή όσον αφορά την επεξεργασία των τιμών εισόδου και την παραγωγή δεδομένων στην έξοδο, ακόμα χρειάζονται απαραίτητα local state stores που να σχετίζονται με την εκάστοτε εφαρμογή που επεξεργάζεται το stream για να έχουμε ανοχή σε σφάλματα. Οι βασικές stateful μετατροπές είναι οι **Windowing**, **Joins** και **Aggregations** [14].

Windowing [17]

Με την συγκεκριμένη μετατροπή μπορούμε να χωρίσουμε το stream μας σε διαφορετικά χρονικά παράθυρα με βάση τον χρόνο, συνήθως το χρησιμοποιούμε σε εφαρμογές σε συνδυασμό με τις δύο επόμενες. Με αυτόν τον τρόπο ο χρήστης μπορεί να ορίσει ένα συγκεκριμένο

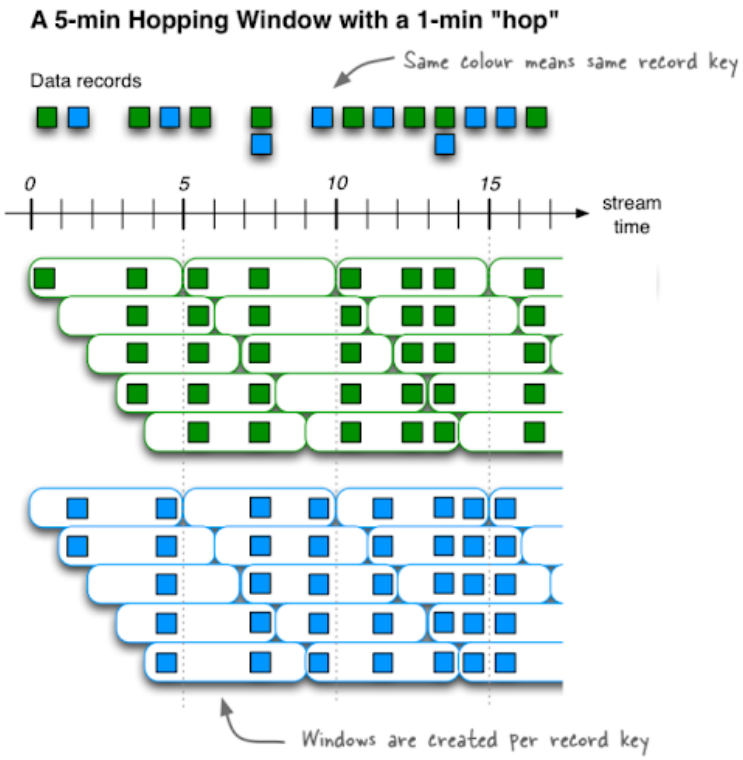
μήκος παραθύρου στο μεσα στο οποίο να επεξεργάζεται τα δεδομένα που του έρχονται με όποιον τρόπο θέλει και παράλληλα να απορρίπτει δεδομένα που έρχονται εκτός σωτής χρονικής σειράς. Είναι πολύ συχνό σε real time εφαρμογές να λαμβάνουμε δεδομένα σε λάθος σειρά με αυτόν τον τρόπο έχουμε τα εργαλεία να αντιμετωπίσουμε αυτό το πρόβλημα διαχειριζόμενοι όπως θέλουμε τα εκτός σειράς γεγονότα [17].

Έχουμε τους παρακάτω τύπους παραθύρων (windows) [14]:

Window name	Behavior	Short description
Hopping time window	Time-based	Fixed-size, overlapping windows
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less windows
Sliding time window	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

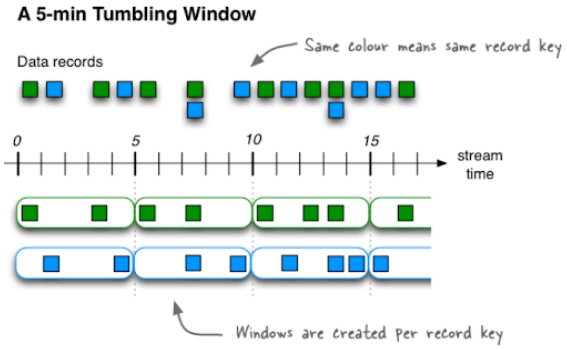
Σχήμα 7.9: Window Names

1. Hopping time windows: Χαρακτηρίζονται από το μήκος τους (window size) και την τιμή της μετατόπισης (hop), δηλαδή τη διαρκεια που προχωράει το παράθυρο στο χρόνο σε σχέση με το προηγούμενο. Με αυτόν τον τρόπο είναι δυνατό να έχουμε επικαλυπτόμενα (overlapping) παράθυρα.



Σχήμα 7.10: Hopping time windows

2. Tumbling time windows: Αποτελούν ειδική περίπτωση του hopping window. Χαρακτηρίζονται μόνο από το μήκος τους (window size), ουσιαστικά αποτελούν ηοππινγ ωινδοως τα οποία έχουν το ίδιο window size και hop, συνεπώς είναι μη επικαλυπτόμενα και δεν παρουσιάζουν κενά μεταξύ τους. Με αυτόν τον τρόπο κάθε δεδομένο θα ανήκει αποκλειστικά σε ένα παράθυρο.



Σχήμα 7.11: Tumbling time windows

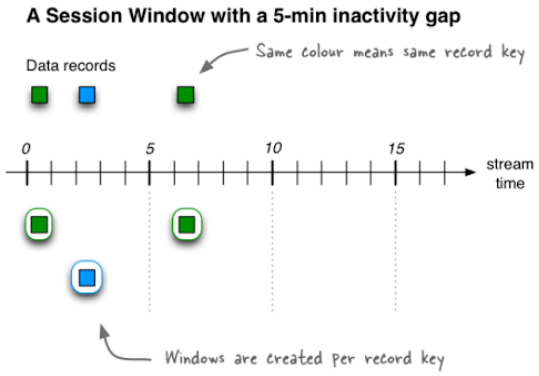
3. Sliding time windows: Παρουσιάζουν αρκετές διαφορές με τα δύο προηγούμενα και χρησιμοποιούνται μόνο σε join διεργασίες. Αποτελεί ουσιαστικά ένα παράθυρο συγκεκριμένου μήκους το οποίο μετατοπίζεται στο χρόνο. Δύο δεδομένα ανήκουν στο ίδιο παράθυρο αν η χρονική διαφορά των timestamps τους είναι μικρότερη ή ίση από το μήκος του παραθύρου. Σε αντίθεση με τα παραπάνω παράθυρα τα δυο άκρα του sliding παραθύρου εντάσσονται στο παράθυρο.

4. Session Windows: Τα sessions αποτελούν ενεργές χρονικές περιόδους οι οποίες χωρίζονται από μια συγκεκριμένου μήκους περίοδο αδράνειας (inactivity period). Όποιο γεγονος πέσει στην περίοδο αδράνειας συγχωνεύεται με τα δεδομένα του συγκεκριμενου session. Αν ένα γεγονός πέσει εκτός του session gap τότε δημιουργείται καινούργιο session.

Ο συγκεκριμένος τύπος παραθύρου διαφέρει με αυτούς που αναφέραμε παραπάνω διότι:

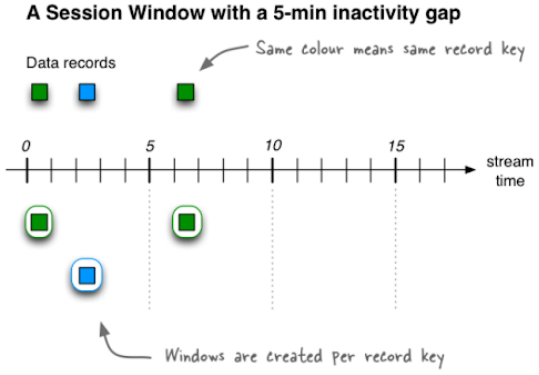
- Όλα τα παράθυρα παρακολουθούνται ανεξάρτητα, με βάση τα κλειδιά (δηλαδή παράθυρα με διαφορετικά κλειδιά έχουν διαφορετικό χρόνο έναρξης και λήξης).
- Το μήκος παραθύρου είναι διαφορετικό, πολλές φορές ακόμα και σε παράθυρα με το ίδιο κλειδί.

Στο σχήμα 7.12 παρουσιάζεται ένα Session Window το οποίο λόγο αδράνειας διαγράφεται και στο επόμενο γεγονός έχουμε δημιουργία νέου session.



Σχήμα 7.12: Inactive Session Window

Αντίθετα, στο σχήμα 7.13 παρουσιάζεται ένα Session Window το οποίο συγχωνεύει τα γεγονότα διότι έχουμε άφιξη μέσα στην περίοδο αδράνειας.



Σχήμα 7.13: Active Session Window

Joins [17]

Η συγκεκριμένη μετατροπή συγχωνεύει δύο streams σε ένα με βάση τα κλειδιά του κάθε stream. Είναι πολύ βασική λειτουργία ειδικά στην περίπτωση που έχουμε δεδομένα από διαφορετικά processing nodes και θέλουμε να τα ενώσουμε. Την χρησιμότητα αυτής της μετατροπής μπορούμε να την δούμε μέσα από ένα παράδειγμα.

Έστω ότι έχουμε μια εφαρμογή που μετράει την συχνότητα εμφάνισης λέξεων σε ένα αρχείο. Για να αυξήσουμε την απόδοση και ταχύτητα έχουμε χωρίσει το αρχείο σε δύο κομμάτια στα οποία τρέχουμε απλά ένα άθροισμα με κλειδί την κάθε λέξη. Στο τέλος του, για να απαντήσουμε για το σύνολο του αρχείου πρέπει να συγχωνεύσουμε τα δύο streams που έχουμε στην έξοδο των αθροιστών με join και να δώσουμε το τελικό αποτέλεσμα.

Αν τώρα υποθέσουμε ότι αντί για αρχείο έχουμε μια ζωντανή ροή όπως για παράδειγμα την μέτρηση των δημοφιλέστερων hashtag στο twitter τότε κλιμακώνοντας την παραπάνω διαδικασία μπορούμε να ακολουθήσουμε την ίδια τακτική. Μια σημαντική διαφορά είναι ότι λόγω του όγκου των δεδομένων δεν μπορούμε να περιμένουμε το τέλος της ροής για να κάνουμε

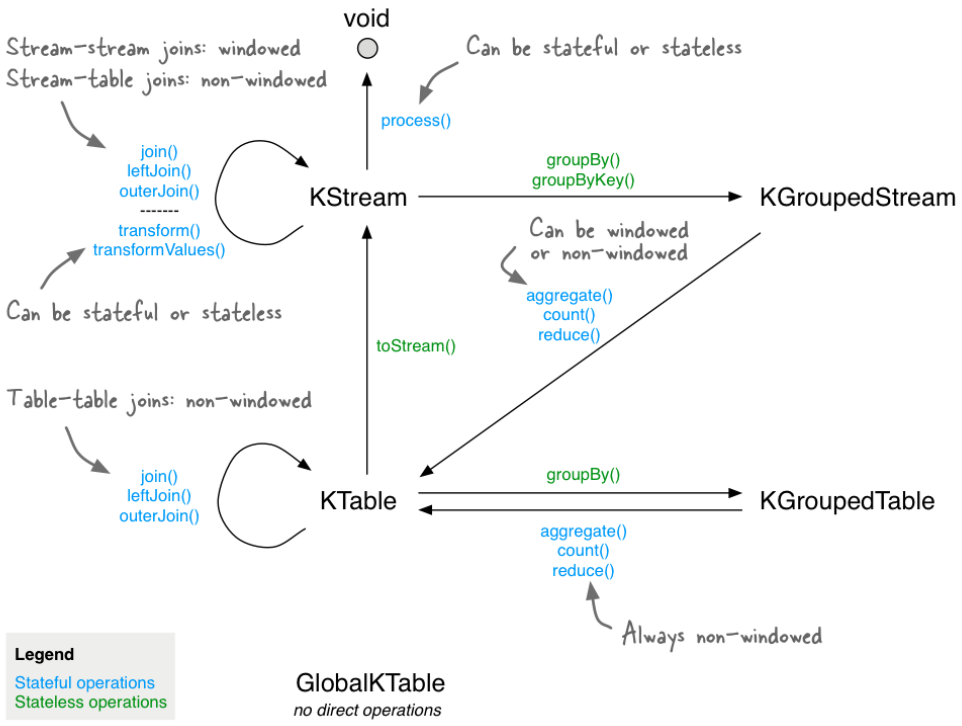
το join συνεπώς πρέπει την μετατροπή join να την εκτελούμε σε τακτά χρονικά διαστήματα και για να έχουμε real time αναπαράσταση της εικόνας των γεγονότων που παρακολουθούμε, αλλά και για να μην μεγαλώνει αενάως το μέγεθος των δύο streams που θέλουμε να κάνουμε join.

Aggregations [17]

Η συγκεκριμένη μετατροπή λαμβάνει ένα stream εισόδου και παράγει ένα νέο συνδυάζοντας διαφορετικά δεδομένα από την είσοδο σε μια καταγραφή εξόδου. Παραδείγματα της μεθόδου είναι εφαρμογές που υπολογίζουν αθροίσματα, κρατούν μετρητές για συγκεκριμένα γεγονότα ή μέσους όρους μετρήσεων. Συνήθως, οι μετατροπές υπολογίζονται σε ένα μήκος παραθύρου έτσι ώστε να μην προκαλούνται προβλήματα με τον αποθηκευτικό χώρο. Σημαντικό είναι να αναφέρουμε ότι η έξοδος οποιασδήποτε Aggregation διεργασίας είναι ένα KTable καθώς και ότι για να εκτελέσουμε οποιαδήποτε Aggregation διεργασία πρέπει να έχουμε ομαδοποιημένα τα δεδομένα μας σε μορφή KGroupStream ή KGroupedTable.

Οι έννοιες KGroupStream και KGroupedTable δεν αποτελούν ουσιαστικά καινούργιες δομές απλά είναι οι έννοιες KStream και KTable ομαδοποιημένες με βάση τα κλειδιά των δεδομένων τους.

Παρακάτω ακολουθεί ένα διάγραμμα που παρουσιάζει πώς αλληλεπιδρούν όλα τα stateful και stateless transformations μεταξύ τους καθώς και το stream input που δέχονται και το stream output που παράγουν.



Σχήμα 7.14: Stateful and Stateless Transformations

Κεφάλαιο 8

Apache Flink

8.1 Εισαγωγή

Το Apache Flink είναι ένας κατανεμημένος επεξεργαστής ροών δεδομένων (streams) 3ης γενιάς. Προσφέρει γρήγορη επεξεργασία streams, υποσχόμενο μικρές καθυστερήσεις (latency) και μεγάλη αποκρισιμότητα (throughput) από το server. Μερικές από τις ιδιότητες που παρατηρούνται στο Apache Flink και το κάνουν να διαφέρει από τις άλλες παρόμοιες εφαρμογές είναι [24]:

1. Η δυνατότητα για event-time και processing-time ανάλυση και επεξεργασία. Πράγμα που μας επιτρέπει και την διαχείριση γεγονότων εκτός σειράς (out-of-order events).

2. Συνέπεια στην επεξεργασία κάθε δεδομένου ακριβώς μια φορά.

3. Καθυστερήσεις της τάξης του millisecond ακόμα και όταν διαχειρίζεται εκατομμύρια δεδομένα το δευτερόλεπτο, καθώς και δυνατότητα κλιμακωσιμότητας των εφαρμογών.

4. Πολλαπλές φιλικές προς τον χρήστη διεπαφές προγραμματισμού εφαρμογών (API), οι οποίες καλύπτουν πλήθος ενεργειών και πιθανών σεναρίων χρήσης. Στην παρούσα εργασία θα ασχοληθούμε αποκλειστικά με το DataStream API. Αλλά αξίζει να σημειωθεί ότι υπάρχουν και άλλα όπως τα ρελατιοναλ APIs, SQL και το LINQ-style Table API.

5. Έτοιμες συναρτήσεις, connectors, οι οποίες συνδέουν την εφαρμογή με τα πιο διαδεδομένα συστήματα αποθήκευσης όπως το Apache Kafka, το Apache Cassandra, το Elasticsearch, το JDBC και το Kinesis, καθώς και με κατανεμημένα συστήματα αρχείων όπως το HDFS και το S3.

6. Δυνατότητα να εκτελεί streaming εφαρμογές συνεχόμενα με πολύ μικρό χρόνο διακοπής και να διαθέτει μεγάλη αντοχή σε σφάλματα.

7. Δυνατότητα ενημέρωσης εφαρμογών και μετακίνησης διεργασιών σε διαφορετικά clusters.

8. Τέλος, δίνει την δυνατότητα της επεξεργασίας των γεγονότων σε κομμάτιας (batches).

Σχήμα 8.1: Apache Flink Logo

8.2 Τοπολογία του Flink

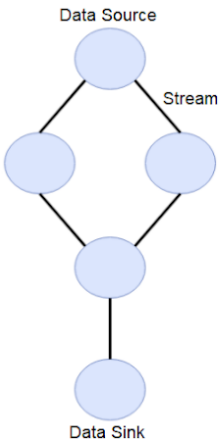
Όπως και στο Kafka Streams και εδώ έχουμε να κάνουμε με streams το οποίο είναι το βασικό κομμάτι κάθε stream processing εφαρμογής. Ο τοπολογικός γράφος του Flink για οποιοδήποτε πρόγραμμα αποτελείται από κόμβους στους οποίους γίνεται η επεξεργασία δεδομένων. Κάθε κόμβος έχει μια είσοδο από την οποία λαμβάνει το stream και μια έξοδο στην οποία εκπέμπει τα αποτελέσματα του [10].

Στην τοπολογία που αναφέραμε υπάρχουν δύο ειδικοί τύποι κόμβων:

1. **Data Sources:** Αποτελεί τον κόμβο από τον οποίο το πρόγραμμα δέχεται τα δεδομένα του. Μπορεί να διαβάζει από κάποιο αρχείο (File-based), από κάποιο socket (Socket-based), από κάποια συλλογή (Collection-based) ή από μια καθορισμένη από τον χρήστη συνάρτηση (Custom).

2. **Data Sinks:** Αποτελεί τον κόμβο που "καταναλώνει" το αποτέλεσμα όλης της επεξεργασίας και μετατροπής του αρχικού stream. Σκοπός του είναι η προώθηση των αποτελεσμάτων αυτών σε αρχεία, sockets, εξωτερικά συστήματα, χώρους αποθήκευσης ή απλά η εκτύπωση τους στην έξοδο του προγράμματος.

Η παρακάτω τοπολογία αποτελεί ένα απλό παράδειγμα αυτών που αναφέραμε. Σημειώνεται ότι σε μια τοπολογία δεν υπάρχει περιορισμός στον αριθμό Data Sources και Data Sinks.



Σχήμα 8.2: Τοπολογία Apache Flink

8.3 Αρχιτεκτονική

Το Flink όπως αναφέραμε είναι ένα κατανεμημένο σύστημα για stateful, παράλληλη επεξεργασία δεδομένων. Η τοπολογία του Flink αποτελείται από πληθώρα διεργασιών οι οποίες πολλές φορές εκτελούνται και σε διαφορετικά μηχανήματα.

8.3.1 Συνιστώσες του Flink Setup

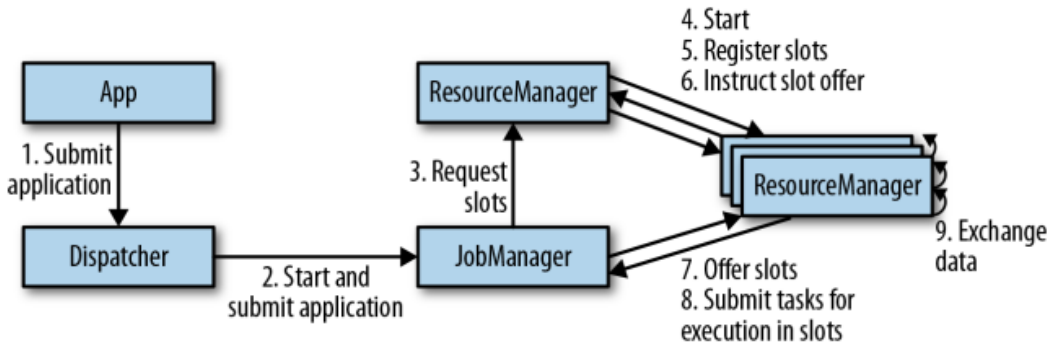
JobManager
Αποτελεί την κυρίαρχη (master) διεργασία που χειρίζεται μια εφαρμογή. Κάθε εφαρμογή την διαχειρίζεται διαφορετικός JobManager. Κάθε εφαρμογή αποτελείται από το JobGraph το οποίο είναι πρακτικά ένα γράφημα που αναπαριστά ένα διάγραμμα ροής (αναλύει ποιές είναι οι είσοδοι της εφαρμογής, τις μετατροπές που πραγματοποιεί και ποιές είναι οι έξοδοι) και ένα JAR file το οποίο περιέχει τις απαραίτητες κλάσεις και βιβλιοθήκες. Η δουλειά του JobManager είναι να μετατρέψει το JobGraph σε ένα ExecutionGraph που αποτελείται από τις διάφορες διεργασίες (tasks) που μπορούν να εκτελεστούν παράλληλα. Ο JobManager ζητά τους απαραίτητους πόρους (TaskManager slots) προκειμένου να εκτελέσει το task από τον ResourceManager. Μόλις λάβει τα απαραίτητα slots διαμοιράζει τα tasks του ExecutionGraph στον TaskManager για να τα εκτελέσει. Κατα την διάρκεια της εκτέλεσης ο JobManager είναι υπεύθυνος για οποιαδήποτε ενέργεια απαιτεί κεντρικό συντονισμό, όπως την ανάνηψη από σφάλματα μέσω checkpoints. [25]

ResourceManager
Το Flink διαθέτει πολλούς διαφορετικούς ResourceManagers για διαφορετικά περιβάλλοντα και διαφορετικούς παρόχους (προιδερ) υπηρεσιών. Όπως αναφέρθηκε είναι υπεύθυνος για την διαχείριση των TaskManager slots. Ζητά από TaskManagers να επιστρέψουν τα αδρανή slots τους στον JobManager ή ακόμα και ζητά από κάποιον πάροχο να προφέρει χώρο για επεξεργασία. Στις αρμοδιότητές του, ακόμα, είναι και ο τερματισμός των αδρανών TaskManagers, για να απελευθερώνεται χώρος. [25]

TaskManager
Αποτελούν τους τις διεργασίες που επεξεργάζονται τα δεδομένα και υπολογίζουν τις μετατροπές που υποδεικνύει το ExecutionGraph. Κάθε ένας έχει συγκεκριμένο αριθμό slots (όπως ένας επεξεργαστής υπολογιστή έχει πυρήνες) ο οποίος περιορίζει τα tasks που μπορεί να εκτελέσει ένας TaskManager. Μέσω το ResourceManager ένας JobManager λαμβάνει έναν ή παραπάνω slots στα οποία εκτελεί τις διεργασίες του. Κατα τη διάρκεια της εκτέλεσης, TaskManager που εκτελούν κομμάτια της ίδιας εφαρμογής ανταλλάσουν δεδομένα. [25]

Dispatcher
Αποτελεί ένα REST (Μεταφορά αναπαραστατικής κατάστασης) interface μέσω του οποίου ο χρήστης υποβάλει εφαρμογές με σκοπό να εκτελεστούν. Όταν υποβάλλεται μια εφαρμογή

δημιουργείται ένας JobManager που αναλαμβάνει τον παρακολούθηση της. Η δουλειά του Dispatcher πολλές φορές δεν είναι απαραίτητη. [25]



Σχήμα 8.3: Flink Setup

8.3.2 Application Deployment

Υπάρχουν δύο τρόποι μια εφαρμογή να γίνει deploy στο Flink [25]:

1. **Framework style:** Η εφαρμογή μετατρέπεται σε ένα JAR αρχείο και υποβάλλεται από τον χρήστη σε μια "υπηρεσία" εκτέλεσης. Αυτή μπορεί να είναι ο Dispatcher, ένας JobManager ή και κάποιος ResourceManager κάποιου παρόχου.

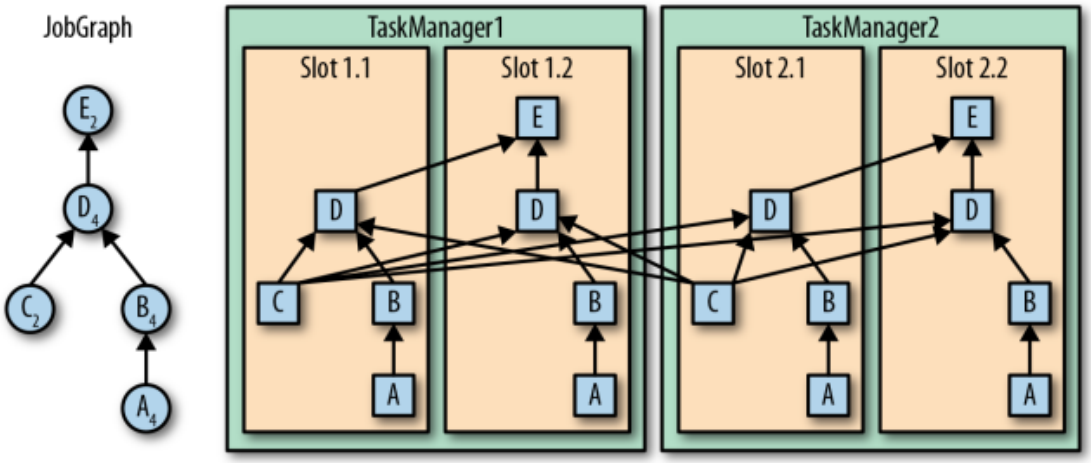
2. **Library style:** Η εφαρμογή εντάσσεται σε ένα αμετάβλητο, στατικό αρχείο που περιέχει εκτελέσιμο κώδικα (container image), όπως για παράδειγμα ένα Docker image, το οποία περιέχει και τον κώδικα για την εκτέλεση των JobManager και ResourceManager. Με την εκτέλεση του container image ξεκινούν JobManager και ResourceManager, όμως ο TaskManager εκτελείται από διαφορετική διεργασία.

Ο πρώτος τρόπος, είναι ο πιο κοινός τρόπος υποβολής μιας διεργασίας από έναν χρήστη σε μια υπηρεσία εκτέλεσης. Στην δεύτερη περίπτωση το Flink δεν αποτελεί την υπηρεσία εκτέλεσης, απλά παρέχεται ως βιβλιοθήκη μέσα στο container image.

8.3.3 Εκτέλεση Task

Ένας TaskManager μπορεί να εκτελεί αρκετά tasks την ίδια στιγμή. Αυτά μπορεί να είναι subtasks του ίδιου τελεστή (data parallelism), διαφορετικού τελεστή (task parallelism) ή ακόμα και εντελώς διαφορετικής εφαρμογής (job parallelism). Επίσης επικοινωνεί με άλλους TaskManagers και ανταλλάσει μαζί τους δεδομένα αν είναι απαραίτητο. [25]

Παρακάτω στο διάγραμμα φαίνεται η σχέση μεταξύ TaskManagers, slots, tasks, και operators.

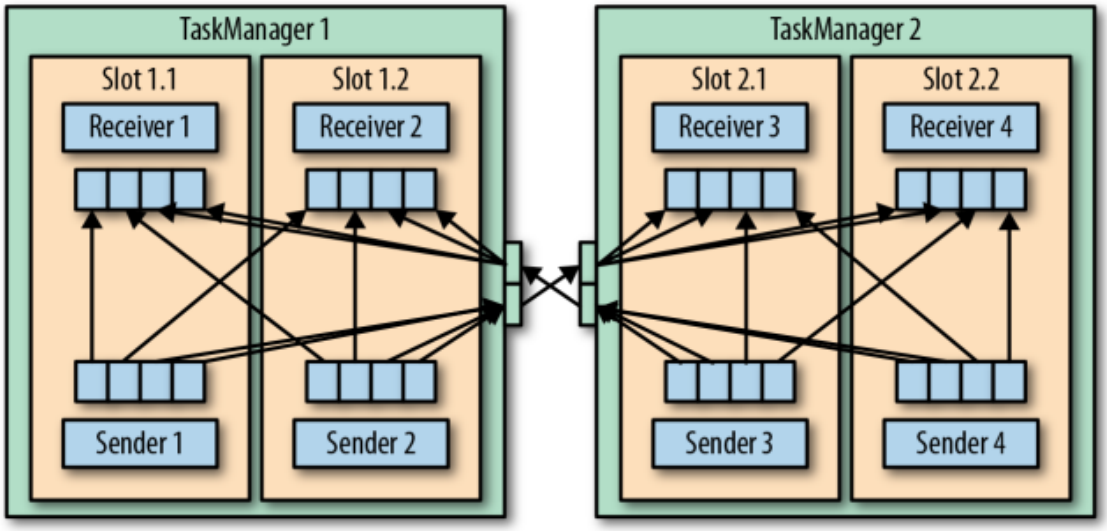


Σχήμα 8.4: Task Execution

Το job graph αποτελείται από 5 operators, οι A και C είναι είσοδοι (sources) και ο E είναι έξοδος (sink). Οι C και E έχουν παραλληλοποίηση δύο και οι υπόλοιποι παραλληλοποίηση τέσσερα. Ο μέγιστος αριθμός παραλληλοποίησης είναι το τέσσερα, συνεπώς θέλουμε το λιγότερο τέσσερα slots για να εκτελέσουμε την εφαρμογή. Στην συγκεκριμένη περίπτωση για την εκτέλεση δίνουμε στην εφαρμογή δύο TaskManagers οι οποίοι έχουν δύο slots ο καθένας. Οι operators με παραλληλοποίηση τέσσερα θα δίνονται σε όλα τα slots ενώ οι operators C και E μοιράζονται στα slots 1.1, 2.1, 1.2, 2.2 αντίστοιχα. Ο τρόπος που χωρίζουμε τα tasks ως slices σε slots επηρεάζει λίγο την συμπεριφορά της εφαρμογής. Πιο συγκεκριμένα tasks στο ίδιο TaskManager μπορούν να ανταλλάσουν δεδομένα χωρίς να επιβαρύνουν το δίκτυο (network). Παρόλα αυτά δεν είναι σωστή πρακτική να υπερφορτώνεται ένας TaskManager με πολλά tasks διότι γίνεται πιο ευάλωτος σε σφάλματα και μειώνεται η απόδοση του, συνεπώς πρέπει εκεί που είναι εφικτό τα tasks να μοιράζονται ανάλογα σε όλους τους διαθέσιμους πόρους. Η εκτέλεση των tasks γίνεται με πολυνηματισμό (multithreading) μέσα στον TaskManager και όχι σε ξεχωριστές διεργασίες, λόγω του ότι τα νήματα threads είναι πιο "ελαφριά" και επικοινωνούν με ευκολότερο τρόπο μεταξύ τους.

8.3.4 Επικοινωνία TaskManagers

Παρακάτω στο διάγραμμα φαίνεται η επικοινωνία των TaskManagers μέσω δικτύου (network) αλλά και των slots του ίδιου TaskManager. Κάθε TaskManager στέλνει και λαμβάνει δεδομένα από άλλους TaskManagers μέσω network buffers ενώ η επικοινωνία μέσα στον ίδιο TaskManager γίνεται με απλό byte buffer και όπως αναφέραμε και παραπάνω δεν επιβαρύνει το δίκτυο. Η επικοινωνία σε επίπεδο δικτύου πραγματοποιείται με πρωτόκολλο TCP. [25]



Σχήμα 8.5: Επικοινωνία μεταξύ των TaskManagers

8.3.5 Fault Tolerance

Οι εφαρμογές που εκτελούνται στο Flink τις περισσότερες φορές αποτελούν streaming εφαρμογές που είναι απαραίτητο να είναι διαθέσιμες 24 ώρες το 24ωρο. Γι αυτό το λόγο η ανάνηψη από σφάλματα αποτελεί σημαντικό παράγοντα για το Flink. Όταν μια εφαρμογή σταματήσει να λειτουργεί πρέπει να επανεκκινήσει, να βρεθεί σε τι στάδιο είχε φτάσει και μετά να συνεχίσει από εκεί. Γενικά μπορεί να προκύψει σφάλμα είτε στον JobManager είτε στον TaskManager. [25]

Σφάλματα TaskManager

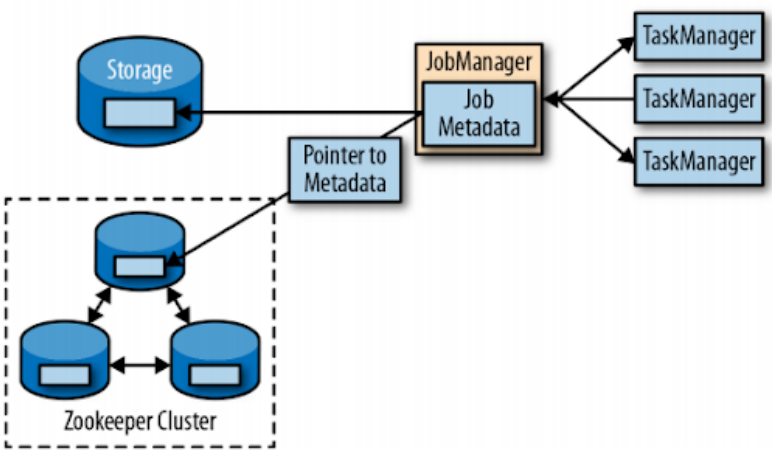
Όπως αναφέραμε, σε εφαρμογές που απαιτούν παραλληλοποίηση χρειάζεται συγκεκριμένος αριθμός slots και TaskManagers. Για παράδειγμα, αν έχουμε εφαρμογή που απαιτεί παραλληλοποίηση 6 και έχουμε 3 TaskManagers με 2 slots ο καθένας μπορούμε να εκτελέσουμε την εφαρμογή, σε περίπτωση όμως που ένας TaskManager αποτύχει για οποιοδήποτε λόγο, ολόκληρη η εφαρμογή σταματά. Είναι έπειτα, αρμοδιότητα του JobManager να ζητήσει από τον ResourceManager για τους πόρους που χρειάζεται και να επανεκκινήσει την διαδικασία, οι επανεκκινήσεις και η επικοινωνία με τον ResourceManager γίνονται ανάλογα με την στρατηγική επανεκκίνησης που έχει επιλεχθεί από την εφαρμογή.

Σφάλματα JobManager

Ο JobManager αποτελεί τον συντονιστή της εφαρμογής και κρατάει την κατάσταση, την πρόοδο που σημειώνει η εφαρμογή, σημαντικά metadata και τα checkpoints (σημεία από τα οποία μπορεί να εκκινηση σε περίπτωση σφάλματος). Από τα παραπάνω καταλαβαίνουμε ότι ο JobManager αποτελεί το single point of failure (μέρος ενός συστήματος που, εάν αποτύχει, θα σταματήσει να λειτουργεί ολόκληρο το σύστημα), συνεπώς το Flink έχει λειτουργία υ-

ψηλής διαθεσιμότητας που σε περίπτωση σφάλματος μεταφέρει τις απαραίτητες πληροφορίες που αναφέρθηκαν σε έναν καινούργιο JobManager. Αυτή η λειτουργία είναι βασισμένη στον Apache Zookeeper, ο οποίος είναι υπεύθυνος να κρατάει πληροφορίες για την θέση όλων των απαραίτητων στοιχείων για την εκτέλεση της υπηρεσίας Flink. Τέτοια στοιχεία είναι, για παράδειγμα, οι χώροι αποθήκευσης των μεταβλητών της εφαρμογής, το JobGraph, τα JAR αρχεία, τα αντίστοιχα states στα οποία η εφαρμογή σταμάτησε και τα checkpoint της εφαρμογής. Σημειώνεται ότι ο Apache Zookeeper είναι ο ίδιος που χρησιμοποιείται και στο Apache Kafka και για αυτόν τον λόγο δεν αναλύεται παραπάνω εδώ.

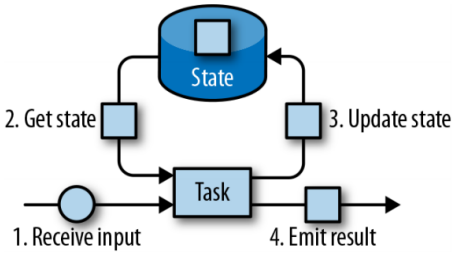
Παρακάτω στο σχήμα 8.6 παρουσιάζεται η τοπολογία ενός Flink setup με Zookeeper cluster.



Σχήμα 8.6: Flink Setup with Zookeeper

Συνοψίζοντας:

Κάθε Task το οποίο κάνει οποιαδήποτε μετατροπή στο stream έχει την παρακάτω μορφή (σχήμα 8.7) κρατώντας την κατάσταση του (state) έτσι ώστε να μπορεί να σώζει την πρόοδο του.



Σχήμα 8.7: Task Diagram

Στη συνέχεια, το Flink είναι υπεύθυνο να κρατά με την μορφή checkpoints ανά τακτά χρονικά διαστήματα όλες αυτές τις πληροφορίες από όλα τα task έτσι ώστε αν υπάρξει κάποιο σφάλμα να μπορεί να επανεκκινήσει. Το πόσο συχνά ανανεώνουμε τα checkpoints ρυθμίζεται

με παραμέτρους μέσα από την εκάστοτε εφαρμογή. Η πολιτική που ακολουθεί το Flink για την δημιουργία checkpoints είναι βασισμένη στον αλγόριθμο *Chandy–Lamport algorithm for distributed snapshots [33]*, και δεν χρειάζεται διακοπή της εκτέλεσης για την αποθήκευση του νέου checkpoint.

8.4 Βασικές έννοιες του Flink

8.4.1 Χρόνος (Time)

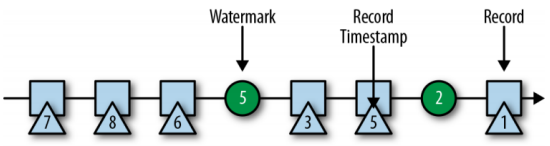
Όπως αναφέραμε και στο κεφάλαιο του Kafka Streams ο χρόνος είναι σημαντική έννοια όταν μιλάμε για εφαρμογές που επεξεργάζονται ροή δεδομένων. Και σε αυτήν την περίπτωση ξεχωρίζουμε το processing time το οποίο είναι ο χρόνος που επεξεργάζονται τα δεδομένα από την εφαρμογή, και το event time που είναι ο πραγματικός χρόνος που δημιουργείται ένα δεδομένο. Αν και το processing time είναι πιο εύκολο να κατανοηθεί λόγο του ότι είναι βασισμένο στην ώρα του τοπικού μηχανήματος, καλύτερη τακτική είναι να βασιζόμαστε στο event time όταν διαχειριζόμαστε δεδομένα διότι παράγουν συνεπή αποτελέσματα. Το Flink προσφέρει ειδικές διεργασίες στον χρήστη με τις οποίες μπορεί να διαχειριστεί το event time με την βοήθεια **timestamps** και **Watermarks**.

Timestamps

Όλα τα γεγονότα που χρησιμοποιούν την event time επεξεργασία πρέπει να περιέχουν ένα timestamps δηλαδή έναν τρόπο να συνδέονται με την χρονική στιγμή με την οποία το γεγονός δημιουργήθηκε. Ανάλογα με την εφαρμογή αυτή η ένδειξη μπορεί να είναι ένας χρόνος ρολογιού ένας απλός αριθμός σειράς ή ό, τι άλλο χρειάζεται, το μόνο περιοριστικό είναι ότι τα timestamps πρέπει να αυξάνονται όσο το stream προχωράει. Σε real time εφαρμογές, συνηθίζεται να υπάρχει και μια ελαστικότητα στην επεξεργασία για γεγονότων που βρίσκονται εκτός χρονικής σειράς (out-of-orderness), όταν αυτή η διαφορά είναι κάτω από ένα όριο που ορίζει η εφαρμογή. [25]

Watermarks

Σε συνδυασμό με τα timetamps μια εφαρμογή πρέπει να παρέχει και τα κατάλληλα Watermarks. Το Watermark είναι ουσιαστικά ένας global μετρητής που υποδεικνύει την χρονική στιγμή την οποία είμαστε πεπεισμένοι ότι δεν θα λάβουμε άλλα γεγονότα εκτός σειράς. Επί της ουσίας, τα Watermarks παρέχουν ένα λογικό ρολόι που ενημερώνει το σύστημα για το event time της εφαρμογής. Στο Flink τα Watermarks αποτελούν μια Long τιμή. [25]



Σχήμα 8.8: Watermark

Όταν λάβουμε το Watermark 5 γνωρίζουμε ότι δεν πρόκειται να έρθει δεδομένο με timestamp μικρότερο από 5.

Δύο βασικές ιδιότητες των Watermarks είναι ότι:

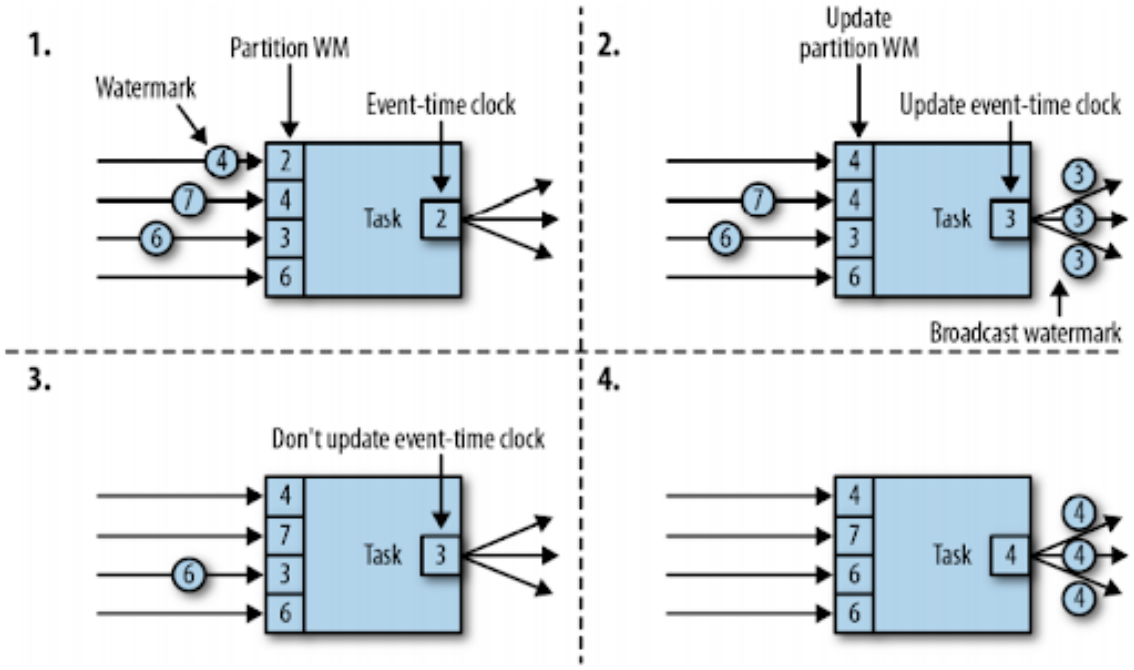
1. Πρέπει όπως και τα timestamps να αυξάνονται όσο προχωράμε μπροστά στο stream.
2. Συνδέονται και προκύπτουν από τα επιμέρους timestamps των δεδομένων. Αν λάβουμε ένα Watermark T τότε με μεγάλη βεβαιότητα μπορούμε να πούμε ότι τα επόμενα timestamps δεδομένων που θα λάβουμε, θα είναι μεγαλύτερα από T.

Τρόπος ενημέρωσης Watermarks σε Task [25]

Όταν ένα Task λαμβάνει ένα Watermark εκτελούνται οι παρακάτω ενέργειες:

- Το Task ανανεώνει το εσωτερικό του ρολόι στην τιμή του timestamp του Watermark.
- Εντοπίζονται όλοι οι μετρητές με τιμή μικρότερη από το timestamp, γι κάθε έναν από αυτούς το task καλεί μια callback συνάρτηση και εκτελεί μετατροπές και εκπέμπει καταγραφές.
- Το Task εκπέμπει ένα Watermark με την ανανεωμένη τιμή.

Στο παρακάτω διάγραμμα φαίνεται η διαδικασία που περιγράψαμε για ένα Τασκ που εκτελεί παράλληλη επεξεργασία.



Σχήμα 8.9: Watermark Update in Task

Δημιουργία Watermarks και Timestamps [25]

Τα Timestamps και Watermarks συνήθως κατασκευάζονται όταν το stream δοθεί ως είσοδος στην εφαρμογή. Γι αυτό το λόγο ο χρήστης στην εφαρμογή είναι υπεύθυνος για την δημιουργία και την ανάθεση τους. Μπορεί πολλές φορές να χρησιμοποιεί πληροφορία που βρίσκεται στα δεδομένα γι αυτήν την ανάθεση. Η ανάθεση Timestamps και Watermarks γίνεται με τρεις τρόπους:

1. **Στην πηγή (source):** Στην συνάρτηση που καταναλώνει τα δεδομένα στην είσοδο της εφαρμογής.
2. **Periodic assigner:** Συνάρτηση που καθορίζεται από τον χρήστη, ανά τακτά χρονικά διαστήματα λαμβάνει το παρόν Watermark και το εκπέμπει.
3. **Punctuated assigner:** Συνάρτηση που καθορίζεται από τον χρήστη, παράγει watermarks από πληροφορίες που είναι εμφωλευμένες στα δεδομένα της εισόδου.

Συνήθως αναθέτουμε Watermarks και Timestamps όσο πιο κοντά στην είσοδο για να μην επηρεάζονται από τους χρόνους επεξεργασίας και αποφεύγουμε να αλλάζουμε τις τιμές τους κατά την διάρκεια της εκτέλεσης της εφαρμογής και των μετατροπών.

8.4.2 DataStream

Το DataStream είναι μια ειδική κλάση η οποία χρησιμοποιείται για να απεικονίσει μια συλλογή από δεδομένα σε ένα πρόγραμμα Flink. Είναι πρακτικά ένα αμετάβλητο σύνολο από δεδομένα που μπορεί να περιέχει και αντίγραφα, τα δεδομένα αυτά μπορεί να είναι πεπερασμένα ή μη. Βασικά χαρακτηριστικά είναι ότι πρώτον είναι αμετάβλητα, δηλαδή, από την στιγμή που κάτι προστεθεί στο stream δεν μπορεί να αφαιρεθεί και δεύτερον ότι δεν μπορούμε να εξετάσουμε μεμονωμένες εγγραφές στο stream παρά μόνο να τις επεξεργαστούμε μέσω των μετατροπών (operations) που βρίσκονται στο DataStream API. [25]

8.5 Μετατροπές και Συναρτήσεις

Παρακάτω παρουσιάζονται οι μετατροπές και συναρτήσεις που προσφέρει το DataStream API του Flink.

8.5.1 Βασικές μετατροπές [26, 11]

Transformation	Description
Map • DataStream → DataStream	Δέχεται μια καταγραφή και παράγει ακριβώς μια άλλη, χρησιμοποιείται κυρίως για την αλλαγή κλειδιου και τιμής καθώς και του τύπου των δεδομένων.
Filter • DataStream → DataStream	Φιλτράρει το KStream ή το KTable απορρίπτοντας κάποιες τιμές με βάση δοσμένους κανόνες. Το ποιά δεδομένα θα απορροφηθούν βασίζεται στο αποτέλεσμα μιας λογικής συνάρτησης που καθορίζει ο χρήστη στην εφαρμογή.
FlatMap • DataStream → DataStream	Μοιάζει με την μετατροπή Map, η μόνη διαφορά της είναι ότι αντί για ακριβώς ένα, μπορεί να εκπέμπει στην έξοδο καμία, μία ή ακόμα και περισσότερες καταγραφές.

Σχήμα 8.10: Basic Transformations

8.5.2 KeyedStream μετατροπές

Το KeyedStream είναι ουσιαστικά ένα DataStream το οποίο έχει χωριστεί σε υπο-stream τα οποία μοιράζονται το ίδιο κλειδί. Σε κάποιες εφαρμογές που η επεξεργασία των γεγονότων πρέπει να γίνεται με βάση κάποιο χαρακτηριστικό τους είναι απαραίτητη η χρήση KeyedStreams. Κάποιες από τις βασικές KeyedStream μετατροπές είναι οι παρακάτω [26, 11]:

Transformation	Description
KeyBy • DataStream → KeyedStream	Δέχεται μια καταγραφή και παράγει μια άλλη βασιζόμενη στα κλειδιά των δεδομένων. Με βάση τα κλειδιά τα γεγονότα του stream χωρίζονται αντίστοιχα σε partitions, έτσι ώστε τα γεγονότα με το ίδιο κλειδί να επεξεργάζονται από το ίδιο Task.
Aggregations • KeyedStream → DataStream	Τα rolling aggregations εφαρμόζονται πάνω σε KeyedStreams και παράγουν συγκεντρωτικά αποτελέσματα (aggregates) όπως το άθροισμα (sum) και η εύρεση μεγίστου, ελαχίστου στοιχείου ή πεδίου στοιχείου.
Reduce • KeyedStream → DataStream	Η reduce μετατροπή αποτελεί γενίκευση των rolling aggregations. Αποτελείται από μια reduce συνάρτηση που συνδυάζει την τιμή που δέχεται στην είσοδο με την reduced τιμή που έχει από προηγούμενα γεγονότα και παράγει στην έξοδο ένα DataStream ίδιου τύπου με το DataStream εισόδου.

Σχήμα 8.11: KeyedStream Transformations

8.5.3 Multistream μετατροπές

Πολλές εφαρμογές δέχονται παραπάνω από ένα stream στην είσοδο τα οποία πρέπει πολλές φορές να επεξεργαστούν ενωμένα ή δέχονται ένα stream το οποίο πρέπει να χωριστεί σε διαφορετικά υπο-stream τα οποία πρέπει να επεξεργαστούν με διαφορετικό τρόπο. Γι αυτό τον λόγο έχουμε τις μετατροπές multistream οι οποίες δέχονται ένα ή περισσότερα streams ως είσοδο και εκπέμπουν στην έξοδο ένα ή περισσότερα streams. Κάποιες από τις βασικές είναι οι παρακάτω [26, 11]:

Transformation	Description
Union • DataStream* → DataStream Το "*" υποδηλώνει την ύπαρξη ενός ή περισσότερων DataStream	Δέχεται δύο ή περισσότερα streams στην είσοδο και τα συγχωνεύει σε ένα stream. Τα γεγονότα συγχωνεύονται με λογική FIFO (First-In-First-Out) και δεν απαλείφουν τα διπλότυπα (duplicates) γεγονότα.
Connect • DataStream,DataStream → ConnectedStreams	Ενώνει δύο data streams κρατώντας τους τύπους τους. Η συγκεκριμένη ενέργεια επιτρέπει την αποθήκευση της προόδου και των δύο stream σε κοινό state και βοηθά πολύ στην περίπτωση ανάνηψης από σφάλματα.
Comap, Coflatmap • ConnectedStreams → DataStream	Παρόμοιο με το map και το flatmap που αναφέραμε παραπάνω απλά αναφέρεται σε connected streams.
Split and Select • DataStream → DataStream* Το "*" υποδηλώνει την ύπαρξη ενός ή περισσότερων DataStream	Αποτελεί την ανάποδη διαδικασία από την μετατροπή Union που περιγράψαμε. Δέχεται ένα stream και το χωρίζει σε δύο ή περισσότερα υπο-streams ίδιου τύπου. Κάθε γεγονός του αρχικού stream μπορεί να οδηγηθεί σε ένα, περισσότερα ή και κανένα υπο-stream εξόδου. Συνεπώς, αυτή η μετατροπή μπορεί να χρησιμοποιηθεί για να φιλτράρει ή και να επαναλαμβάνει γεγονότα.

Σχήμα 8.12: Multistream Transformations

8.5.4 Window μετατροπές

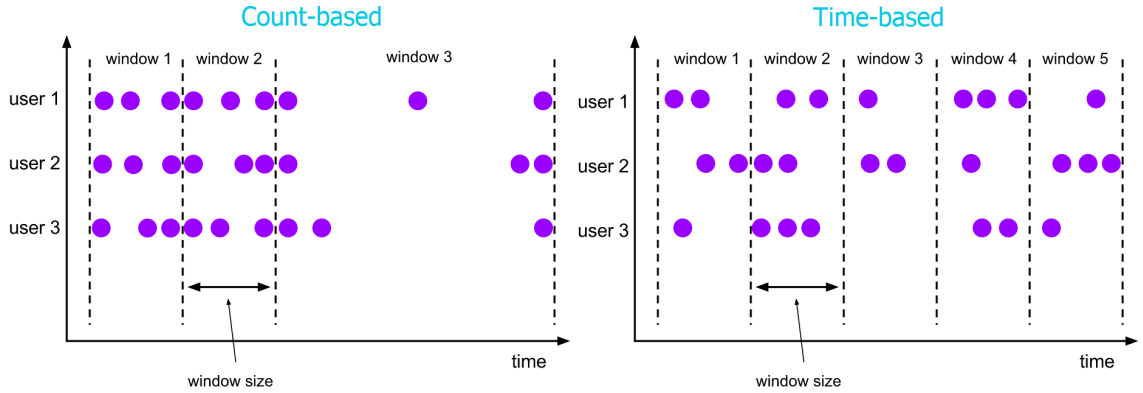
Μετατροπές σαν αυτές που παρουσιάστηκαν παραπάνω επεξεργάζονται ένα γεγονός την φορά και εκπέμπουν ένα γεγονός στην έξοδο ενημερώνοντας και το state του task. Παρόλα αυτά κάποιες φορές κάποιες ενέργειες απαιτούν ένα σύνολο από γεγονότα για να υπολογίσουν τα αποτελέσματά τους.

Αν για παράδειγμα, η εφαρμογή μας υπολογίζει τον μέσο όρο των δεδομένων εισόδου σε ένα απεριόριστο stream τότε θα πρέπει να περιορίσουμε τον αριθμό των δεδομένων που συγκρατούνται. Πολλές φορές μας ενδιαφέρει να μετράμε δεδομένα για μια δοσμένη χρονική διάρκεια. Έστω ότι, η εφαρμογή μας που υπολογίζει μέσο όρο, χρησιμοποιούνταν για να μετράει τις θερμοκρασίες σε μια πόλη, η πληροφορία του μέσου όρου της θερμοκρασίας για συγκεκριμένα χρονικά διαστήματα (ώρες μέρες, εβδομάδες) έχει συνήθως μεγαλύτερη αξία α-πό τον μέσο όρο της θερμοκρασίας συνολικά, από την στιγμή που αρχίσαμε να λαμβάνουμε τα πρώτα μας δεδομένα. Χρειαζόμαστε λοιπόν να έχουμε μια συνεχή δημιουργία πεπερασμένων

streams δεδομένων που προέρχονται από τον κατακερματισμό ενός μεγάλου απεριόριστου stream. Μπορούμε να σκεφτούμε αυτά τα πεπερασμένα stream σα "πλαίσια" στα οποία αναπτύθεται γεγονότα με βάση κάποια χαρακτηριστικά των δεδομένων τους ή συχνότερα με βάση τον χρόνο. Όταν ένα παράθυρο κλείνει, τότε στο σύνολο των δεδομένων του γίνονται οι καθορισμένοι από την εφαρμογή υπολογισμοί. Ένα παράθυρο "κλείνει", ανάλογα με τον τύπο του, για διάφορους λόγους, όπως θα εξετάσουμε παρακάτω. [5, 12]

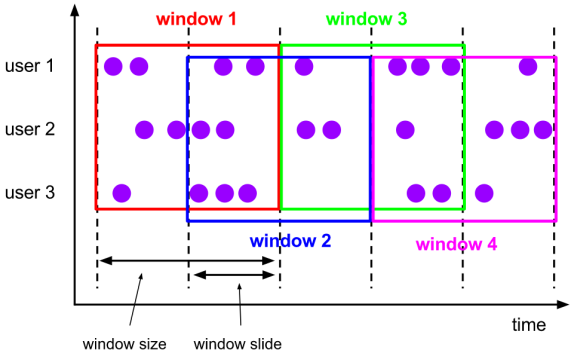
Τύποι παραθύρων:

1. **Tumbling Windows:** Αναθέτουν δεδομένα σε μη επικαλυπτόμενα παράθυρα συγκεκριμένου μήκους. Όταν το σύνορο του παραθύρου ξεπεραστεί, το παράθυρο "κλείνει" και εκτελούνται οι υπολογισμοί. Το σύνορο αυτό αποτελεί ένα χρονικό όριο για την περίπτωση των time-based tumbling windows ή έναν αριθμό εισαχθέντων δεδομένων για την περίπτωση των count-based tumbling windows.



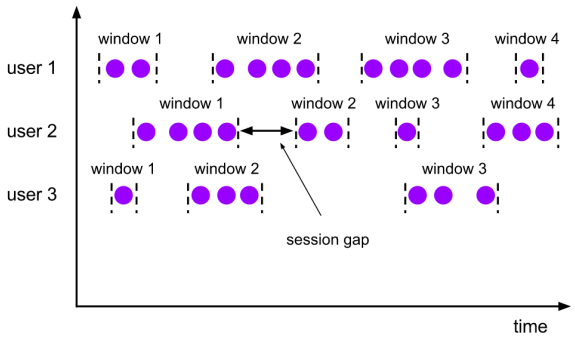
Σχήμα 8.13: Tumbling Window

2. **Sliding Windows:** Αναθέτουν δεδομένα σε επικαλυπτόμενα παράθυρα συγκεκριμένου μήκους. Με αυτόν τον τόπο ένα γεγονός μπορεί να ανήκει σε παραπάνω από ένα παράθυρα. Τα sliding παράθυρα ορίζονται με βάση το μήκος τους και την χρονική μετατόπιση τους (slide). Το παρακάτω παράθυρο έχει μήκος 10 λεπτά και slide 5 λεπτά.



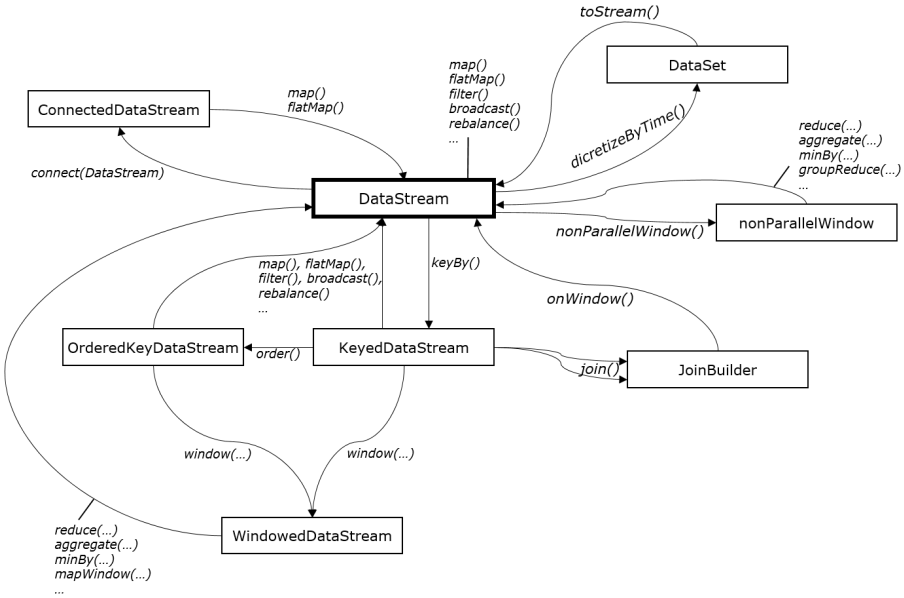
Σχήμα 8.14: Sliding Window

3. **Session Windows:** Ομαδοποιούν τα δεδομένα με βάση περιόδους απασχόλησης (activity). Τα session windows είναι μη επικαλυπτόμενα και δεν έχουν σταθερό μήκος. Το παράθυρο "κλείνει" και εκτελεί τους υπολογισμούς του με τα δεδομένα που διαθέτει, όταν δεν δέχεται νέα δεδομένα για κάποιο χρονικό διάστημα, όταν δηλαδή περνά ένα συγκεκριμένο χρονικό όριο από λήψη του τελευταίου στοιχείου (inactivity gap). Εκτός από ένα στατικό inactivity gap, ο χρήστης μπορεί να αναθέσει την δικιά του συνάρτηση για τον υπολογισμό κάθε φορά του inactivity gap. Όταν έχουμε έλευση νέου δεδομένου και το προηγούμενο παράθυρο έχει κλείσει λόγω αδράνειας, δημιουργείται καινούργιο session window.



Σχήμα 8.15: Session Window

Παρακάτω ακολουθεί ένα διάγραμμα που παρουσιάζει πώς αλληλεπιδρούν όλα τα transformations μεταξύ τους στο Apache Flink.



Σχήμα 8.16: FLink Transformations

Κεφάλαιο 9

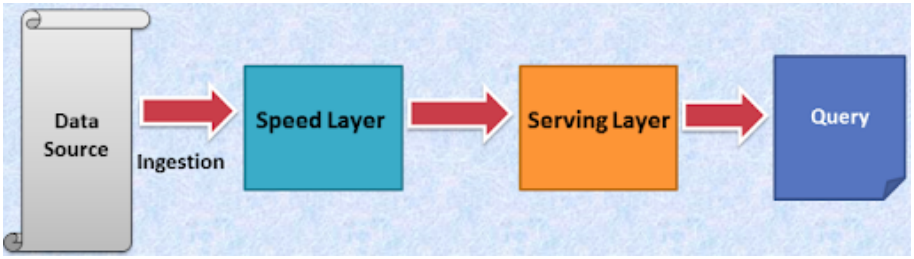
Παρουσίαση Επιλεγμένης Αρχιτεκτονικής

9.1 Εισαγωγή

Στην παρούσα εργασία, εφόσον μελετήθηκαν εκτενώς οι τεχνολογίες και τα πρωτόκολλα που παρουσιάστηκαν στα προηγούμενα κεφάλαια κατασκευάσαμε από την αρχή ένα σύστημα το οποίο αποτελείται από διαφορετικά επίπεδα (layers) τα οποία συνεργάζονται και επικοινωνούν προκειμένου, μέσω μεταφοράς streams να επιτύχουν κάποιους σκοπούς και να δημιουργήσουν μια συνολική δομή συστήματος η οποία να μπορεί να δημιουργεί, να δέχεται, να επεξεργάζεται, να αποθηκεύει και να οπτικοποιεί δεδομένα σε πραγματικό χρόνο (real time).

9.2 Αρχιτεκτονική

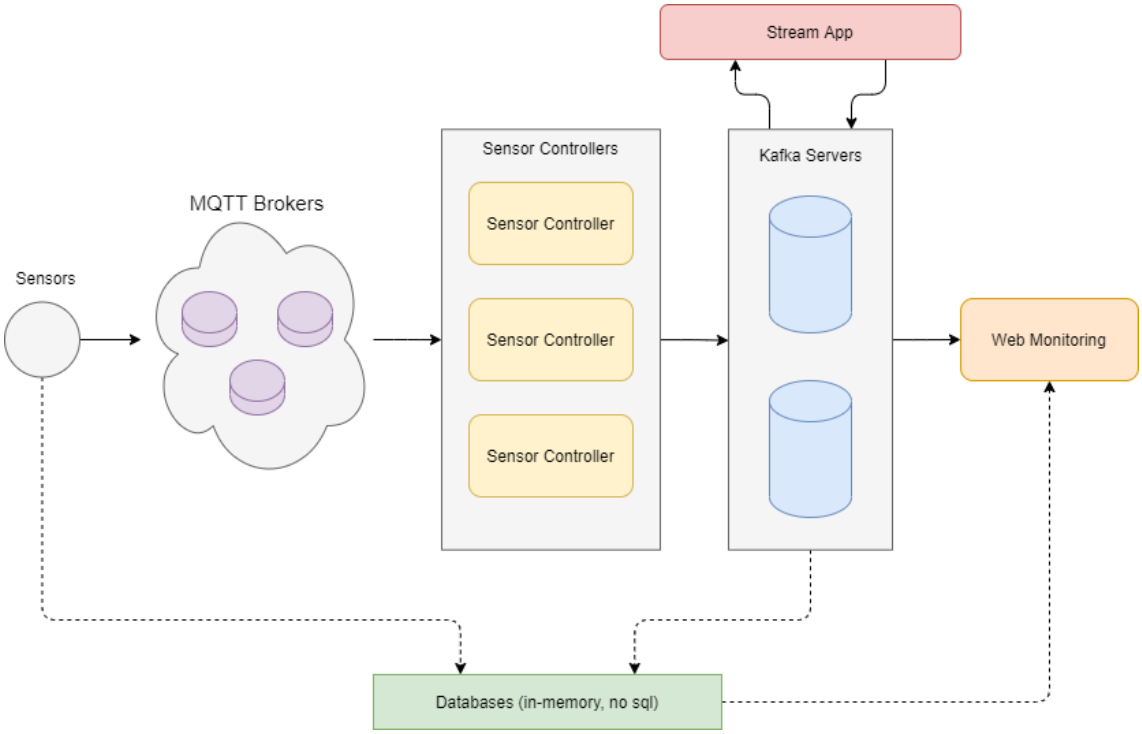
Η κατασκευή του συστήματος βασίστηκε στα πρότυπα της Kappa αρχιτεκτονικής. Η Kappa αρχιτεκτονική είναι μια διάταξη αρχιτεκτονικής λογισμικού, η οποία αντί να βασίζεται και να χρησιμοποιεί σχεσιακές βάσεις δεδομένων (relational databases) ή key-value αποθηκευτικές πρακτικές όπως το Apache Cassandra, ακολουθεί ένα σύστημα που προσθέτει τιμές σε ένα αμετάβλητο αρχείο καταγραφής. Έπειτα, από αυτό το αρχείο καταγραφής, τα δεδομένα διανέμονται μέσω συγκεκριμένων εργαλείων σε βοηθητικούς αποθηκευτικούς χώρους με σκοπό να χρησιμοποιηθούν από τους χρήστες. [38]



Σχήμα 9.1: Kappa Architecture

Η Kappa αρχιτεκτονική είναι ουσιαστικά μια απλουστευμένη μορφή της Lamda αρχιτεκτονικής. Η Lamda αρχιτεκτονική είναι μια αρχιτεκτονική για επεξεργασία δεδομένων σχεδιασμένη για να χειρίζεται μεγάλο όγκο δεδομένων χρησιμοποιώντας τεχνικές batch processing και stream processing. Με αυτόν τον τρόπο καταφέρνει να ισορροπήσει την καθυστέρηση, την ικανότητα διαμεταγωγής (throughput) και την ανοχή σε σφάλματα. Όπως και η Kappa αρχιτεκτονική αποτελείται από διαφορετικά επίπεδα που είναι υπεύθυνα για διάφορες λειτουργίες. Τα τρία επίπεδα είναι το Batch layer, το Speed layer και το Serving layer. Το Batch layer είναι υπεύθυνο για την προεπεξεργασία και τον χωρισμό των δεδομένων σε πακέτα. Το Speed layer είναι υπεύθυνο για την επεξεργασία των δεδομένων και την εκτέλεση των data processing εφαρμογών. Το Serving layer είναι υπεύθυνο για την παρουσίαση και την αποθήκευση των αποτελεσμάτων του Speed layer [7]. Όπως φαίνεται και από την παραπάνω ανάλυση η Kappa αρχιτεκτονική είναι μια Lamda αρχιτεκτονική από την οποία το batch layer έχει αφαιρεθεί και τα δεδομένα προωθούνται κατευθείαν στο speed layer για λόγους απλούστευσης και ευκολότερης διατήρησης του κώδικα της εφαρμογής.

Με το παραπάνω πρότυπο της Kappa εφαρμογής κατασκευάσαμε το σύστημά μας. Παρακάτω παρουσιάζεται σε σχεδιάγραμμα (σχήμα 9.2) και αναλύεται εκτενώς στα υποσυστήματά του.



Σχήμα 9.2: Proposed Architecture

Σημείωση: Τα διακεκομμένα κομμάτια δεν έχουν υλοποιηθεί σε βάθος, απλά αναφέρονται στο σχήμα για λόγους πληρότητας του συστήματός μας.

9.2.1 Data Source

Το Data Source επίπεδο μας αποτελείται κυρίως από τους αισθητήρες οι οποίοι παράγουν δεδομένα οποιασδήποτε μορφής σε πραγματικό χρόνο. Αυτά τα δεδομένα στέλνονται στη βαθμίδα των sensor controllers σύμφωνα με το πρωτόκολλο MQTT, μέσω των MQTT brokers και αποθηκέυονται σε βάση δεδομένων για να επανασταλόυν αν τυχόν χαθούν. Λόγω του μεγάλου αριθμού των sensors το πιο πιθανό είναι σε αυτό το σημείο να μην υπάρχει ένας μόνο MQTT broker αλλά να χρησιμοποιείται ένα cluster από brokers με την βοήθεια open source λογισμικών όπως το RabbitMQ έτσι ώστε να μην επιβαρύνουμε έναν μόνο broker με τον διαμοιρασμό της πληροφορίας.

Επόμενη στην σειρά του data source επιπέδου είναι η βαθμίδα που είναι υπεύθυνη να λαμβάνει τα δεδομένα από τους αισθητήρες κάνοντας subscribe στα topics που αυτοί εκπέμπουν και να τα γράφει-προωθεί με την σειρά της σε έναν Kafka Server, ο οποίος όπως και το κομμάτι του MQTT μπορεί να αποτελείται από πολλά clusters ή ακόμα να είναι και σύνολο διακομιστών (servers) και όχι μόνο ένας. Ο Kafka Server είναι το τελευταίο κομμάτι αυτού του επιπέδου, και αποτελεί το σημείο από το οποίο ξεκινάνε τα δεδομένα να παρουσιάζονται ως ένα αμετάβλητο αρχείο με την μορφή ροής δεδομένων (data stream). Το αποτέλεσμα του παρόντος επιπέδου, τίθεται ως είσοδος, στο επόμενο επίπεδο στην αλυσίδα, το οποίο είναι το speed layer.

9.2.2 Speed layer

Το Speed layer επίπεδο είναι πρακτικά αυτό που διαφοροποιεί την λειτουργία και τον σκοπό κάθε εφαρμογής. Κάθε σύστημα που ασχολείται με την επεξεργασία μετρήσεων αισθητήρων, ότι συναρτήσεις και αν εκτελεί στα δεδομένα, ακολουθεί την παραπάνω αλυσίδα βαθμίδων. Μόνη αλλαγή αποτελεί για τις διάφορες εφαρμογές που χρησιμοποιεί να αλλάζει τον κώδικα του Steam App επιπέδου. Το Speed layer, λοιπόν, δέχεται ένα αμετάβλητο stream από το προηγούμενο επίπεδο Data Source το οποίο το προωθεί στην data streaming εφαρμογή που έχει επιλεχθεί κάθε φορά. Η εφαρμογή μπορεί να βασίζεται σε Spark Streaming λογική η οποία αναλύθηκε σε προηγούμενο κεφάλαιο και να χρησιμοποιεί batches, μπορεί να χρησιμοποιεί λογική Kafka Streams, μπορεί να χρησιμοποιεί λογική Apache Flink ή μπορεί να χρησιμοποεί πολλές διαφορετικές εφαρμογές διαφορετικών τεχνολογιών. Σε όλες τις παραπάνω περιπτώσεις δίνεται η δυνατότητα μέσω της εφαρμογής ο χρήστης να εκτελέσει στα δεδομένα όλες τις συναρτήσεις που αναφέρθηκαν στα κεφάλαια που αναλύθηκε κάθε τεχνολογία. Μπορεί, λοιπόν, να εκτελέσει αθροίσματα σε δεδομένα, μέσους όρους, φιλτράρισμα, joins και γενικά να μετασχηματίσει τα δεδομένα του όπως χρειάζεται. Παρακάτω, θα αναλύσουμε στο δικό μας σενάριο πως αντιμετωπίζουμε τα δεδομένα μας. Τελευταίο κομμάτι του speed layer, αποτελεί και πάλι ο Kafka Server στον οποίο αποθηκεύονται τα δεδομένα στην τελευταία τους μορφή. Όπως και την προηγούμενη φορά που συναντήσαμε Kafka Server στην αρχιτεκτονική μας, αυτός μπορεί να αποτελείται από πολλούς διαφορετικούς clusters ή και από ένα σύνολο διακομιστών (servers) και όχι μόνο έναν. Από αυτόν τον Kafka Server, αντλεί τα δεδομένα του το Serving layer.

9.2.3 Serving layer

Σε αυτό το κομμάτι τα δύο του στοιχεία, παρουσίαση και αποθήκευση, μπορούν να τοποθετηθούν παράλληλα αλλά και σε σειρά χωρίς να επηρεάζει κάπως η σχετική σειρά την λειτουργία του συστήματος. Το παρόν επίπεδο είναι ουσιαστικά υπεύθυνο για την παρουσίαση και την αποθήκευση σε βάση δεδομένων των αποτελεσμάτων. Στην περίπτωση της παρουσίασης μπορεί να έχουμε μία απλή web εφαρμογή που δείχνει σε πραγματικό χρόνο τις αλλαγές και παρακολουθεί την ροή των δεδομένων. Το κομμάτι της αποθήκευσης των δεδομένων μπορεί να αποτελείται από οποιαδήποτε βάση δεδομένων. Μια καλή επιλογή είναι μια in memory key-value database που μπορεί να χρησιμοποιηθεί και σαν κατανεμημένο σύστημα. Ένα τέτοιο παράδειγμα είναι μια βάση Redis, η οποία μπορεί να χρησιμοποιηθεί σαν ιστορικό αλλά και για να αποθηκεύει τα αποτελέσματα του συστήματος με σκοπό να γίνονται ερωτήσεις (queries) πάνω σε αυτά από χρήστες ή και από άλλες εφαρμογές. Παρόλα αυτά, μπορεί να χρησιμοποιηθεί όποιου είδους βάση προτιμάται από τον χρήστη.

9.3 Παρουσίαση του δικού μας σεναρίου και αντιμετώπισή του

Η παραπάνω αρχιτεκτονική κατασκευάστηκε με σκοπό την διαχείριση ενός κτιρίου. Το σενάριο που έχουμε είναι το εξής:

Διαθέτουμε ένα κτίριο το οποίο έχει αρκετούς ορόφους και διαφορετικά δωμάτια σε κάθε όροφο. Σε κάθε δωμάτιο, έχουμε αισθητήρες οι οποίοι ελέγχουν και εκπέμπουν μετρήσεις για διαφορετικά χαρακτηριστικά του δωματίου, όπως για παράδειγμα την θερμοκρασία του και την ενεργειακή του κατανάλωση, μάλιστα στην γενική περίπτωση θα μπορούσε το κτίριο να είναι κάποιο εργοστάσιο οπότε να ελέγχεται με αισθητήρες και η ενεργειακή του παραγωγή. Γενικά δεν υπάρχει περιορισμός για τον τύπο των αισθητήρων και τις τιμές που παρακολουθούν, σκοπός της αρχιτεκτονικής μας είναι να προσφέρει δυνατότητα επεξεργασίας στις τιμές που στέλνονται από αυτούς. Θέλουμε δηλαδή να έχουμε σε πραγματικό χρόνο συνδυάζοντας τις μετρήσεις από όλους τους αισθητήρες το συνολικό αποτέλεσμα για το σύνολο του κτιρίου, του κάθε ορόφου.

9.3.1 Ανάλυση Data Source

Ακολουθώντας την αλυσίδα που παρουσιάσαμε ξεκινάμε με τους αισθητήρες. Κάθε αισθητήρας που χρησιμοποιούμε είναι συμβατός με πρωτόκολλο MQTT. Κάθε αισθητήρας κάνει publish τον τύπο του ο οποίος αποτελείται από έναν αριθμό, την τιμή του μεγέθους που παρακολουθεί και το χρονόσημό (timestamp) αυτής της μέτρησης σε ένα topic. Μπορούμε να επιλέξουμε είτε κάθε δωμάτιο να κάνει publish σε διαφορετικό topic, είτε να έχουμε διαφορετικό topic ανάλογα με τον όροφο ή ακόμα και να κάνουμε publish σε topics ανάλογα με τον τύπο του αισθητήρα (αισθητήρας θερμότητας, αισθητήρας ενέργειας, κτλ.).

Μετά τους αισθητήρες ακολουθούν οι MQTT brokers οι οποίοι είναι υπεύθυνοι να διαμοιράσουν τα μηνύματα στους sensor controllers. Για τους MQTT brokers μπορούμε να θέσουμε

το Quality of Service και το retention policy για τα μηνύματα. Κάθε sensor controller κάνει subscribe στο συγκεκριμένο topic στο οποίο κάνουν publish οι αισθητήρες που θέλει να παρακολουθήσει και δουλειά του είναι να λαμβάνει τα μηνύματα που έρχονται και να τα γράφει (produce) σε συγκεκριμένο topic στον Kafka Server με την μορφή key-value. Και πάλι η επιλογή του ποια δεδομένα θα γραφούν σε ποια topics γίνεται με βάση του τι θέλουμε να μετρήσουμε και πως θέλουμε να είναι χωρισμένα τα δεδομένα μας, και έχει μικρό αντίκτυπο στην επίδοση της αρχιτεκτονικής μας, αν γίνει σωστός διαμοιρασμός του όγκου των δεδομένων. Σε αυτό το σημείο, έχουμε τα δεδομένα ως μια ροή δεδομένων με μορφή key-value, όπου το key είναι ο τύπος του αισθητήρα σε string και το value αποτελείται από ένα json το οποίο έχει την μορφή value = { "value": τιμή μέτρησης, "timestamp_custom": χρονόσημο μέτρησης }.

Το Data Source layer υλοποιήθηκε σε γλώσσα προγραμματισμού Python.

9.3.2 Ανάλυση Speed layer

Τον Kafka Server ακολουθεί η βαθμίδα της εφαρμογής. Σε αυτό το σημείο δοκιμάστηκαν πολλές διαφορετικές εφαρμογές με βάση τα εργαλεία που αναλύθηκαν στα παραπάνω κεφάλαια. Σκοπός του Speed layer είναι η συλλογή και μετατροπή των αρχικών δεδομένων των αισθητήρων σε χρήσιμη πληροφορία. Για τον σκοπό αυτό συλλέγουμε τα δεδομένα ανά τακτά χρονικά διαστήματα (χρονικά παράθυρα) τα μετατρέπουμε μέσω συναρτήσεων και τα οδηγούμε στην έξοδο του συστήματος. Παρακάτω παρουσιάζεται το πώς υλοποιεί η κάθε εφαρμογή την συγκεκριμένη διαδικασία.

Apache Spark

Πρώτο χρησιμοποιήθηκε το εργαλείο Apache Spark, η λογική του όμως απαιτεί την επεξεργασία και την μετατροπή δεδομένων τα οποία προέρχονται από batches, συνεπώς μετά από τις αρχικές προσπάθειες και εφαρμογές που αναπτύχθηκαν και εκτελέστηκαν αποφασίστηκε να μην αναλυθεί περεταίρω, διότι αποφασίστηκε το κτιριακό μας ψηφιακό οικοσύστημα να ακολουθήσει προοπτικές Kappa αρχιτεκτονικης στην οποία το batch layer απουσιάζει. Ως εκ τούτου, το εν λόγω εργαλείο δεν θα αναλυθεί σε βάθος, παρόλα αυτά η λειτουργία του και η συμπεριφορά του όσον αφορά έναν τελικό χρήστη δεν διαφέρει σημαντικά από τα επόμενα δύο εργαλεία που θα παρουσιάσουμε.

Kafka Streams

Επόμενο στην σειρά εργαλείο είναι η βιβλιοθήκη Kafka Streams. Όπως αναφέραμε και στο ομώνυμο κεφάλαιο η βιβλιοθήκη αυτή επεξεργάζεται και διαχειρίζεται δεδομένα σε μορφή stream. Τα δεδομένα που θα οδηγηθούν ως είσοδος στην εφαρμογή μας βρίσκονται στον Kafka Server σε μορφή ζευγαριού key-value που αναφέρθηκε προηγουμένως. Χωρίς βλάβη της γενικότητας θα περιγράψουμε την διαδικασία για την μετατροπή και παρακολούθηση τριών αισθητήρων σε ένα δωμάτιο του κτιρίου μας με την βοήθεια της βιβλιοθήκης Kafka Streams.

Η εφαρμογή ξεκινάει κατασκευάζοντας έναν kafka consumer που καταναλώνει τα δεδομένα

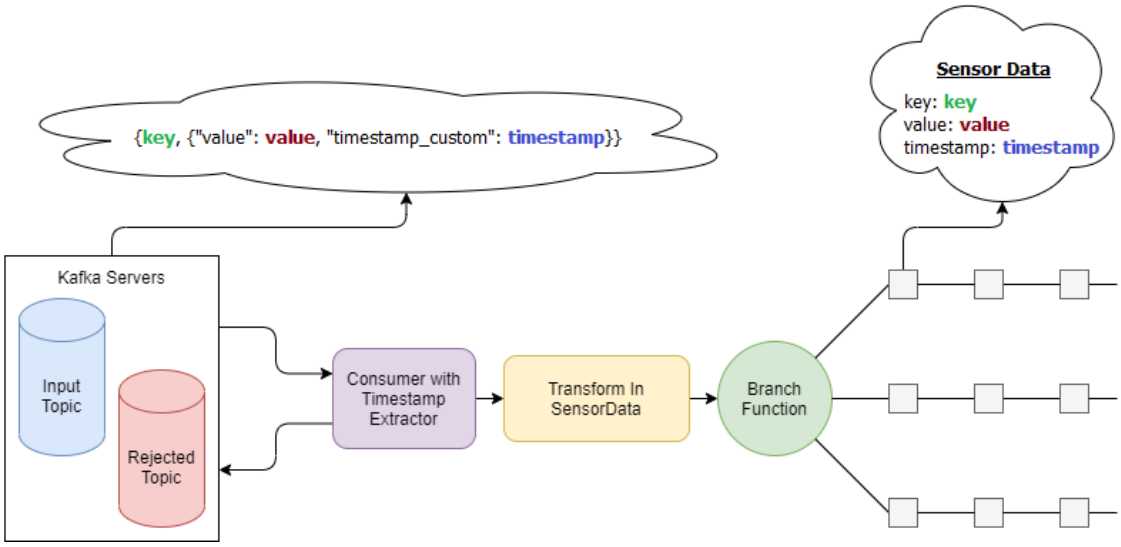
από το Kafka topic στο οποίο γράφει ο producer της sensor controllers βαθμίδας. Από εκεί διαβάζουμε τα δεδομένα τα οποία έρχονται σε μορφή key-value, όπου key ο τύπος του αισθητήρα και value η τιμή και το χρονόσημο του.

Σε μια streaming εφαρμογή το πιο σημαντικό κομμάτι είναι να μπορούμε να βασιστούμε και να εγγυηθούμε ότι τα δεδομένα μας έρχονται με την σωστή σειρά και ότι αποτελούν τιμές οι οποίες είναι χρονικά έγκυρες. Στην συγκεκριμένη περίπτωση οι τιμές που μας προωθούν οι αισθητήρες πρέπει να έχουν χρονικά αυξανόμενα χρονόσημα. Προσθέτουμε, λοιπόν, στο κομμάτι που καταναλώνουμε τα δεδομένα έναν προσαρμοσμένο timestamp extractor ο οποίος είναι υπεύθυνος να απορρίπτει τα δεδομένα που έρχονται εκτός σειράς.

Από μια πλευρά δεν θέλουμε να εισέρχονται καθυστερημένα δεδομένα στην εφαρμογή μας αλλά θα ήταν κακή πρακτική απλά να απορρίπτουμε αυτές τις πληροφορίες διότι μπορεί να είναι σοβαρές και σημαντικές για την παρακολούθηση των αισθητήρων και να διαθέτουν πληροφορίες οι οποίες είναι χρήσιμες για τον χρήστη. Γι αυτό το λόγο αποφασίστηκε ο συγκεκριμένος extractor να είναι υπεύθυνος για την καταγραφή αυτών των δεδομένων σε ένα ξεχωριστό topic στον Kafka Server με σκοπό ο τελικός χρήστης να μπορεί να ανατρέξει σε αυτά για σκοπούς είτε ενός ιστορικού είτε για να εκτελέσει διαφορετικές εφαρμογές με είσοδο αυτά τα δεδομένα. Συνηθισμένη πρακτική αποτελεί η ενσωμάτωση αυτών των δεδομένων στα αποτελέσματα της εφαρμογής παρακολούθησης του κτιρίου μέσα από άλλες εφαρμογές που διαχειρίζονται τα λάθη αυτά. Ένας τρόπος που ακολουθείτε από παρόμοιες εμπορικές εφαρμογές είναι ανα τακτά χρονικά διαστήματα ή σε χρόνους που το δίκτυο δεν παρουσιάζει μεγάλη κίνηση τα απορριφθέντα δεδομένα χρησιμοποιούνται για να ενσωματωθούν και να διορθώσουν τα live δεδομένα. Στην παρούσα αρχιτεκτονική απλά αποθηκεύονται σε Kafka topic με σκοπό να μην χαθούν και να αξιοποιηθούν από τον τελικό χρήστη όπως αυτός κρίνει σωστό. Τα απορριφθέντα δεδομένα αποθηκεύονται στο καινούργιο topic με την ίδια μορφή που είχαν στο topic από το οποίο αντλεί δεδομένα η εφαρμογή.

Στη συνέχεια, τα δεδομένα που έχουν σωστό χρονόσημο μετασχηματίζονται σε αντικείμενα στο σύστημα τα οποία ονομάζονται SensorData. Κάθε SensorData περιέχει το κλειδί το οποίο προέρχεται από τον kafka, την τιμή του και το χρονόσημό του. Η τιμή και το χρονόσημο προέρχεται από το πεδίο value στο kafka server το οποίο όπως αναφέραμε έχει την ακόλουθη μορφή value = { "value": τιμή μέτρησης, "timestamp_custom": χρονόσημο μέτρησης }.

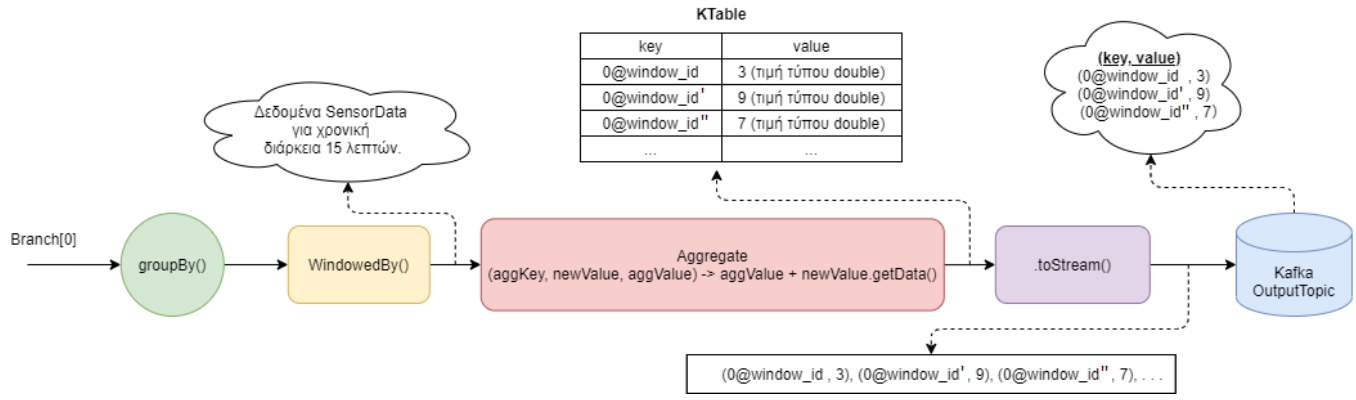
Τέλος τα δεδομένα χωρίζονται σε υπο-streams, με βάση τον τύπο του αισθητήρα από τον οποίο προήλθε η μέτρηση (δηλαδή με βάση το key), με την βοήθεια της συνάρτησης branch() που αναλύθηκε στο κεφάλαιο του Kafka Streams. Παρακάτω παρουσιάζεται ένα σχεδιάγραμμα (σχήμα 9.3) της διαδικασίας που παρουσιάσαμε.



Σχήμα 9.3: From key-value pair to SensorData Object

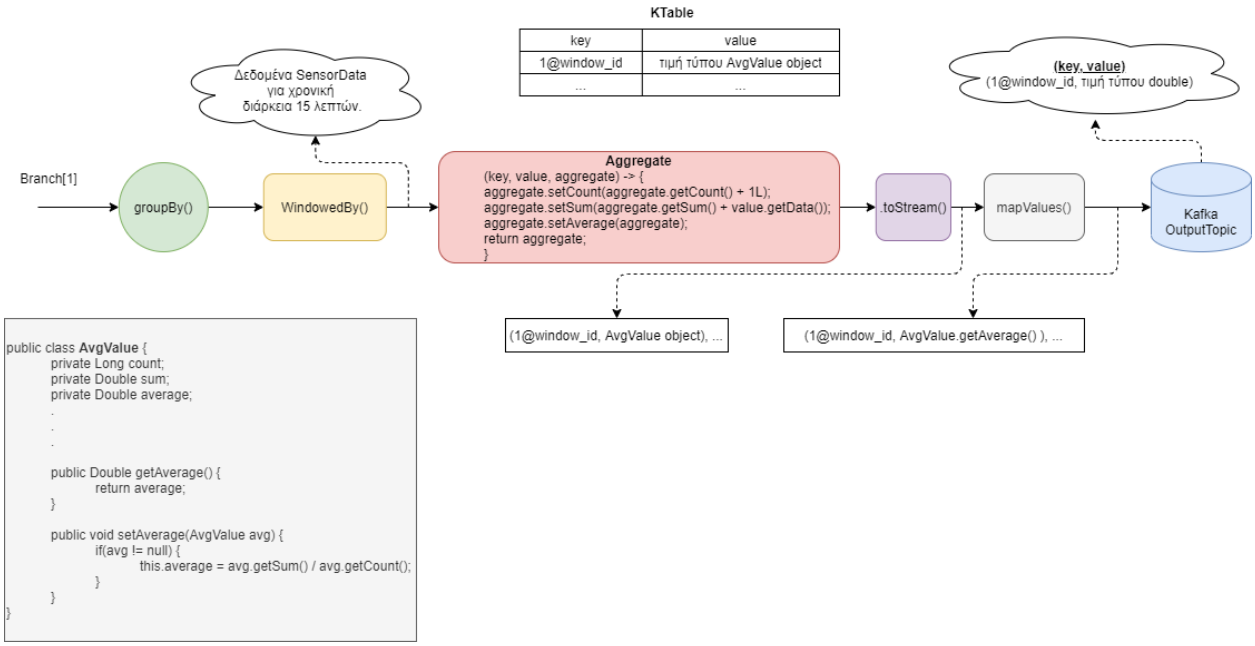
Σε αυτό το σημείο έχουμε τα τρία διαφορετικά branches, στο πρώτο έχουμε τους αισθητήρες τύπου 0, στο δεύτερο αυτούς που είναι τύπου 1, και το τέλος τους τύπου 2. Έπειτα παίρνουμε το κάθε υπο-stream, και για κάθε ένα από αυτά κάνουμε τους ανάλογους μετασχηματισμούς.

Πρώτα διαχειριζόμαστε το πρώτο υπο-stream, θέλουμε να εκτελούμε την συνάρτηση μετασχηματισμού μας για ένα συγκεκριμένο χρονικό παράθυρο. Μας ενδιαφέρει δηλαδή να παρακολουθούμε το άθροισμα τον αισθητήρων τύπου 0 ανά κάποιες συγκεκριμένες χρονικές στιγμές, για τον σκοπό αυτό δημιουργούμε ένα window τύπου tumbling με μήκος ίσο με την διάρκεια για την οποία θέλουμε να παρακολουθούμε. Έστω, στην συγκεκριμένη περίπτωση, ότι θέλουμε να παρακολουθούμε τα αθροίσματα του αισθητήρα τύπου 0 ανά 15 λεπτά, δημιουργούμε λοιπόν ένα tumbling παράθυρο με μήκος 15 λεπτά και αθροίζουμε μέσω του μετασχηματισμού aggregate() όλα τα δεδομένα που δεχόμαστε για το χρονικό διάστημα τον 15 λεπτών. Κάθε φορά που κλείνει ένα χρονικό παράθυρο επιστρέφεται μια εγγραφή σε ένα KTable η οποία έχει την μορφή (Windowed<String>, Double) το Windowed<String> είναι ο τύπος του πεδίου που αποτελείται από το κλειδί (τύπο αισθητήρα) και το χαρακτηριστικό id του παραθύρου ενώ το Double είναι ο τύπος του πεδίου που περιέχει το αποτέλεσμα της άθροισης. Σημαντική παρατήρηση είναι πως η άθροιση ξαναξεκινά από το μηδέν για κάθε νέο παράθυρο.



Σχήμα 9.4: Processing for branch 0

Συνέχεια έχει το δεύτερο υπο-stream όπου η διαδικασία είναι η ίδια με το πρώτο μέχρι το σημείο της συνάρτησης μετασχηματισμού. Στην συγκεκριμένη περίπτωση θέλουμε να επιστρέφουμε τον μέσο όρο των μετρήσεων των αισθητήρων τύπου 1, συνεπώς χρησιμοποιούμε ξανα την συνάρτηση aggregate() όμως την μετασχηματίζουμε ώστε να δέχεται αντικείμενα και όχι απλούς αριθμούς. Με αυτόν τον τρόπο αντί να κρατάμε απλά ένα επιμέρους άθροισμα, χρησιμοποιούμε, αθροίζουμε και συγχωνεύουμε αντικείμενα τύπου AvgValue. Το αντικείμενο AvgValue περιέχει δύο πεδία το μερικό άθροισμα μέχρι εκείνη την στιγμή (sum) και το σύνολο των αντικειμένων που έχει αθροίσει (count). Κάθε φορά, λοιπόν, που δεχόμαστε μέσα στον μετασχηματισμό aggregate() ένα SensorData, το μετατρέπουμε εσωτερικά σε ένα AvgValue, με τιμή συμ την τιμή της μέτρησης (value) του και count ένα, και έπειτα το συγχωνεύουμε με το ήδη υπάρχων AvgValue προσθέτοντας τα πεδία του μερικού αθροίσματος (sum) και του count μεταξύ τους. Και σε αυτήν την περίπτωση, κάθε φορά που κλείνει ένα χρονικό παράθυρο επιστρέφεται μια εγγραφή σε ένα KTable η οποία έχει την μορφή (Windowed<String>, AvgValue) το Windowed<String> είναι ο τύπος του πεδίου που αποτελείται από το κλειδί (τύπο αισθητήρα) και το χαρακτηριστικό id του παραθύρου ενώ το AvgValue είναι ο τύπος του πεδίου που περιέχει το αντικείμενο AvgValue. Σημαντική παρατήρηση είναι πως και εδώ, η άθροιση ξαναξεκινά από το μηδέν με count 0 για κάθε νέο παράθυρο.

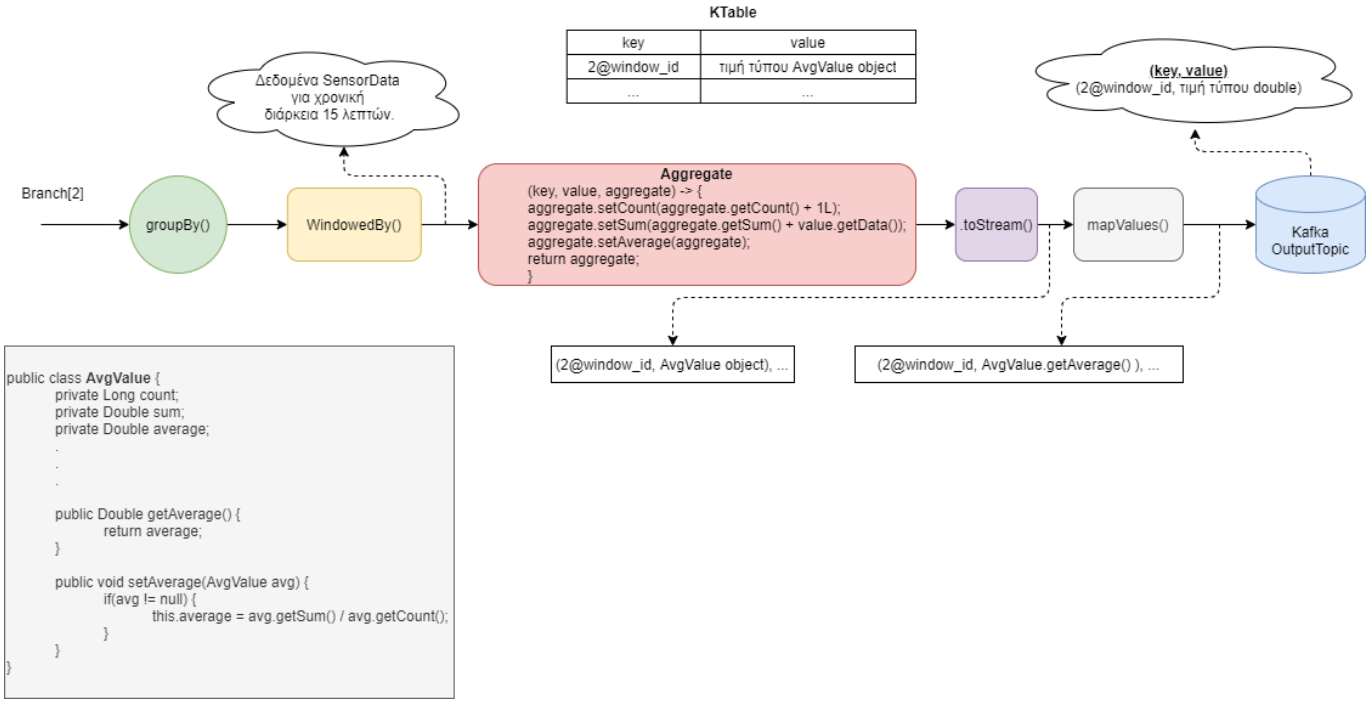


Σχήμα 9.5: Processing for branch 1

Τέλος επεξεργαζόμαστε το τρίτο και τελευταίο υπο-stream με τον ίδιο τρόπο που επεξεργαζόμαστε το δεύτερο λαμβάνοντας τον μέσο όρο των μετρήσεων ανά 15 λεπτά (σχήμα 9.6).

Τελευταίο βήμα της streaming εφαρμογής μας είναι η αποθήκευση των KTables που προέκυψαν από τις μετατροπές σε ένα Kafka Server σε συγκεκριμένο topic. Σε αυτό το σημείο μπορούμε να επιλέξουμε που θέλουμε να αποθηκεύσουμε τα αποτελέσματα των τριών επιμέρους KTables. Μπορεί να επιλέξουμε να αποθηκεύσουμε κάθε διαφορετικό KTable στο δικό του topic, μπορούμε να τους αποθηκεύσουμε όλους σε κοινό topic ή με όποιον άλλο τρόπο βοηθά την λογική που ακολουθούμε στην εφαρμογή. Σε κάθε περίπτωση πρέπει να έχουμε το μυαλό μας να διαμοιράζουμε σωστά τον όγκο των δεδομένων, να διαχειριζόμαστε αποδοτικά την μνήμη μας και να έχουμε στο μυαλό μας την γενική εικόνα όλης της λειτουργικότητας που προσπαθούμε να οικοδομήσουμε. Στο συγκεκριμένο παράδειγμα, εφόσον μιλάμε για δεδομένα από μικρό αριθμό αισθητήρων, για ένα δωμάτιο ανά 15 λεπτά καλή πρακτική είναι να αποθηκεύσουμε τους KTables μας σε κοινό topic. Δημιουργούμε λοιπόν έναν producer για κάθε KTable ο οποίος αναλαμβάνει την αποθήκευση των αποτελεσμάτων σε Kafka Server στην μορφή ζευγαριού key-value.

Στη πρώτη και την τρίτη περίπτωση έχουμε KTable με μορφή εγγραφών (Windowed<String>, Double) οπότε η αντιστοιχία είναι key → Windowed<String> και value → Double. Στην δεύτερη έχουμε KTable με μορφή εγγραφών (Windowed<String>, AvgValue) οπότε η αντιστοιχία είναι key → Windowed<String> και value → AvgValue.getAverage(), όπου getAverage() αποτελεί συνάρτηση του αντικειμένου AvgValue, η οποία το μόνο που κάνει είναι να συνδυάζει τα δύο πεδία του (sum και count) και να γυρνά το μέσο όρο ως sum/count.



Σχήμα 9.6: Processing for branch 2

Apache Flink

Επόμενο στην σειρά εργαλείο είναι το Apache Flink. Όπως αναφέραμε και στο σχετικό κεφάλαιο το εργαλείο αυτό επεξεργάζεται και διαχειρίζεται δεδομένα σε μορφή stream. Τα δεδομένα που θα οδηγηθούν ως είσοδος στην εφαρμογή μας βρίσκονται στον Kafka Server σε μορφή ζευγαριού key-value που αναφέρθηκε προηγουμένως. Χωρίς βλάβη της γενικότητας θα περιγράψουμε την διαδικασία για την μετατροπή και παρακολούθηση τριών αισθητήρων σε ένα δωμάτιο του κτιρίου μας με την βοήθεια του Apache Flink.

Η εφαρμογή ξεκινάει κατασκευάζοντας έναν kafka consumer που καταναλώνει τα δεδομένα από το Kafka topic στο οποίο γράφει ο producer της sensor controllers βαθμίδας. Από εκεί διαβάζουμε τα δεδομένα τα οποία έρχονται σε μορφή key-value, όπου key ο τύπος του αισθητήρα και value η τιμή και το χρονόσημο του. Σε μια streaming εφαρμογή το πιο σημαντικό κομμάτι είναι να μπορούμε να βασιστούμε και να εγγυηθούμε ότι τα δεδομένα μας έρχονται με την σωστή σειρά και ότι αποτελούν τιμές οι οποίες είναι χρονικά έγκυρες. Στην συγκεκριμένη περίπτωση οι τιμές που μας προωθούν οι αισθητήρες πρέπει να έχουν χρονικά αυξανόμενα χρονόσημα. Προσθέτουμε, λοιπόν, στο κομμάτι που καταναλώνουμε τα δεδομένα έναν προσαρμοσμένο Watermark Strategy το οποίο όπως αναφέρθηκε και στο κεφάλαιο για το Apache Flink είναι υπεύθυνο να εξάγει από τα δεδομένα το timestamp και στο που παράγει κάθε δεδομένο να συνοδεύεται από αυτό το timestamp. Τελευταίο στην αλυσίδα που αφορά την είσοδο δεδομένων στην εφαρμογή μας είναι όπως και στη Kafka Streams εφαρμογή η μετατροπή του ζευγαριού key-value του Kafka Server σε αντικείμενο SensorData, το οποίο γίνεται με ακριβώς με τον ίδιο τρόπο που παρουσιάστηκε στην εφαρμογή του Kafka Streams.

Σε αυτό το στάδιο έχουμε μετασχηματίσει τα δεδομένα μας στην ειδική κλάση DataStream του Flink, η οποία περιέχει αντικείμενα SensorData και είμαστε έτοιμοι να ξεκινήσουμε τους αντίστοιχους υπολογισμούς και τους μετασχηματισμούς.

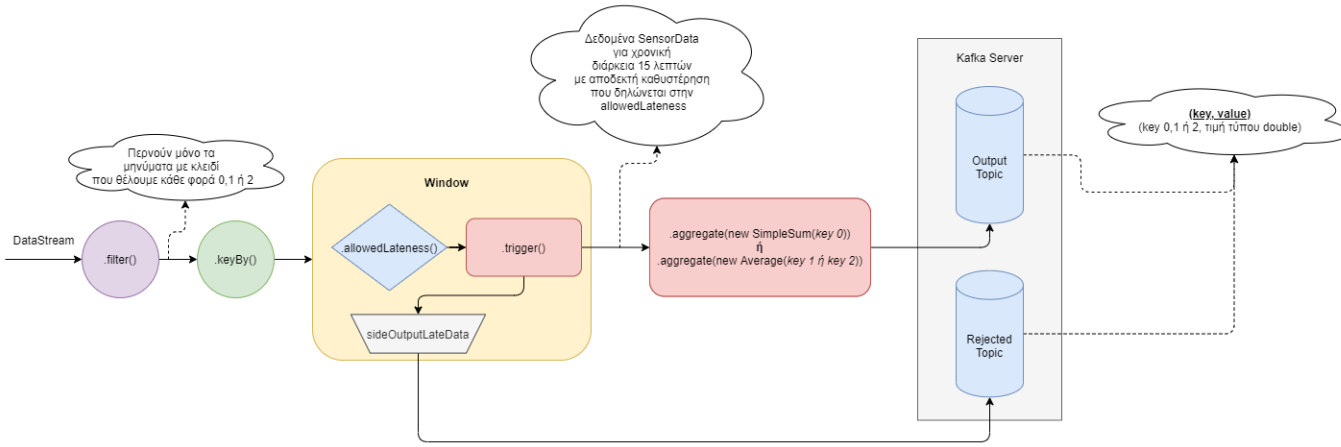
Πρώτα θέλουμε να πραγματοποιήσουμε το απλό άθροισμα για τους αισθητήρες τύπου 0. Δυστυχώς στο Apache Flink δεν έχουμε συνάρτηση που χωρίζει το stream σε υπο-streams όπως η branch() διότι η αντίστοιχη συνάρτηση split είναι deprecated και έχει πάψει να χρησιμοποιείται [9]. Θα χρησιμοποιήσουμε λοιπον, ως είσοδο όλο το DataStream και με την βοήθεια της μεθόδου filter() θα απομονώσουμε μόνο τα αντικείμενα SensorData τα οποία έχουν κλειδί το τύπο αισθητήρα 0. Στην συνέχεια θα δημιουργήσουμε, όπως και στην περίπτωση της εφαρμογής του Kafka Streams, tumbling παράθυρα τα οποία θα παρακολουθούν τα δεδομένα και θα εκτελούν τους μετασχηματισμούς κάθε 15 λεπτά. Στην συγκεκριμένη εφαρμογή προσθέσαμε και μία μέθοδο που μας επιτρέπει να δεχόμαστε δεδομένα τα οποία έρχονται με χρονική καθυστέρηση μικρότερη από μια συγκεκριμένη τιμή. Με αυτόν τον τρόπο η μέθοδος allowedLateness() επιτρέπει στα δεδομένα εισόδου να συμπεριληφθούν στο παράθυρο αν η χρονική καθυστέρηση προσέλευσής τους είναι μικρότερη από μια χρονική τιμή. Σε αντίθεση με το Kafka Streams η διαλογή των χρονικά άκυρων πακέτων γίνεται ακριβώς μετά από αυτήν την μέθοδο allowedLateness(). Στην περίπτωση του Flink δεν χρειάζεται να επιβαρύνουμε την συνάρτηση που εκπέμπει τα timestamp με την συλλογή και απόρριψη των χρονικά άκυρων δεδομένων, το Flink διαχειρίζεται εσωτερικά τα timestamps και επιτρέπει να περάσουν στο παράθυρο υπολογισμού μόνο χρονικά έγκυρα γεγονότα και γεγονότα

με μικρότερη καθυστέρηση από το όριο που έθεσε η allowedLateness(). Τα υπόλοιπα απορριφθέντα δεδομένα συλλέγονται σε ένα ξεχωριστό DataStream το οποίο ορίζουμε με την μέθοδο sideOutputLateData().

Μετά και από αυτόν τον έλεγχο τα αντικείμενα SensorData τα οποία περνούν στο παράθυρο αθροίζονται με την μέθοδο aggregate(). Το Apache Flink χρησιμοποιεί την κλάση AggregateFunction ως είσοδο στην μέθοδο aggregate(). Αυτή η κλάση αποτελεί ουσιαστικά μια generic κλάση που χρησιμοποιεί accumulators που προστίθενται και συγχωνεύονται για αθροιστικούς σκοπούς, οπότε η δουλειά μας είναι να προσαρμόσουμε την κλάση AggregateFunction στα δικά μας δεδομένα και να κάνουμε overload τις τέσσερις βασικές μεθόδους της (createAccumulator, add, get Result, merge) να δέχονται στην είσοδο αντικείμενα τύπου SernsorData και να δημιουργούν accumulators για να πραγματοποιήσουμε μετά την πρόσθεση, την συγχώνευση και την προβολή του αποτελέσματος αυτών. Στην έξοδο της AggregateFunction έχουμε και τα αποτελέσματα της μετατροπής τα οποία είναι πλειάδες (tuples) της μορφής Tuple<String, Double> όπου τύπου String είναι το key του SensorData και τύπου Double είναι το αποτέλεσμα της πρόσθεσης και συγχώνευσης στους accumulators όλων των τιμών (value) των SensorData που ανήκουν στο παράθυρο.

Στη συνέχεια έχουμε τους αισθητήρες τύπου 1. Για αυτούς τους αισθητήρες, ακολουθούμε την ίδια διαδικασία που ακολουθήσαμε και στους τύπου 0 με τη διαφορά ότι κάνουμε filter() με διαφορετικό key στη μέθοδο aggregate() χρησιμοποιούμε διαφορετική AggregateFunction. Αυτή την φορά όπως και στην εφαρμογή Kafka Streams θέλουμε να υπολογίσουμε τον μέσο όρο των τιμών των αισθητήρων οπότε φροντίζουμε ο accumulator που κατασκευάζεται κατά την άθροιση εκτός από τα παιδία value του SensorData να προσθέτει και ένα στο πεδίο count για να κρατάμε και την πληροφορία για τον αριθμό των τιμών που έχουμε αθροίσει. Στην έξοδο της AggregateFunction έχουμε και πάλι τα αποτελέσματα της μετατροπής τα οποία είναι πλειάδες (tuples) της μορφής Tuple<String, Double> όπου τύπου String είναι το key του SensorData και τύπου Double είναι το αποτέλεσμα της πρόσθεσης και συγχώνευσης στους accumulators όλων των τιμών (value) των SensorData διαιρεμένο με τον αριθμό των SensorData που προστέθηκαν στο συγκεκριμένο παράθυρο (count).

Τέλος έχουμε τους αισθητήρες τύπου 2, όπου η διαδικασία υπολογισμού είναι ίδια με την διαδικασία που παρουσιάστηκε για τους αισθητήρες τύπου 1. Σημαντική παρατήρηση είναι πως και στην συγκεκριμένη εφαρμογή η παρακολούθηση του αθροίσματος και του μέσου όρου ξαναξεκινά από το μηδέν για κάθε νέο παράθυρο.



Σχήμα 9.7: Processing in Flink

```java
public class SimpleSum implements AggregateFunction<SensorData, CustomAccumulator, Tuple2<String, Double>> {

    private String key;

    public SimpleSum(String key) {
        this.key = key;
    }

    @Override
    public CustomAccumulator createAccumulator() {
        return new CustomAccumulator(this.key, 0L, (double) 0);
    }

    @Override
    public CustomAccumulator add(SensorData value, CustomAccumulator accumulator) {
        String key = value.getKey();
        Double sum = accumulator.getSum() + value.getData();

        accumulator.setKey(key);
        accumulator.setSum(sum);
        return accumulator;
    }

    @Override
    public Tuple2<String, Double> getResult(CustomAccumulator accumulator) {
        return new Tuple2<String, Double>(accumulator.getKey(), accumulator.getSum());
    }

    @Override
    public CustomAccumulator merge(CustomAccumulator a, CustomAccumulator b) {
        a.setSum(a.getSum() + b.getSum());
        return a;
    }
}
```

```java
public class Average implements AggregateFunction<SensorData, CustomAccumulator, Tuple2<String, Double>> {

    private String key;

    public Average(String key) {
        this.key = key;
    }

    @Override
    public CustomAccumulator createAccumulator() {
        return new CustomAccumulator(this.key, 0L, (double) 0);
    }

    @Override
    public CustomAccumulator add(SensorData value, CustomAccumulator accumulator) {
        String key = value.getKey();
        Double sum = accumulator.getSum() + value.getData();
        Long count = accumulator.getCount() + 1L;

        accumulator.setKey(key);
        accumulator.setSum(sum);
        accumulator.setCount(count);

        return accumulator;
    }


    @Override
    public Tuple2<String, Double> getResult(CustomAccumulator accumulator) {
        return new Tuple2<>(accumulator.getKey(), accumulator.getSum() / (double) accumulator.getCount());
    }

    @Override
    public CustomAccumulator merge(CustomAccumulator a, CustomAccumulator b) {
        a.setSum(a.getSum() + b.getSum());
        a.setCount(a.getCount() + b.getCount());
        return a;
    }
}
```

Σχήμα 9.8: SimpleSum and Average classes

Τελευταίο βήμα της streaming εφαρμογής μας είναι η αποθήκευση των πλειάδων που προέκυψαν από τις μετατροπές σε ένα Kafka Server σε συγκεκριμένο topic. Σε αυτό το σημείο μπορούμε να επιλέξουμε που θέλουμε να αποθηκεύσουμε τα αποτελέσματα των επιμέρους πλειάδων. Για κάθε σενάριο που θα επιλέξουμε, όπως και για την εφαρμογή Kafka Streams πρέπει να έχουμε το μυαλό μας να διαμοιράζουμε σωστά τον όγκο των δεδομένων, να διαχειριζόμαστε αποδοτικά την μνήμη μας και να έχουμε στο μυαλό μας την γενική εικόνα όλης της λειτουργικότητας που προσπαθούμε να οικοδομήσουμε. Στο συγκεκριμένο παράδειγμα, εφόσον μιλάμε για δεδομένα από μικρό αριθμό αισθητήρων, για ένα δωμάτιο ανά 15 λεπτά καλή πρακτική είναι να αποθηκεύσουμε τις πλειάδες μας σε κοινό topic. Δημιουργούμε λοιπόν έναν producer ο οποίος αναλαμβάνει με την μέθοδο addSink() την αποθήκευση των αποτελεσμάτων σε Kafka Server στην μορφή ζευγαριού key-value. Σε όλες τις περιπτώσεις έχουμε πλειάδες με μορφή εγγραφών (String, Double) οπότε η αντιστοιχία είναι key → String και value → Double.

Όπως εξηγήσαμε και παραπάνω έχουμε και τρία DataStream τα οποία προέρχονται από την μέθοδο sideOutputLateData() και περιέχουν τα δεδομένα που ήρθαν εκτός σειράς. Ακολουθώντας κοινή αντιμετώπιση στις δύο εφαρμογές Kafka Streams και Apache Flink αποφασίστηκε τα εν λόγω DataStreams να αποθηκευτούν σε ένα topic όπου θα συγκεντρώνουμε όλα τα απορριφθέντα δεδομένα με την ίδια μορφή που τα λάβαμε στην είσοδο έτσι ώστε ο χρήστης να μπορεί να τα διαχειριστεί όπως θέλει.

Το Speed layer υλοποιήθηκε σε γλώσσα προγραμματισμού Java.

9.3.3 Ανάλυση Serving layer

Αποτελεί την τελευταία βαθμίδα της αρχιτεκτονικής και αποτελείται από δύο βασικά κομμάτια. Το πρώτο κομμάτι είναι η Web εφαρμογή η οποία διαβάζει τα δεδομένα από τον τελευταίο Kafka Server και τα παρουσιάζει σε πραγματικό χρόνο, είναι υπεύθυνη δηλαδή για την οπτικοποίηση των δεδομένων και την προβολή τους στον χρήστη της εφαρμογής. Για το συγκεκριμένο κομμάτι χρησιμοποιούμε έναν Kafka consumer ο οποίος καταναλώνει δεδομένα από το τελικό topic των αποτελεσμάτων των παραθύρων και τα παρουσιάζει σε μια Web φόρμα. Για το συγκεκριμένο κομμάτι χρησιμοποιήθηκε το εργαλείο flask-socketio [2] το οποίο αποτελεί βιβλιοθήκη της γλώσσας python και παρέχει μικρής καθυστέρησης αλληλεπίδραση σε εφαρμογές που ακολουθούν το μοντέλο client-server.

Το δεύτερο κομμάτι της βαθμίδας υλοποιεί μια redis inmemory database στην οποία αποθηκεύονται τα τελικά δεδομένα των παραθύρων με σκοπό είτε την σύνταξη ενός αρχείου μετρήσεων, είτε τη δυνατότητα να γίνονται queries πάνω στα δεδομένα τα οποία αποθηκεύονται.

Το Serving layer υλοποιήθηκε σε γλώσσα προγραμματισμού Python.

9.3.4 Κλιμακωσιμότητα Ανάλυσης

Σημειώνεται ότι η λογική του Data Source, του Speed layer και του Serving layer παραμένει ίδια ακόμα και αν αντί για ένα δωμάτιο και μόνο τρεις τύπους αισθητήρων έχουμε πολλά περισσότερα δωμάτια και πολλούς περισσότερους αισθητήρες. Η αρχιτεκτονική έχει κατασκευαστεί ώστε να δέχεται στην είσοδο τις τιμές των δωματίων και των αισθητήρων σε μορφή παραμέτρων και είναι ικανή να ανταποκριθεί σε πολύ μεγαλύτερο όγκο δεδομένων. Παραπάνω έγινε η απλοποίηση, με σκοπό να αναλυθεί εκτενώς η πορεία της πληροφορίας από την αρχή μέχρι το τέλος μέσα στο σύστημα και να μην ληφθεί το κανένα επίπεδο ως ¨μαύρο κουτί' (black box). Είναι προφανές ότι το δωμάτιο και ο αισθητήρας αποτελούν τα μικρότερα δυνατά στοιχεία διαίρεσης μέσα στο σύνολο κτιριακού μας ψηφιακού οικοσυστήματος αφού ένα cluster αισθητήρων αποτελείται από πολλούς μεμονωμένους αισθητήρες και ένα κτίριο αποτελείται από πολλούς ορόφους οι οποίοι με την σειρά τους αποτελούνται από πολλά δωμάτια.

9.4 Αξιολόγηση-Πειράματα

Αφού κατασκευάσαμε τον κορμό της αρχιτεκτονική μας εκτελέσαμε εκτενείς χρονικά προσομοιώσεις, για να καταφέρουμε να την βελτιώσουμε και να εντοπίσουμε εξαρτήσεις ανάμεσα στα επίπεδα και τα εργαλεία καθώς και να προτείνουμε μελλοντικές προσθήκες και διαφοροποιήσεις στον τρόπο λειτουργίας. Εκτελέσαμε δύο είδους πειράματα, ένα για τον υπολογισμό των χαμένων μηνυμάτων στο σύστημα και ένα για τον χρόνο που περνά από την στιγμή που παράγεται το μήνυμα μέχρι την στιγμή που είναι διαθέσιμο στο speed layer. Τα δύο πειράματα εκτελέστηκαν ανάμεσα στο Data Source και το Speed layer, διότι σε αυτό το σημείο έχουμε διαφορετικές τεχνολογίες που συνδέονται μεταξύ τους (MQTT - Apache Kafka) και παρουσιάζει ενδιαφέρον η παρακολούθηση της συμπεριφοράς του συστήματος μας. Το Speed layer και το Serving layer αποτελούν μεταφορά δεδομένων μεταξύ κοινών εργαλείων και όπως διαπιστώθηκε η πειραματική μελέτη δεν παρουσιάζει κάποια ιδιαίτερη σημασία.

Τα παρακάτω διαγράμματα αποτελούν μετρήσεις αποτελεσμάτων που εκτελέστηκαν με έναν MQTT broker, έναν Kafka Server με ένα partition σε τοπικό υπολογιστή με τα παρακάτω χαρακτηριστικά:

Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz
Installed RAM	8,00 GB (7,87 GB usable)
System type	64-bit operating system, x64-based processor

Σχήμα 9.9: PC Specifications

Τα δεδομένα που χρησιμοποιήθηκαν είναι συνθετικά και έχουν την μορφή { "value": τιμή μέτρησης τύπου long, "timestamp_custom": χρονόσημο μέτρησης τύπου long}. Στα πειράματα εξετάζεται η αντοχή του συστήματος στον όγκο δεδομένων και οι χρονικές καθυστερήσεις, συνεπώς η χρήση πραγματικών μετρήσεων από αισθητήρες κτιρίων δεν είναι απαραίτητη.

Αρχικά μελετήσαμε την συμπεριφορά του συστήματός μας για διαφορετικές τιμές του MQTT quality of service (QoS) σε συνδυασμό με τη σημαία retain στα μηνύματα. Ο όγκος

των δεδομένων που χρησιμοποιούμε είναι 2000 μηνύματα το δευτερόλεπτο, δηλαδή 2000 μετρήσεις αισθητήρων ανά δευτερόλεπτο. Το διάγραμμα που λάβαμε παρουσιάζεται παρακάτω:

Σχήμα 9.10: Different QoS & Retention Policies

Από το διάγραμμα παρατηρούμε ότι το quality of service παίζει σημαντικό ρόλο στην συμπεριφορά του συστήματος και στο ποσοστό των μηνυμάτων που χάνονται ενώ το retention policy για τα μηνύματα δεν προκαλεί μεγάλες αλλαγές στην συμπεριφορά. Συνεπώς, αποφασίσαμε να εκτελέσουμε πιο ειδικές προσομοιώσεις για τα διάφορα quality of service κρατώντας το retention policy true.

9.4.1 Προσομοίωση χαμένων πακέτων

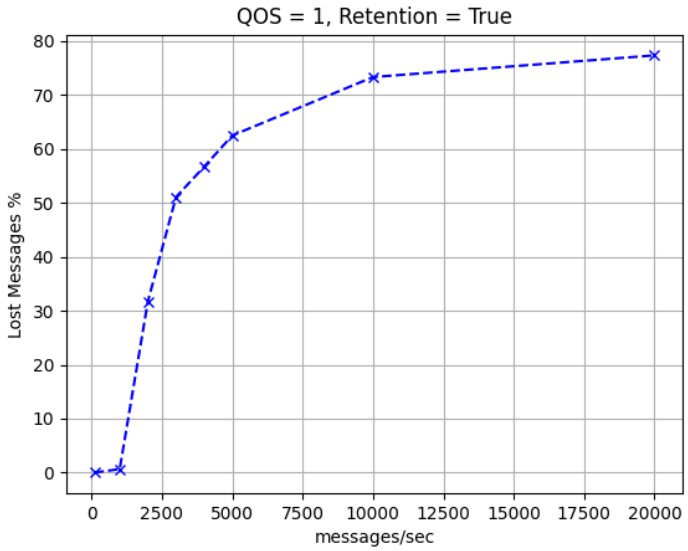
Πρώτη προσομοίωση ήταν να κρατάμε σταθερό το QoS και να αυξάνουμε τον αριθμό των μηνυμάτων ανά δευτερόλεπτο παρακολουθώντας το ποσοστό των χαμένων πακέτων. Το διάγραμμα που λάβαμε για QoS 0 είναι το παρακάτω:

Σχήμα 9.11: Losses for QoS 0

Για την μετάδοση δεδομένων με το QoS 0 ο αποστολέας (αισθητήρας) στέλνει μοναδικό μήνυμα στον MQTT broker και δεν περιμένει απάντηση συνεπώς η ανταλλαγή και η προώθηση των μηνυμάτων γίνεται με μεγάλη ταχύτητα και στο σύστημα δεν παρατηρείται συσσώρευση δεδομένων που τελικά οδηγεί στην απόρριψη μηνυμάτων. Όπως είναι φυσικό υπάρχει ένα ανώτατο όριο για το οποίο το σύστημα δεν χάνει μηνύματα, στο συγκεκριμένο παράδειγμα μετά από τα 4000 μηνύματα το δευτερόλεπτο αρχίζουμε να έχουμε μια ανοδική πορεία στο ποσοστό των χαμένων μηνυμάτων. Αξίζει να παρατηρηθεί πως το ποσοστό απόρριψης ενώ μετά τα 4000 αρχίζει μια έντονη ανοδική πορεία έπειτα από ένα σημείο σταθεροποιείται. Πράγματι, όπως είναι και λογικό, όσο και να γεμίσουμε το σύστημά μας, αν αυτό συνεχίσει να δουλεύει και δεν κλείσει λόγω αποτυχίας κάποιου server, ποτέ αυτό δεν πρόκειται να απορίψει όλα τα εισερχόμενα μηνύματα.

Επαναλάβαμε την ίδια προσομοίωση μόνο που αυτή την φορά επιλέξαμε QoS 1. Το διάγραμμα που προέκυψε παρουσίασε την ίδια συμπεριφορά.

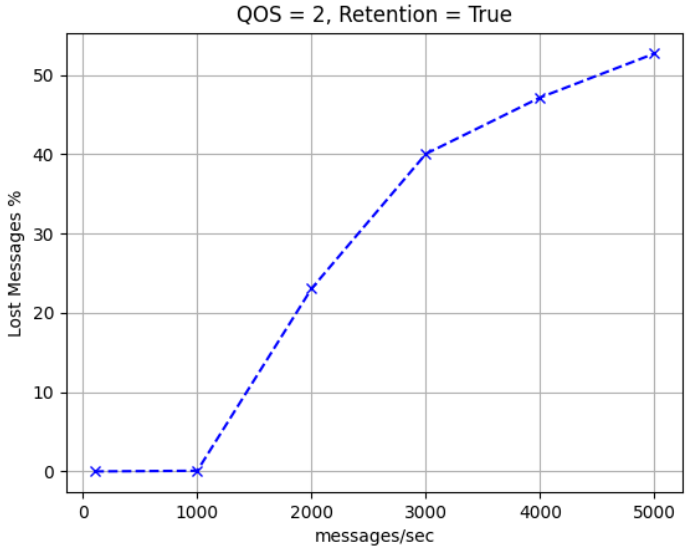
Για την μετάδοση δεδομένων με το QoS 1 ο αποστολέας (αισθητήρας) στέλνει μήνυμα στον MQTT broker και περιμένει απάντηση επιβεβαίωσης, συνεπώς η ανταλλαγή και η προώθηση των μηνυμάτων γίνεται με μικρότερη ταχύτητα συγκριτικά με το QoS 0. Ως αποτέλεσμα στο σύστημα παρατηρείται μεγαλύτερη συσσώρευση δεδομένων που τελικά οδηγεί στην απόρριψη μηνυμάτων. Στο σύνολό της η καμπύλη ακολουθεί την ίδια συμπεριφορά, παρουσιάζοντας μία προς τα αριστερά μετατόπιση. Όπως εξηγήσαμε, στο QoS 1, η προώθηση των μηνυμάτων



Σχήμα 9.12: Losses for QoS 1

πραγματοποιείται πιο αργά συνεπώς η ουρά εξυπηρέτησης αδειάζει πιο αργά και παρουσιάζει μικρότερη αντοχή σε ταυτόχρονο μεγάλο όγκο δεδομένων. Παρόλα αυτά και στο συγκεκριμένο διάγραμμα μετά την έντονη αύξηση του ποσοστού απόρριψης αυτό τείνει να σταθεροποιηθεί σε μια τιμή.

Τέλος επαναλάβαμε την ίδια προσομοίωση για QoS 2. Το διάγραμμα που προέκυψε παρουσίασε την ίδια συμπεριφορά με τα δύο προηγούμενα.



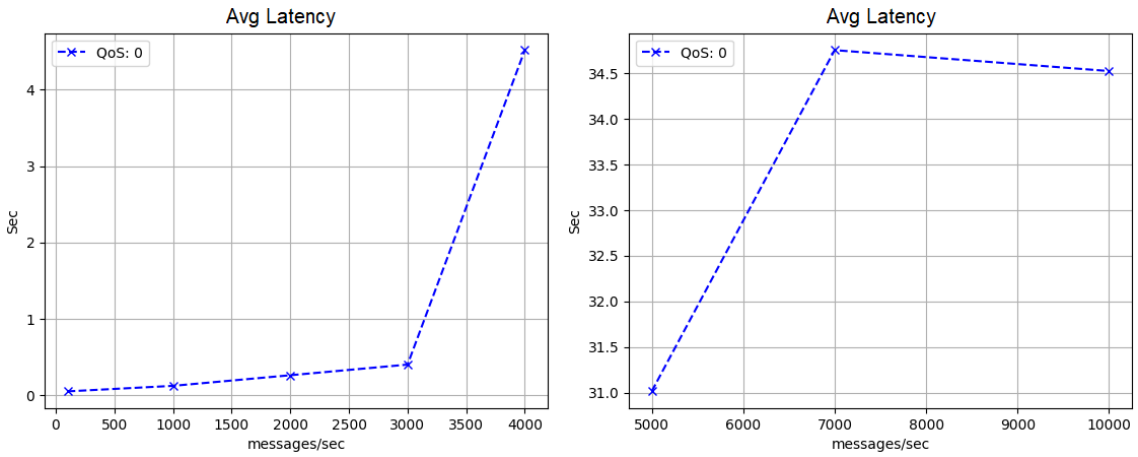
Σχήμα 9.13: Losses for QoS 2

Για την μετάδοση δεδομένων με το QoS 2 ο αποστολέας (αισθητήρας) στέλνει μήνυμα στον MQTT broker και περιμένει απάντηση επιβεβαίωσης την οποία μόλις λάβει ενημερώνει

τον broker, συνεπώς η ανταλλαγή και η προώθηση των μηνυμάτων γίνεται με μικρότερη ταχύτητα συγκριτικά με το QoS 0. Ως αποτέλεσμα, στο σύστημα παρατηρείται μεγαλύτερη συσσώρευση δεδομένων που τελικά οδηγεί στην απόρριψη μηνυμάτων. Στο σύνολό της η καμπύλη ακολουθεί την ίδια συμπεριφορά με τις δύο προηγούμενες.

9.4.2 Προσομοίωση χρόνου αποθήκευσης των μηνυμάτων στον Kafka Server

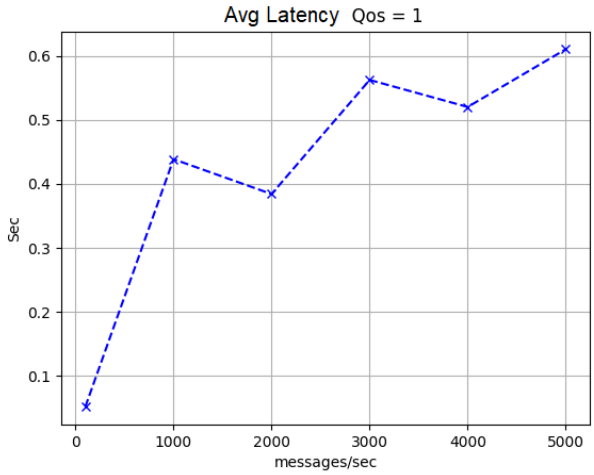
Δεύτερη προσομοίωση ήταν να κρατάμε σταθερό το QoS και να αυξάνουμε τον αριθμό των μηνυμάτων ανά δευτερόλεπτο παρακολουθώντας τον μέσο χρόνο που αυτά κάνουν για να αποθηκευτούν στον Kafka Server. Πιο συγκεκριμένα μετράμε την χρονική διαφορά ανάμεσα στο timestamp της μέτρησης που στέλνει ο αισθητήρας και στον χρόνο που το μήνυμα αποθηκεύεται στον Kafka Server στο τέλος Data Source επιπέδου. Το διάγραμμα που λάβαμε για QoS 0 είναι το παρακάτω:



Σχήμα 9.14: Average Latency for QoS 0

Όπως παρουσιάσαμε και στην προσομοίωση για τα χαμένα πακέτα το QoS 0 είναι το γρηγορότερο quality of service, παρόλα αυτά όσο ο όγκος των μηνυμάτων μεγαλώνει τόσο μεγαλώνει και το μήκος της ουράς εξυπηρέτησης. Συνεπώς, ο μέσος χρόνος που ένα μήνυμα παραμένει στην ουρά και περιμένει να αποθηκευτεί (produced) στο topic του Kafka Server μεγαλώνει όσο μεγαλώνουμε τα μηνύματα ανά δευτερόλεπτο. Και στην συγκεκριμένη περίπτωση βλέπουμε πως για μια τιμή η αύξηση αρχίζει να γίνεται ραγδαία και συνεχίζεται για μεγάλο διάστημα μέχρι που σταθεροποιείται όταν το σύστημα αρχίζει να χάνει μεγάλο αριθμό πακέτων. Συνεπώς παρουσιάζει άνω όριο που σχετίζεται με το άνω όριο της ουράς εξυπηρέτησης.

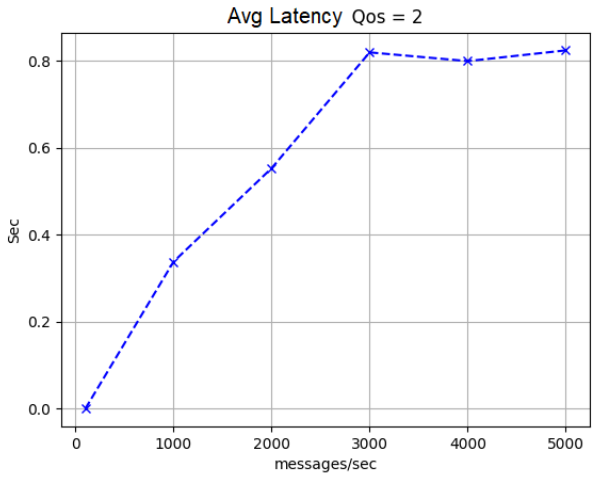
Επαναλάβαμε την ίδια προσομοίωση μόνο που αυτή την φορά επιλέγουμε QoS 1. Το διάγραμμα που προέκυψε παρουσίασε την ίδια συμπεριφορά και είναι το παρακάτω:



Σχήμα 9.15: Average Latency for QoS 1

Όπως και στην περίπτωση των χαμένων πακέτων λόγω του QoS η ουρά εξυπηρέτησης προχωράει με πιο αργούς ρυθμούς συνεπώς έχουμε μια αυξανόμενη καθυστέρηση όσο μεγαλώνει ο όγκος των μηνυμάτων. Ο χρόνος που ένα μήνυμα παραμένει στην ουρά και περιμένει να αποθηκευτεί (produced) στο topic του Kafka Server μεγαλώνει όσο μεγαλώνουμε τα μηνύματα ανα δευτερόλεπτο όπως ακριβώς και στην περίπτωση του QoS 0. Όπως και στο προηγούμενο διάγραμμα έτσι και σε αυτό παρουσιάζεται μια τάση προς σταθεροποίηση όταν το σύστημα αρχίζει να χάνει μεγάλο αριθμό πακέτων.

Τέλος επαναλάβαμε την ίδια προσομοίωση για QoS 2. Το διάγραμμα που προέκυψε παρουσίασε την ίδια συμπεριφορά με τα δύο προηγούμενα.



Σχήμα 9.16: Average Latency for QoS 2

Μια βασική και αξιοσημείωτη παρατήρηση προκύπτει αναλύοντας συνδιαστηκά τα αποτελέσματα των χρόνων των τριών QoS στην περίπτωση που έχουμε 5000 μηνύματα το λεπτό. Ενώ μέχρι και τα 5000 μηνύματα οι χρόνοι ακολουθούν την αναμενόμενη συμπεριφορά, δηλαδή έχουμε μικρότερους χρόνους καθυστέρησης στο QoS 0 λόγω του ότι παρουσιάζει την μεγαλύτερη ταχύτητα εξυπηρέτησης στα 5000 μηνύματα παρατηρούμε τις παρακάτω μετρήσεις:

	QoS 0	QoS 1	QoS 2
Lost Messages (%)	22.367295116	62.499512195	52.7001169591
Avg Time (sec)	31.016564905015464	0.6101815368780714	0.8246474929760299

Σχήμα 9.17: Simulation Results

Με μια πρώτη ματιά, το αποτέλεσμα δεν είναι λογικό μέχρι να παρατηρήσουμε ότι ο λόγος που η γρηγορότερη ουρά παρουσιάζει μεγαλύτερη καθυστέρηση είναι το γεγονός ότι άλλες δύο ουρές χάνουν μεγάλο αριθμό δεδομένων. Αν θεωρητικά είχαμε μια ουρά με άπειρο μήκος και δεν είχαμε απορρίψεις μηνυματων τότε και για τα άλλα δύο QoS θα παρατηρούνταν μεγάλες καθυστερήσεις. Ο τρόπος που έρχονται τα μηνύματα επηρεάζει σε αυτήν την περίπτωση τις μετρήσεις παρουσιάζοντας μια ψευδή αίσθηση ταχύτητας. Θυμίζουμε ότι επειδή μιλάμε για αισθητήρες τα δεδομένα δεν έρχονται απολύτως κατανεμημένα μέσα στην χρονική περίοδο αλλά η είσοδός τους στο σύστημα γίνεται με τρόπο που μπορεί να περιγραφεί καλύτερα ως χρονικές αιχμές (spikes) στις οποίες ταυτόχρονα δεδομένα κατακλύζουν το σύστημα.

9.5 Γενικά συμπεράσματα για την αρχιτεκτονική

Όλα τα παραπάνω ειδικά σενάρια επικεντρώθηκαν στο κομμάτι του Data Source της αρχιτεκτονικής, δηλαδή από την στιγμή που ο αισθητήρας εκπέμπει το μήνυμα του μέχρι αυτό να γραφτεί στον Kafka Server στο τέλος του επιπέδου Data Source. Αυτό έγινε διότι από διάφορα πειράματα που εκτελέσαμε στην αρχιτεκτονική ανακαλύφθηκε ότι το bottleneck του συστήματος μας παρουσιαζόταν σε αυτό το σημείο. Στα παρακάτω επίπεδα η ροή της πληροφορίας γίνεται ανάμεσα σε Kafka Servers και όπως αναφέραμε και προηγουμένως υπάρχουν δικλείδες ασφαλείας για να αποθηκεύονται τα χαμένα μηνύματα όταν αυτά απορρίπτονται από το σύστημα και να ανατροφοδοτούνται σε αυτό αν είναι απαραίτητο.

Επίσης, ο μέσος χρόνος παραμονής ενός μηνύματος στο σύστημα έχει νόημα μόνο στο κομμάτι του πόσο γρήγορα αυτό το μήνυμα μπορεί να τροφοδοτηθεί στην ελεγκτική και επεξεργαστική εφαρμογή του Speed layer δηλαδή μας ενδιαφέρει ο χρόνος παραμονής του στην Data Source βαθμίδα. Αυτό συμβαίνει διότι στην συγκεκριμένη αρχιτεκτονική η έξοδος του Speed layer (τα δεδομένα του Kafka Server του Speed layer) αποτελείται από χρονικά παράθυρα συνεπώς η έννοια του timestamp του μηνύματος χάνεται μέσα στα πολύ πιο γενικά όρια

του παραθύρου. Τέλος, το Serving layer είναι απλά βαθμίδα οπτικοποίησης και αποθήκευσης και δεν παίρνουν μέρος σε αυτό χρονοβόρες διαδικασίες που μπορούμε να βελτιώσουμε αλλάζοντας διάφορες παραμέτρους.

Ύστερα από τις παραπάνω προσομοιώσεις για το MQTT προσπαθήσαμε να παρέμβουμε και στον τρόπο λειτουργίας του Kafka Server αυξάνοντας των αριθμό των clusters, των partition των topics και του replication factor. Λόγω όμως της αδυναμίας του υπολογιστή χρειάστηκε να μεταφερθούμε σε διαφορετικό περιβάλλον εκτέλεσης.

Με τις προσομοιώσεις στο νέο περιβάλλον διαπιστώθηκε πως μπορούμε να βελτιώσουμε τα ποσοστά απόρριψη και τον μέσο χρόνο παραμονής του κάθε μηνύματος ενώ διαχειριζόμαστε ταυτόχρονα πολλά μηνύματα ανά δευτερόλεπτο. Καταφέραμε δηλαδή να πολλαπλασιάσουμε την υπολογιστική δύναμη του συστήματος της αρχιτεκτονικής μας εκτελώντας την σε ανώτερο hardware, ικανό να υποστηρίξει μεγαλύτερη μνήμη και περισσότερους πυρήνες επεξεργασίας. Αυτό μας επέτρεψε να εκμεταλλευτούμε τις δυνατότητες του Kafka Server προσθέτοντας παραπάνω από ένα partitions σε κάθε topic, με αυτόν τον τρόπο οι διαδικασίες εγγραφής της βαθμίδα Sensor Controllers και οι διαδικασίες ανάγνωσης και εγγραφής της βαθμίδα Stream App κατάφεραν να παραλληλοποιηθούν και να προσδώσουν ταχύτητα και αξιοπιστία στο σύστημα. Ακόμα, μεγαλύτερη παραλληλοποίηση και δύναμη επεξεργασίας παρατηρήθηκε όταν η αρχιτεκτονική χρησιμοποίησε παραπάνω από ένα cluster για να εκτελέσει τις λειτουργίες της τόσο στο κομμάτι του Kafka Server και της εφαρμογής Stream όσο και στο κομμάτι του MQTT που χρησιμοποιήθηκε cluster από brokers για την διευκόλυνση της διανομής των μηνυμάτων των αισθητήρων. Τέλος, άξια σημείωσης αποτελεί η παρατήρηση πως το replication factor λειτούργησε κάπως αρνητικά στην ταχύτητα της εφαρμογής αφού τα δεδομένα χρειάστηκε να αποθηκευτούν πάνω από μια φορές σε διαφορετικά κομμάτια των servers, προσέδωσε όμως ασφάλεια στην διαχείριση των δεδομένων και ανοχή σε τυχόν σφάλματα του δικτύου των servers, καθώς η ύπαρξη των δεδομένων σε πολλαπλά σημεία τα προστατεύει από το να χαθούν από το σύστημα.

Κεφάλαιο 10

Συμπεράσματα-Μελλοντικές εφαρμογές

10.1 Σύνοψη και συμπεράσματα

Με όλα τα παραπάνω δεδομένα και τις αναλύσεις που παρατέθηκαν σε κάθε κεφάλαιο είναι φανερό πως δεν υπάρχει μια αρχιτεκτονική του συστήματος που να αποτελεί πανάκεια για κάθε λύση όσον αφορά το επιμέρους fine tuning του MQTT, του Kafka Server και του προγράμματος οπτικοποίησης των δεδομένων.

Ο κορμός της τρέχουσας αρχιτεκτονικής παραμένει ίδιος, όπως όμως σε κάθε εμπορική και επιστημονική καινοτομία ο τελικός χρήστης αποφασίζει την επιτρεπτή αναλογία απόδοσης/κόστους που θέλει να έχει το σύστημα του. Για παράδειγμα, όπως είδαμε στα πειράματα με επιλογή του quality of service 0 στο MQTT υπάρχει ένα σαφές πλεονέκτημα ταχύτητας, παρόλα αυτά στο κεφάλαιο για το MQTT παρουσιάσαμε τα μειονεκτήματα της επιλογής του quality of service 0. Ακόμα, παρατηρήσαμε ότι με την χρήση πολλών partitions και πολλαπλών clusters στον Kafka Server στην αρχιτεκτονική μειώνονται οι απώλειες πακέτων και βελτιώνεται η ταχύτητα, παρόλα αυτά αναφέραμε και τα μειονεκτήματα αυτής της στρατηγικής τονίζοντας τις υπολογιστικές απαιτήσεις που πρέπει να ικανοποιεί το hardware του συστήματος.

Σε κάθε περίπτωση τα μεγάλα πλεονεκτήματα του συστήματος είναι ότι, πρώτον, μπορεί με ελάχιστες αλλαγές να ικανοποιήσει τις ανάγκες σχεδόν όλων των πιθανών περιπτώσεων χρήσης που αφορούν επεξεργασία δεδομένων μετρήσεων από αισθητήρες σε πραγματικό χρόνο και δεύτερον, ότι είναι εξαιρετικά κλιμακώσιμο πράγμα που του επιτρέπει να διαχειρίζεται μεγάλο και αυξανόμενο όγκο πληροφοριών με ελάχιστες αλλαγές στο software.

Παραδείγματα κώδικα: `https://github.com/SouliotisStefanos/Thesis`

10.2 Εκτίμηση μελλοντικών εφαρμογών

Στην παρούσα εργασία μελετήθηκαν αρχιτεκτονικές για επεξεργασία και μετατροπή δεδομένων σε πραγματικό χρόνο. Η παραπάνω όμως προτεινόμενη αρχιτεκτονική, θα μπορούσε να επεκταθεί ώστε να υποστηρίζει και κομμάτι ελεγκτών που δέχονται τα επεξεργασμένα δεδομένα και αντιδρούν στις μετρήσεις με συγκεκριμένους τρόπους. Υπάρχει δηλαδή δυνατότητα, με βασικό κορμό το συγκεκριμένο σύστημα, να δημιουργήσουμε ένα αυτόματα διαχειριζόμενο και ελεγχόμενο ψηφιακό οικοσύστημα το οποίο να αυτορυθμίζεται και να λειτουργεί με πολύ μικρή εξωτερική παρέμβαση. Μπορούμε, λοιπόν, με αυτόν τον τρόπο να φτιάξουμε ένα "έξυπνο" κτίριο ή κτιριακό συγκρότημα με έλεγχο θερμοκρασία, έλεγχο ενεργειακών δαπανών και διαχείριση εκπεμπόμενων ρύπων τον οποίο θα αυτορυθμίζεται μέσα από προσαρμοσμένες διαδικασίες που θα πυροδοτούνται από τις μετρήσεις αισθητήρων του.

Μάλιστα, μπορούμε να εντάξουμε σε αυτό το μοντέλο και μηχανισμούς προβλέψεων που βασίζονται στην μηχανική μάθηση (Machine Learning). Αυτό θα μας επιτρέψει να κατασκευάσουμε ένα σύστημα με τεχνητή νοημοσύνη το οποίο δεν θα αντιδρά απλά στα προβλήματα όταν αυτά εμφανίζονται και ανιχνεύονται από τους αισθητήρες, αλλά θα μπορεί να προβλέπει ορισμένα γεγονότα και να κάνει διορθωτικές κινήσεις στο ψηφιακό οικοσύστημα. Για παράδειγμα, μπορούμε να εκπαιδεύουμε την αρχιτεκτονική μας να διαφοροποιείται ανάλογα με το φυσικό περιβάλλον που βρίσκεται, τις εποχιακές αλλαγές αλλά και σε επαναλαμβανόμενα γεγονότα τα οποία έχουν συγκεκριμένη συχνότητα. Επιπλέον, θα μπορούσαμε να εκπαιδεύσουμε τον ελεγκτικό μηχανισμό της θερμοκρασίας του κτιρίου να διαφοροποιείται ανάλογα με την εποχή, το κλίμα και τις συνήθειες των ενοίκων.

Στην πορεία της παρούσας εργασίας εστιάσαμε στην παραγωγή, μεταφορά και επεξεργασίας ροών δεδομένων τα οποία έχουν μικρό μέγεθος και μορφή value, timestamp). Η αρχιτεκτονική που παρουσιάσαμε παραμένει πρωτοποριακή εφόσον οι εφαρμογές για επεξεργασία live δεδομένων σε έξυπνα κτίρια παρουσιάζουν εμπορική ζήτηση, παρόλα αυτά στις μέρες μας υπάρχει πολύς διαθέσιμος όγκος πληροφορίας σε μορφή εικόνων και video, ο οποίος είναι δύσκολο να επεξεργαστεί και να αναλυθεί με ανθρώπινη παρέμβαση λόγου του τεράστιου όγκου και της συχνότητας έλευσης των νέων δεδομένων.

Λαμβάνοντας αυτό υπόψη, θα ήταν ενδιαφέρον να προσπαθήσουμε να μετατρέψουμε το σύστημά μας έτσι ώστε στο Data Source επίπεδο να δέχεται ροή δεδομένων που αποτελείται από εικόνες και video και να επεξεργάζεται live τα δεδομένα. Σίγουρα για να είναι κάτι τέτοιο δυνατό θα πρέπει να μεταφέρουμε και μεγάλο μέρος των υπολογισμών μας και της επεξεργασίας μας στο edge computing κομμάτι του συστήματός δηλάδη στους ίδιους τους sensors και στους sensors controllers, καθώς πλέον τα δεδομένα μας θα έχουν αρκετά μεγάλο μέγεθος.

Βιβλιογραφία

[1] Cloud Computing History. https://en.wikipedia.org/wiki/Cloud_computing.

[2] Flask-SocketIO. https://flask-socketio.readthedocs.io/en/latest/.

[3] MQTT Packet Format. https://openlabpro.com/guide/mqtt-packet-format/.

[4] The "Only" Coke Machine on the Internet. https://www.cs.cmu.edu/~coke/history_long.txt, 1998.

[5] Fabian Hueske, Vasiliki Kalavri. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. Chapter 2. Stream Processing Fundamentals - Stream Processing with Apache Flink, 2019.

[6] Finematics. Apache Kafka Explained. https://www.youtube.com/watch?v=JalUUBKdcA0&ab_channel=Finematics, 2019.

[7] Wikipedia. Lambda architecture. https://milinda.pathirage.org/kappa-architecture.com/.

[8] Amazon. Amazon Cloud Computing. https://aws.amazon.com/what-is-cloud-computing/, 2020.

[9] Apache Flink Documentation. Flink Deprecated List 1.6. https://ci.apache.org/projects/flink/flink-docs-release-1.6/api/java/index.html?deprecated-list.html.

[10] Apache Flink Documentation . Flink DataStream API Programming Guide. https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html#what-is-a-datastream, 2020.

[11] Apache Flink Documentation. Operators. https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/operators/, 2020.

[12] Apache Flink Documentation. Windows. https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html, 2020.

[13] Apache Kafka Documentation. KAFKA STREAMS ARCHITECTURE. https://kafka.apache.org/26/documentation/streams/architecture, 2020.

[14] Apache Kafka Documentation. Stateful Transformations. https://kafka.apache.org/26/documentation/streams/developer-guide/dsl-api.html#stateful-transformations, 2020.

[15] Apache Kafka Documentation. Stateless Transformation . https://kafka.apache.org/26/documentation/streams/developer-guide/dsl-api.html#stateless-transformations, 2020.

[16] Cloudfare. Edge Computing Explained. https://www.cloudflare.com/learning/serverless/glossary/what-is-edge-computing/.

[17] Confluent. Kafka Streams Concepts. https://docs.confluent.io/3.1.1/streams/concepts.html.

[18] Data Flair. Kafka Streams, Stream, Real-Time Processing & Features. https://data-flair.training/blogs/kafka-streams/.

[19] Dominik Obermaier. Lightweight and scalable IoT Messaging with MQTT. https://www.slideshare.net/dobermai/lightweight-and-scalable-iot-messaging-with-mqtt, 2019.

[20] Edureka. Internet of Things (IoT) Architecture. https://www.youtube.com/watch?v=FRxRT0DjE7A&fbclid=IwAR3fvbogcKDE5MLumkgc7ClrpWg5XzwOJWzzLx8KXXtiPU5CipnfRJi_wpg&ab_channel=edureka, 2018.

[21] Elin Vinka. What is Zookeeper and why is it needed for Apache Kafka? https://towardsdatascience.com/getting-started-with-apache-kafka-in-python-604b3250aa05, 2018.

[22] Eric Griffith. What Is Cloud Computing? https://www.pcmag.com/news/what-is-cloud-computing, 2020.

[23] Eric Knorr. Cloud Computing Explained. https://www.infoworld.com/article/2683784/what-is-cloud-computing.html, 2018.

[24] Fabian Hueske, Vasiliki Kalavri. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. Chapter 1. Introduction to Stateful Stream Processing - Stream Processing with Apache Flink, 2019.

[25] Fabian Hueske, Vasiliki Kalavri. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. Chapter 3. The Architecture of Apache Flink - Stream Processing with Apache Flink, 2019.

[26] Fabian Hueske, Vasiliki Kalavri. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. Chapter 5. The DataStream API (v1.7) - Stream Processing with Apache Flink, 2019.

[27] Frank Kane . Intro to Spark Streaming . https://www.youtube.com/watch?v=dYBWZTZT6o0&ab_channel=FrankKane, 2016.

[28] Hamdamboy. Message queuing telemetry transport (mqtt) message format. https://www.slideshare.net/HamdamboyUrunov/message-queuing-telemetry-transport-mqtt-message-format, 2017.

[29] IBM. IBM Cloud Computing. https://www.ibm.com/cloud/learn/cloud-computing, 2020.

[30] ipc2u. Τι είναι το MQTT και γιατί το χρειαζόμαστε στο IIoT; Περιγραφή του πρωτοκόλλου MQTT. https://cy.ipc2u.com/articles/articles-and-reviews/ti-einai-to-mqtt-kai-giati-to-chreiazomaste-sto-iiot/.

[31] Jad Jebara, Rajan Sodhi. Edge Computing vs Cloud Computing. https://www.computer.org/publications/tech-news/data-center-insider/edge-computing-vs-cloud-computing.

[32] Jake Frankenfield. Cloud Computing Explained. https://www.investopedia.com/terms/c/cloud-computing.asp, 2020.

[33] K. MANI CHANDY, LESLIE LAMPORT. Distributed Snapshots: Determining Global States of Distributed Systems. https://lamport.azurewebsites.net/pubs/chandy.pdf, 1985.

[34] Karen Rose, Scott Eldridge, Lyman Chapin. The internet of things: An Overview. Understanding the Issues and Challenges of a More Connected World. https://www.internetsociety.org/wp-content/uploads/2017/08/ISOC-IoT-Overview-20151221-en.pdf, 2015.

[35] Keith D. Foote. A Brief History of the Internet of Things. https://www.dataversity.net/brief-history-internet-things/, 2016.

[36] Lakshmi Deepa S. Understanding MQTT Protocol Packet Format . https://www.bevywise.com/blog/understanding-mqtt-protocol-packet-format/, 2020.

[37] Microsoft. Microsoft Azure Cloud Computing. https://azure.microsoft.com/en-in/overview/what-is-cloud-computing/#cloud-deployment-types, 2020.

[38] milinda.pathirag . Kappa Architecture. https://milinda.pathirage.org/kappa-architecture.com/, 2020.

[39] Mohan Kumar. Internet-of-things-(IOT). https://www.slideshare.net/MohanKumarG/internetofthings-iot-aseminar-ppt-by-mohankumarg?fbclid=IwAR2JshIFVzSUF55zqTiVCDIL1KblnXTOQQlPsG1GLxyyWTRnHAOkCEHTxPM, 2017.

[40] Nadhif Ikbar Wibowo. The Future of IoT-Based Energy Usage Management. https://steemit.com/robotina/@nadhifikbarw/the-future-of-iot-based-energy-usage-management, 2017.

[41] Pallavi Sethi, Smruti R. Sarangi. Internet of Things: Architectures, Protocols, and Applications. https://www.hindawi.com/journals/jece/2017/9324035/, 2017.

[42] Paul Miller. Edge Computing Benefits. https://www.theverge.com/circuitbreaker/2018/5/7/17327584/edge-computing-cloud-google-microsoft-apple-amazon, 2018.

[43] Simplilearn. Spark Streaming Tutorial, Spark Streaming Example, Spark Tutorial. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm, 2019.

[44] Steve Ranger. Cloud Computing Explained. https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/, 2018.

[45] STL Partners. Edge Computing Use Cases. https://stlpartners.com/edge-computing/what-is-edge-computing/.

[46] Tara Salman. Internet of Things Protocols and Standards. https://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_prot/.

[47] Tutorialspoint. Apache Spark. https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm.

[48] Tutorialspoint. Apache Spark - RDD. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm.