Σύστημα IoT live-streaming

Ομάδα 14

Μαρία-Ηώς Γλάρου
ΗΜΜΥ
ΕΜΠ
03118176
Αθήνα, Ελλάδα
emiglarou@gmail.com

Ανδρέας Νικόλαρος
ΗΜΜΥ
ΕΜΠ
03118401
Αθήνα, Ελλάδα
nikolarosandreas@gmail.com

Παναγιώτης Ντάγκας
ΗΜΜΥ
ΕΜΠ
03118018
Αθήνα, Ελλάδα
panosdagas2000@hotmail.com

Abstract: ***Στα πλαίσια του μαθήματος Ανάλυση και Σχεδιασμός Πληροφοριακών Συστημάτων χρησιμοποιήθηκαν συνδυαστικά εργαλεία ανοιχτού κώδικα και κατασκευάστηκε ένα live streaming σύστημα που είναι το πρωτότυπο ενός πραγματικού IoT συστήματος.***

Keywords: IoT, Big Data, live-streaming, message brokers, simulation

I. ΣΚΟΠΟΣ

Ο σκοπός της παρούσας εργασίας είναι η υλοποίηση ενός πρωτοτύπου ενός IoT συστήματος, το οποίο λαμβάνει, επεξεργάζεται και απεικονίζει γραφικά μετρήσεις που μεταδίδονται από αισθητήρες σε οικιακό περιβάλλον.

Πιο συγκεκριμένα υλοποιήθηκε προσομοίωση δεδομένων, μέσω κώδικα, τα οποία αντιστοιχούν σε δεδομένα από δέκα αισθητήρες κίνησης, θερμοκρασίας, ηλεκτρικών συσκευών και νερού.

Τα δεδομένα αυτά μεταβιβάζονται στο σύστημα με τη χρήση message broker για την μετέπειτα επεξεργασία και αναπαραγωγή τους. Ακολουθήθηκε μοντέλο το οποίο καθιστά ανεξάρτητη τη μεταβίβαση κάθε είδους μετρήσεων, ώστε να ελαχιστοποιούνται απώλειας στην περίπτωση διακοπής σύνδεσης ενός broker.

Στη συνέχεια κατασκευάστηκε στρώμα ζωντανής επεξεργασίας/μετάδοσης των δεδομένων των αισθητήρων ώστε να υπολογίζονται ημερήσιες καταναλώσεις ρεύματος και νερού, μέση θερμοκρασία ημέρας καθώς και τυχόν διαρροές σε νερό και ενέργεια.

Όλες οι μετρήσεις αποθηκεύονται τόσο στην αρχική όσο και στην επεξεργασμένη μορφή τους σε timeseries βάση δεδομένων ενώ τα αποτελέσματα των μετρήσεων απεικονίζονται κατάλληλα σε γραφικό περιβάλλον με τη μορφή διαγραμμάτων.

Ιδιαίτερη προσοχή δόθηκε σε ετεροχρονισμένα δεδομένα, τα οποία είναι πιθανό να υπάρξουν από κάποια απώλεια δικτυακής σύνδεσης κάποιου αισθητήρα. Αποφασίστηκε η εφαρμογή κατωφλίου μέσω του οποίου γίνεται διαλογή των ετεροχρονισμένων δεδομένων τα οποία λαμβάνονται υπόψιν (έως 2 ημέρες προγενέστερα), ενώ τα υπόλοιπα φιλτράρονται, απορρίπτονται και αποθηκεύονται σε ξεχωριστό πίνακα ούτως ώστε να είναι διαθέσιμα για τυχόν ελέγχους του συστήματος.

Τελικός στόχος είναι η κατασκευή ενός συστήματος πραγματικού χρόνου, επεκτάσιμου και κλιμακούμενου, το οποίο δύναται να κατασκευαστεί σε μεγάλες ποσότητες και να βγει στην αγορά. Το σύστημα αυτό θα δίνει τη δυνατότητα στον καταναλωτή να ελέγχει τις συσκευές της εγκατάστασής του απομακρυσμένα. Με τον τρόπο αυτό μπορούν να ανιχνευθούν σε σύντομο χρονικό διάστημα διαρροές, οι οποίες σε διαφορετική περίπτωση θα φαίνονταν μόνο σε επόμενους λογαριασμούς νερού ή ρεύματος, με αποτέλεσμα την εξοικονόμηση ενέργειας και χρήματος.

II. ΠΕΡΙΓΡΑΦΗ ΥΠΟΔΟΜΗΣ ΚΑΙ ΛΟΓΙΣΜΙΚΟΥ

Το σύστημα υλοποιήθηκε σε περιβάλλον Windows 10.

Χρησιμοποιήθηκε γλώσσα προγραμματισμού Python, έκδοση 3.10.4, για την κατασκευή του κώδικα ο οποίος προσομοιώνει τα δεδομένα των αισθητήρων.

Ο message broker που χρησιμοποιήθηκε είναι ο mosquitto, έκδοση 2.0.15, στον οποίο υλοποιήθηκαν τρεις ξεχωριστοί brokers(ενέργεια, νερό, θερμοκρασία-κίνηση) σε τρεις διαφορετικές θύρες. Ο συγκεκριμένος broker χειρίζεται τα μηνύματα μέσω του πρωτοκόλλου MQTT. Το πρωτόκολλο αυτό παρέχει μία αποτελεσματική και lightweight μέθοδο ανταλλαγής μηνυμάτων χρησιμοποιώντας το publish-subscribe μοντέλο. Ταυτόχρονα δίνει τη δυνατότητα κρυπτογράφησης για την ασφαλέστερη μετάδοση των μηνυμάτων[1].

Το γραφικό περιβάλλον MQTT explorer χρησιμοποιήθηκε για τον άμεσο και ευκολότερο έλεγχο των συνδέσεων σε σχέση με την απλή γραμμή εντολών.

Υπήρξε η δυνατότητα επιλογής μεταξύ των mosquitto MQTT και rabbit MQ. Τελικά επιλέχθηκε το mosquitto καθώς είναι σχεδιασμένο αποκλειστικά για συσκευές IoT και είναι ιδανικό για συσκευές χαμηλής κατανάλωσης οι οποίες στέλνουν μηνύματα σε ένα ευρυζωνικά περιορισμένο δίκτυο, σε αντίθεση με το rabbit MQ που είναι πρωτόκολλο γενικού σκοπού. Επίσης, το mosquitto MQTT δίνει τη δυνατότητα απλούστερης υλοποίησης ενός publisher/client σε σύγκριση με το rabbit MQ [2]. Δίνεται επιπλέον η δυνατότητα επιλογής quality of service για την ελαχιστοποίηση χαμένων μηνυμάτων.

Για τη σύνδεση μεταξύ του στρώματος συσκευών και των brokers χρησιμοποιήθηκε Paho client, ο οποίος παρέχει τη δυνατότητα σύνδεσης του script ως publisher/client στους διάφορους brokers. Συγκεκριμένα επιτρέπει την εγγραφή στα διάφορα topics τόσο για δημοσίευση όσο και για λήψη δημοσιευμένων μηνυμάτων.

Για το στρώμα ζωντανής μετάδοσης/επεξεργασίας χρησιμοποιήθηκε το εργαλείο ανοιχτού κώδικα Apache Spark, έκδοση 2.4.0, σε συνδυασμό με Hadoop, έκδοση 2.7. Το Apache Spark παρέχει ήδη έτοιμα frameworks τα οποία καθιστούν ευκολότερη την επεξεργασία μεγάλου όγκου

δεδομένων. Παράλληλα παρέχεται υψηλού επιπέδου API για πολλές γλώσσες προγραμματισμού, μεταξύ αυτών και η Python με τη Scala που χρησιμοποιήθηκαν ενώ υποστηρίζει και εφαρμογές σε cloud.

Το μεγαλύτερο προτέρημα του Spark είναι το ότι πραγματοποιεί in-memory υπολογισμούς με αποτέλεσμα την πολύ μεγάλη ταχύτητά του. Προσφέρει ευελιξία υποστηρίζοντας μεγάλο εύρος πηγών δεδομένων μεταξύ των οποίων τα Hadoop Distributed File System (HDFS) και Apache Kafka καθώς και το πρωτόκολλο MQTT. Η πρόσβαση στα δεδομένα από κάθε πηγή γίνεται μέσω μίας καθολικής διεπαφής γεγονός που το καθιστά κατάλληλο για Big Data συστήματα.

Το Apache Spark χρησιμοποιήθηκε πάνω σε Hadoop για τη λήψη των δεδομένων από τους brokers, την επεξεργασία για τον υπολογισμό των μέσων καταναλώσεων και των διαρροών σε πραγματικό χρόνο. Η σύνδεση με τους brokers έγινε με τον connector που παρέχεται από την κλάση *org.apache.bahir.sql.streaming.mqtt.MQTTStreamSourcePr ovider*. Έγινε χρήση του structured streaming interface που προσφέρεται από το Apache Spark για την επεξεργασία των δεδομένων με την μορφή dataframes που εμπλουτίζονται σε πραγματικό χρόνο με νέα δεδομένα.

Η βάση δεδομένων που χρησιμοποιήθηκε για την αποθήκευση τους είναι η Mongo DB με χρήση του timeseries plugin. Η MongoDB είναι μια ευρέως διαδεδομένη NoSQL βάση δεδομένων, σχεδιασμένη για την ευέλικτη και κλιμακούμενη αποθήκευση και ανάκτηση δεδομένων μεγάλου όγκου.

Συγκεκριμένα χρησιμοποιήθηκε συλλογή δεδομένων μορφής timeseries, δηλαδή μίας εξειδικευμένης μορφής για την αποθήκευση και αναζήτηση δεδομένων τα οποία χαρακτηρίζονται από κάποια χρονική σφραγίδα (timestamp).

Η timeseries βάση δεδομένων είναι σχεδιασμένη με τέτοιο τρόπο ώστε να αξιοποιεί τη χρονική σφραγίδα των δεδομένων ώστε να κάνει εύκολη την αναζήτηση τους για υπολογισμούς όπως η παρατήρηση αλλαγών στο χρόνο, η απόδοση κάποιου συστήματος και η ανίχνευση μη αναμενόμενων τιμών σε πραγματικό χρόνο.

Τα πλεονεκτήματα που προσφέρονται από μία timeseries βάση δεδομένων, όσων αφορά τα σειριακά δεδομένα αισθητήρων όπως αυτά που λαμβάνονται είναι τα εξής:

- Είναι κατασκευασμένη και προσαρμοσμένη ακριβώς πάνω σε δεδομένα με χρονική σφραγίδα.
- Καθιστά εύκολη την αναζήτηση με τη χρήση queries που βασίζονται στον περιορισμό του χρονικού διαστήματος (π.χ. σε ημερήσια βάση).
- Παρέχει ενσωματωμένες συναρτήσεις για τον υπολογισμό μέσων όρων και συγκεντρωτικών καταναλώσεων (aggregation functions) βάσει δοθέντος χρονικού διαστήματος.
- Δύναται να αποθηκεύσει μεγάλο όγκο χρονοσειριακών δεδομένων όπως αυτά που παρέχονται από τους αισθητήρες.
- Χρησιμοποιεί τρόπους συμπίεσης των δεδομένων με αποτέλεσμα των περιορισμό των απαιτούμενων πόρων.

Τελικά, στη βάση αποθηκεύονται τόσο τα δεδομένα με την αρχική τους μορφή όσο και τα aggregated καθώς και τα δεδομένα καθυστερημένης άφιξης (σε ξεχωριστούς πίνακες).

Επιπλέον χρησιμοποιήθηκε η υπηρεσία MongoDB Atlas η οποία παρέχει υπηρεσίες cloud για τη βάση δεδομένων. Με αυτό τον τρόπο το τελικό προϊόν γίνεται κλιμακώσιμο και διευκολύνεται η μετέπειτα γραφική απεικόνιση τον δεδομένων.

Καθώς η απευθείας σύνδεση μεταξύ Apache Spark και MongoDB δεν ήταν δυνατή χρησιμοποιήθηκε το λογισμικό confluent, έκδοση 6.1.0, το οποίο παρέχει περιβάλλον με Apache Kafka brokers σε συνδυασμό με έτοιμα plugins τα οποία επιτρέπουν τη σύνδεση του Apache Kafka με την MongoDB μέσω sink connectors. Για τον sink connector της MongoDB χρησιμοποιήθηκε το plugin *com.mongodb.kafka.connect.MongoSinkConnector [3]*.

Το confluent απαιτεί λογισμικό linux, ωστόσο παρέχεται η δυνατότητα χρήσης του σε περιβάλλον Windows μέσω του WSL2 (Windows Subsystem for Linux), ενώ μέσω Powershell scripting επιτράπηκε η δικτυακή κίνηση προς το Kafka το οποίο τρέχει στο εικονικό περιβάλλον του WSL2.

Τα aggregated δεδομένα, όπως και τα δεδομένα καθυστερημένης άφιξης, οδηγούνται στα αντίστοιχα Kafka topics και από εκεί με τη χρήση των sink connectors στις κατάλληλες timeseries συλλογές.

Όσον αφορά τα μη επεξεργασμένα (raw) δεδομένα που λαμβάνονται από τους αισθητήρες, δημιουργήθηκε μία γέφυρα, σε Python με τη χρήση Paho client και Kafka client, μεταξύ MQTT και Apache Kafka για να αποθηκευτούν στη συνέχεια στη MongoDB με χρήση των αντίστοιχων sink connectors.

Τελευταίο βήμα είναι η απεικόνιση των μετρήσεων, επεξεργασμένων και μη, σε γραφικό περιβάλλον. Το γραφικό περιβάλλον που χρησιμοποιήθηκε είναι το Grafana.

Το Grafana είναι μια πλατφόρμα ανοιχτού κώδικα που εξειδικεύεται στην ανάλυση και οπτικοποίηση δεδομένων με τη χρήση διαδραστικών πινάκων. Δίνει τη δυνατότητα για:

- Συγχώνευση δεδομένων διαφορετικής προέλευσης (βάσεις δεδομένων, cloud).
- Δημιουργία διαδραστικών πινάκων για απεικόνιση δεδομένων πραγματικού χρόνου.
- Ειδοποιήσεις σε περίπτωση που κάποια μέτρηση ξεπεράσει κάποιο κατώφλι.
- Πληθώρα διαφορετικών plugins.

Συγκεκριμένα χρησιμοποιήθηκε το MongoDB DataSource plugin το οποίο επιτρέπει τη σύνδεση του Grafana με τη βάση στο cloud μέσω ενός string identifier για την πραγματοποίηση queries και απεικόνιση των δεδομένων.

Για τη χρήση της enterprise έκδοσης του συγκεκριμένου plugin, η οποία ήταν αναγκαία για τη σύνδεση, λήψη και απεικόνιση των δεδομένων, χρησιμοποιήθηκε το free trial 14 ημερών που δίνεται από την εταιρεία.

Το Grafana όπως και η MongoDB βρίσκεται σε cloud και δεν τρέχει τοπικά. Παράχθηκε dashboard με πίνακες και charts για την απεικόνιση μη επεξεργασμένων και aggregated δεδομένων.

III. Περιγραφη των βηματων για το στησιμο του συστηματος και των πειραματων

Ο κώδικας βρίσκεται εδώ:

A. *Device Layer:*

Για να υλοποιηθεί το στρώμα συσκευών χρησιμοποιήθηκε Python scripting, ώστε να προσομοιωθεί η λειτουργία ενός αληθινού οικιακού περιβάλλοντος με τους αισθητήρες που αναφέρθηκαν στο προηγούμενο κεφάλαιο. Επομένως, δημιουργήθηκε το αρχείο datascript.py, το οποίο βρίσκεται στον φάκελο DeviceLayer του Github repository.

Με τη χρήση της βιβλιοθήκης Paho MQTT δημιουργούνται clients, οι οποίοι συνδέονται στους brokers κάνοντας subscribe στα αντίστοιχα topics των brokers που ανάλογα με τη μέτρηση (ενέργεια, νερό, θερμοκρασία/κίνηση) ανοίγουν σε διαφορετικές πόρτες του localhost [4]. Στη προσομοίωση οι ημερομηνίες όλων των δεδομένων ξεκινούν από την 01-01-2023 και ώρα 00:00. Υλοποιήθηκε μία λούπα η οποία τρέχει επ' αόριστον και σε κάθε επανάληψη της δημιουργεί εικονικές μετρήσεις με τυχαίο τρόπο στο εύρος τιμών που ζητείται για κάθε αισθητήρα χρησιμοποιώντας τη συνάρτηση *random.uniform* της python. Στο τέλος της λούπας χρησιμοποιείται η συνάρτηση *sleep* της βιβλιοθήκης *time* ώστε η λούπα να επαναλαμβάνεται ανά 1 δευτερόλεπτο. Ζητείται το 1 δευτερόλεπτο της προσομοίωσης να αντιστοιχεί σε 15 λεπτά πραγματικού χρόνου επομένως μετά το πέρας του *sleep* αυξάνεται η χρονική σφραγίδα που αποστέλλεται κατά 15 λεπτά με χρήση της συνάρτησης *timedelta* της βιβλιοθήκης *datetime* της python.

Παράλληλα, χρησιμοποιείται και ένας δείκτης ώστε ανά 96 λούπες, που αντιστοιχούν σε μία ημέρα σε πραγματικό χρόνο, να αποστέλλονται και οι ημερήσιες μετρήσεις Etot, Wtot, οι οποίες αντιστοιχούν σε συνολικές αθροιστικές μετρήσεις ενέργειας και νερού και επομένως μία νέα τιμή στο διάστημα που αναφέρεται στην εκφώνηση προστίθεται στην προϋπολογισμένη τιμή των Etot, Wtot. Για τον αισθητήρα κίνησης δημιουργούνται 5 ψευδοτυχαίοι αριθμοί πριν την αρχή μίας ημέρας στο διάστημα 0 έως και 95 και αν ο μετρητής έχει την τιμή ενός από αυτούς, τότε αποστέλλεται και η μέτρηση κίνησης με τιμή 1 (ανίχνευση κίνησης).

Τα δεδομένα ενός αισθητήρα αποστέλλονται στο αντίστοιχο topic με τη μορφή: <timestamp> | <value> σε μορφή string και όχι σε json για λόγους ταχύτητας, αφού το μήνυμα με απλό string έχει μικρότερο μέγεθος payload.

Τέλος, ανά 20 δευτερόλεπτα, δηλαδή 5 ώρες πραγματικού χρόνου αποστέλλεται και ένα ετεροχρονισμένο δεδομένο για τον αισθητήρα νερού W1, το οποίο αντιστοιχεί σε μέτρηση δύο ημερών πριν από την χρονική σφραγίδα που έχει εκείνη τη στιγμή η προσομοίωση. Το ίδιο συμβαίνει και ανά 120 δευτερόλεπτα, δηλαδή 30 ώρες πραγματικού χρόνου για ετεροχρονισμένα δεδομένα 10 ημερών. Αυτή η διαδικασία υλοποιείται με τον έλεγχο του μετρητή που προαναφέρθηκε.

Να σημειωθεί, πως εφόσον έχει επιλεγεί στο στρώμα μηνυμάτων να χρησιμοποιηθεί Quality of Service με τιμή 1 για το πρωτόκολλο MQTT ο paho client πρέπει να περιμένει να λάβει ένα μήνυμα αναγνώρισης (ACK) από τον broker για να συνεχίσει, αλλιώς ξαναστέλνει την μέτρηση και προχωράει στις επόμενες. Επιπλέον, χρησιμοποιούνται 3 διαφορετικοί clients, όπου ο καθένας στέλνει δεδομένα στον αντίστοιχο broker (Energy, Water, Temp).

Επειδή η εργασία έχει καθαρά χαρακτήρα προσομοίωσης, τα ετεροχρονισμένα δεδομένα 10 ημερών αποστέλλονται με ένα είδους αναγνωριστικό (flag), ώστε να φιλτράρονται κατάλληλα. Αυτή η σχεδιαστική επιλογή προέκυψε από προβλήματα που εντοπίστηκαν σε επόμενο στρώμα και θα αναλυθούν.

B. *Messaging Broker Layer:*

Χρησιμοποιήθηκε το Mosquitto MQTT, όπως προαναφέρθηκε για την υλοποίηση αυτού του στρώματος. Συγκεκριμένα, δημιουργήθηκαν 3 messaging brokers στο ίδιο σύστημα (localhost) που ανοίγουν σε διαφορετικές πόρτες. Η προκαθορισμένη πόρτα του Mosquitto είναι η 1883 η οποία ξεκινάει ως υπηρεσία στο μηχάνημα κατά την εκκίνηση, ενώ δημιουργήθηκαν και άλλα δύο αρχεία .conf στο φάκελο που έχει εγκατασταθεί το mosquito ώστε να σηκώνονται και άλλοι δύο brokers στις πόρτες 1884, 1885 αντίστοιχα με την εντολή *mosquitto -c <path_to_config_file>*.

Επιλέχθηκε να χρησιμοποιηθούν 3 brokers για να επιτευχθεί μεγαλύτερη κλιμακωσιμότητα του συστήματος καθώς και για την καλύτερη διαχείριση των topics, ώστε κάθε broker να αναλαμβάνει ένα είδους μέτρησης (Energy, Water, Temperature/Movement). Σε ένα πραγματικό σύστημα που θα έτρεχε στο Cloud θα ήταν η καλύτερη επιλογή, ώστε αν για κάποιον λόγο ένας broker σταματήσει να λειτουργεί να μην διαλυθεί ολόκληρο το σύστημα.

Επιλέχθηκε, επιπλέον, να δημιουργηθεί ένα topic ανά μέτρηση-αισθητήρα. Αυτό δεν είναι η καλύτερη πρακτική σε ένα πραγματικό σύστημα, καθώς δεν είναι κλιμακώσιμη. Ωστόσο, στην εργασία υπάρχει ένας μικρός σχετικά αριθμός από αισθητήρες, επομένως υπάρχει η δυνατότητα για μια τέτοια επιλογή. Σε ένα πραγματικό σύστημα, θα ήταν καλύτερη υλοποίηση να ομαδοποιηθούν κάποιοι αισθητήρες σε topics και από το DeviceLayer να στέλνουν και το δικό τους αναγνωριστικό, ώστε ένα topic να περιέχει για παράδειγμα όλες τις μετρήσεις της ενέργειας, ή μέρος αυτών.

Εδώ, χρησιμοποιήθηκε ο broker στην πόρτα 1883 για τα δεδομένα της ενέργειας, όπου δημιουργήθηκε ένα topic ανά αισθητήρα, συγκεκριμένα τα topics:

- IoT/energy/Airconditioning/HVAC1
- IoT/energy/Airconditioning/HVAC2
- IoT/energy/Rest_appliances/MiAC1
- IoT/energy/Rest_appliances/MiAC2
- IoT/energy/Etot

Ο broker στην πόρτα 1884 για τα δεδομένα του νερού με τα αντίστοιχα topics:

- IoT/water/W1
- IoT/water/Wtot

Ο broker στην πόρτα 1885 για τα δεδομένα της θερμοκρασίας και της κίνησης με τα αντίστοιχα topics:

- IoT/temperature/TH1
- IoT/temperature/TH2
- IoT/movement/Mov1

Αυτή η προσέγγιση δίνει τη δυνατότητα στο Device Layer να αποστέλλει τα δεδομένα χωρίς το αναγνωριστικό τους και επομένως να στέλνονται με μικρότερο payload, αφού η μέτρηση αντιστοιχίζεται άμεσα με το topic στο οποίο γράφεται. Πέρα από αυτή τη σχεδιαστική επιλογή, αποφασίστηκε να χρησιμοποιηθεί Quality of Service με τιμή 1 για τη μεταφορά των δεδομένων. Αυτό έγινε, καθώς αν το QoS είναι 0 δεν υπάρχει καμία εξασφάλιση από τον broker ότι τα μηνύματα έχουν παραδοθεί. Από την άλλη το QoS με τιμή 2 θα επαναλαμβάνει την αποστολή των δεδομένων επ' αόριστον μέχρι να λάβει επιβεβαίωση από τον broker με αποτέλεσμα να δημιουργηθεί πολύ μεγάλη κίνηση στο δίκτυο κάτι που είναι ανεπιθύμητο. Επομένως, η καλύτερη επιλογή ήταν το QoS με τιμή 1, ώστε αν δεν ληφθεί επιβεβαίωση την πρώτη φορά από τον broker να γίνει μόνο μία επαναμετάδοση και μετά το Device Layer να συνεχίσει κανονικά τη λειτουργία του. Έτσι, υπάρχει και ένας τρόπος αντιμετώπισης χαμένων μηνυμάτων, αλλά αποφεύγεται και η πλημμύρα του δικτύου με μηνύματα. Εξ' άλλου, τα δεδομένα που στέλνονται δεν απαιτούν τόσο αυστηρή μεταχείριση των χαμένων μηνυμάτων και το σύστημα θα μπορέσει να λειτουργήσει αν τυχόν λίγα πακέτα χαθούν. Αυτό υλοποιήθηκε προσθέτοντας στα .conf αρχεία την επιλογή *max qos 2 [5]*.

Επίσης, τέθηκαν οι μεταβλητές max_inflight_bytes και max_inflight_messages σε τιμή 0, ώστε να μην περιορίζεται ο αριθμός των bytes/μηνυμάτων που μπορούν να αποσταλούν χωρίς να ληφθεί επιβεβαίωση. Επίσης, αποφασίστηκε να χρησιμοποιηθεί persistency, ώστε αν ένας client είναι αποσυνδεδεμένος για κάποιον λόγο την ώρα που στάλθηκε το μήνυμα και μετά ξανασυνδεθεί να μην χάσει κάποια μέτρηση. Γι' αυτό τον λόγο τέθηκε τη μεταβλητή persistence σε true και το persistent_client_expiration σε 2 λεπτά. Επιπλέον, επιτράπηκε η σύνδεση όλων των clients από το τοπικό μηχάνημα στους brokers θέτοντας τη μεταβλητή allow_anonymous σε true. Όλα τα .conf αρχεία για το mosquitto μπορούν να βρεθούν στον φάκελο MQTTBrokers στο github repository.

Να σημειωθεί, πως οι clients που στέλνουν τα δεδομένα δεν είναι persistent, καθώς αυτό δεν υποστηριζόταν από τη βιβλιοθήκη Paho MQTT για την python.

Γ. Live Streaming Layer:

Για το στρώμα ζωντανής επεξεργασίας/μετάδοσης χρησιμοποιήθηκε το Structured Streaming του Apache Spark 2.4.0 πάνω σε Hadoop 2.7 [6]. Ήταν αναγκαίο να χρησιμοποιηθεί μία έκδοση 2 του Spark, καθώς για να γίνει η σύνδεση με τους message brokers του Mosquitto χρειάζεται ο connector που παρέχεται από την κλάση: *org.apache.bahir.sql.streaming.mqtt.MQTTStreamSourceProvider*

που ανήκει στο bahir package *org.apache.bahir:spark-sql-streaming-mqtt_2.11:2.4.0 [7]*. Το συγκεκριμένο package δεν είναι δυνατόν να χρησιμοποιηθεί με κάποια πιο καινούρια έκδοση του Spark, πράγμα που δημιουργήσαμε μερικά προβλήματα στην υλοποίηση του στρώματος.

Αρχικά, έγινε εγκατάσταση του Apache Spark 2.4.0 – Hadoop 2.7 με χρήση των binaries που παρέχονται από την επίσημη σελίδα του Apache Spark. Επίσης, αναγκαία είναι η χρήση της Java 8 από το Spark, επομένως η μεταβλητή περιβάλλοντος JAVA_HOME τέθηκε να είναι αυστηρά στο path της Java 8. Στη συνέχεια, αντιμετωπίστηκαν κάποια προβλήματα με το Hadoop, καθώς διαπιστώθηκε πως για να τρέξει σε windows χρειάζεται να προστεθούν στον φάκελο bin του Spark δύο αρχεία από το github repo [8] των winutils στον φάκελο της αντίστοιχης έκδοσης του Hadoop 2.7, συγκεκριμένα τα winutils.exe και hadoop.dll.

Επιπλέον, χρειάστηκε να προστεθούν χειροκίνητα κάποια jar αρχεία για το Hadoop ώστε να καταφέρει να ξεκινήσει σε windows σε συνδυασμό με το package του bahir. Αυτά προστέθηκαν στο path C:\Users\<user>\.iv2\jars για να ξεπεραστεί το πρόβλημα. Αυτά τα αρχεία υπάρχουν στον φάκελο windows_essentials που βρίσκεται στον φάκελο LiveStreamingLayer του github repository. Για να τρέξει κάποιος τον κώδικα του Live Streaming Layer που υπάρχει στον αντίστοιχο φάκελο μπορεί να χρησιμοποιήσει την εντολή:

spark-shell --packages org.apache.bahir:spark-sql-streaming-mqtt_2.11:2.4.0,org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.0 -i <filename>

ώστε να συμπεριλάβει τα packages του bahir για τον MQTT connector και τον Kafka connector που χρειάζεται για το επόμενο στρώμα.

Όσον αφορά την υλοποίηση του κώδικα για τα aggregations χρησιμοποιήθηκε η γλώσσα Scala, καθώς είναι πιο συμβατή με το Spark σε σχέση με την Python (pyspark). Ο κώδικας χωρίστηκε σε 4 μέρη: Energy, Water, Temp, Leak. Τα αρχεία Energy, Water, Temp λαμβάνουν τα δεδομένα από τον αντίστοιχο broker και εκτελούν τα ζητούμενα aggregations, ενώ στο αρχείο Leak.scala έχει υλοποιηθεί το aggregation διαρροής για τους αισθητήρες νερού. Οι κώδικες τρέχουν τοπικά σε ένα μηχάνημα και επομένως η υλοποίηση δεν είναι clustered. Ωστόσο, σε ένα πραγματικό σύστημα η καλύτερη επιλογή θα ήταν να υλοποιηθεί σε cluster, ώστε κάθε node να αναλαμβάνει τα aggregations μιας κατηγορίας (πχ. Ενέργειας) και να υπάρχει ένας master node που θα κατανέμει τον φόρτο εργασίας ανάλογα στο κάθε node. Αυτή η επιλογή θα έκανε το σύστημα πιο κλιμακώσιμο και πιο γρήγορο.

Αρχικά, για τη σύνδεση και τη λήψη των δεδομένων από το αντίστοιχο broker topic χρησιμοποιείται η *readStream* της spark με την κλάση του *MQTTStreamSourceProvider* με option "topic" το όνομα του topic, option "QoS" με τιμή 1 και με option "persistence" με τιμή "memory", ώστε ο client να είναι persistent και να λαμβάνει τα μηνύματα με quality of service. Επίσης, για τις ημερομηνίες που λαμβάνονται τέθηκε η μεταβλητή του spark session: spark.conf σε timeZone UTC, καθώς τα δεδομένα έρχονται σε timestamp ζώνης UTC, ενώ το Spark χρησιμοποιούσε UTC+2, που είναι η τοπική ζώνη ώρας και έτσι υπήρχαν διαφορές στη μετατροπή των timestamps.

Για τα aggregations των δεδομένων, εφόσον χρησιμοποιείται το Structured Streaming, επιλέχθηκαν τα window aggregations, αφού προσφέρουν εύκολο interface για ομαδοποίηση δεδομένων με βάση την ημερομηνία, αλλά και φιλτράρισμα ετεροχρονισμένων δεδομένων. Επομένως, τα δεδομένα που λαμβάνονται από την readStream, αρχικά «καθαρίζονται» και δημιουργείται ένα dataframe με 2 στήλες, τις DateTime (για το timestamp που δημιουργήθηκαν τα δεδομένα) και Value για την τιμή της μέτρησης. Ειδική περίπτωση υπάρχει στην υλοποίηση Water.scala για το aggregation του W1 αισθητήρα, όπου προτού γίνει κάποιο

aggregation, ωστόσο, γίνεται έλεγχος για late events που καταφτάνουν 10 ημέρες αργότερα. Στο συγκεκριμένο σύστημα, αυτό γίνεται με τον εντοπισμό του flag που έχει σταλεί από το Device Layer πάνω στα ετεροχρονισμένα δεδομένα 10 ημερών. Δηλαδή, φιλτράρονται όλα τα entries του dataframe, ώστε να κρατηθούν στο dataframe όσα δεν εμπεριέχουν κάποιο τέτοιο flag, ενώ αυτά που το περιέχουν αποθηκεύονται σε ξεχωριστό dataframe και γράφονται τελικά σε διαφορετικό topic στο Kafka. Σε ένα πραγματικό σύστημα, τα ετεροχρονισμένα δεδομένα μπορούν να φιλτραριστούν εύκολα με τον έλεγχο του current timestamp που παρέχει το spark. Ωστόσο, εφόσον υλοποιήθηκε προσομοίωση αυτό δεν ήταν εφικτό και επομένως χρησιμοποιήθηκε το flag.

Αφού γίνει το καθάρισμα, τα δεδομένα στο dataframe ομαδοποιούνται χρησιμοποιώντας το window aggregation πάνω στο column DateTime του dataframe με χρονική διάρκεια μίας ημέρας και slide ακριβώς μίας ημέρας, ώστε να δημιουργηθεί ένα tumbling window και να ομαδοποιηθούν τα δεδομένα της ίδιας ημέρας. Επιπλέον, χρησιμοποιείται watermark στα δεδομένα για να γίνει χειρισμός των ετεροχρονισμένων δεδομένων. Στη γενική περίπτωση, χρησιμοποιείται ένα πολύ μικρό χρονικό διάστημα σαν watermark (της τάξης των μερικών δευτερολέπτων) και αυτό έχει να κάνει με την προσομοίωση, αφού ετεροχρονισμένα δεδομένα έχει μόνο το W1. Για το W1 το watermark είναι 3 ημέρες, ώστε να γίνονται αποδεκτά τα ετεροχρονισμένα δεδομένα 2 ημερών ενώ να απορρίπτονται από 3 ημέρες και πάνω.

Στη συνέχεια ακολουθεί το aggregation, όπου για το απλό aggregation που ζητείται, το AggDay[x] για τους αισθητήρες δεκαπενταλέπτου, τα sum και average παρέχονται από τις έτοιμες συναρτήσεις του Apache Spark και απλώς καλούνται στο column Value του dataframe. Επίσης, γίνεται ένα round σε 2 ψηφία, καθώς υπάρχει το γνωστό πρόβλημα σε πράξεις με floating points. Το τελικό dataframe που αποστέλλεται περιέχει 2 πεδία, το ένα είναι η ημερομηνία του τέλους της επόμενης ημέρας του aggregation, η οποία λαμβάνεται από το πεδίο end του window που δημιουργείται (είναι ένα StructType) και ένα πεδίο με την τιμή του υπολογισμένου aggregation.

Για τα aggregations της διαφοράς των ημερήσιων αισθητήρων Etot, Wtot δεν παρέχεται κάποια συνάρτηση από τις βιβλιοθήκες του Apache Spark. Επομένως, ήταν ανάγκη να υλοποιηθεί μία User Defined Aggregate Function ή UDAF. Αρχικά, τα δεδομένα που καταφθάνουν ομαδοποιούνται σε windows 2 ημερών με slide μίας ημέρας (sliding window), ώστε σε κάθε window να υπάρχουν τιμές δύο διαδοχικών ημερών για να υπολογιστεί η διαφορά τους.

Υπάρχει προκαθορισμένη δομή για τη δημιουργία μίας τέτοιας συνάρτησης η οποία παρέχεται στο documentation του Spark, ενώ αντλήθηκαν περισσότερες πληροφορίες για αυτή από το google [9]. Αρχικά, η νέα κλάση που δημιουργείται για το aggregation της ημέρας (στον κώδικα αναφέρεται ως *ValueDifferenceUDAF*) πρέπει να επεκτείνει την κλάση *UserDefinedAggregateFunction*. Στη συνέχεια, πρέπει να οριστεί το inputSchema, που είναι το schema στο οποίο καλείται η συνάρτηση, εδώ ένα struct που περιγράφει το πεδίο Value στο entry του dataframe. Επιπλέον, ορίζεται το bufferSchema που είναι ένα struct για τον buffer που θα αποθηκεύει όλες τις ενδιάμεσες τιμές που χρειάζονται για να γίνει το aggregation. Εδώ, κρατείται στον buffer μόνο η τιμή της προηγούμενης ημέρας του Etot/Wtot.

Οι μέθοδοι που υλοποιούνται για το UDAF είναι οι εξής:

- Μέθοδος initialize: αρχικοποίηση του buffer σε 0.
- Μέθοδος update: υπολογισμός της νέας τιμής του buffer όταν έρθει ένα νέο δεδομένο του αισθητήρα. Εδώ υπολογίζεται η διαφορά. Κανονικά θα έπρεπε να αποθηκευτεί και η νέα τιμή που καταφθάνει στον buffer για τον υπολογισμό των επόμενων διαφορών, ωστόσο λόγω της ομαδοποίησης που έχει γίνει με το window, πάντα θα υπάρχουν μόνο 2 τιμές ανά window για να βρεθεί η διαφορά, επομένως χρειάζεται να υπολογιστεί μόνο μία διαφορά συνολικά ανά window.
- Μέθοδος merge: το Spark μπορεί να μοιράσει το aggregation σε διαφορετικά threads, επομένως πρέπει να οριστεί πως θα ενοποιηθεί το αποτέλεσμα από 2 tasks του ίδιου aggregation. Εδώ, για να λειτουργήσει το aggregation απλώς υπολογίζεται η διαφορά της τιμής των 2 buffer και αυτό είναι το πραγματικό σημείο που υπολογίζεται η διαφορά!
- Μέθοδος evaluate: τι επιστρέφει το aggregation, εδώ επιστρέφει την τιμή του buffer.

Όσον αφορά την UDAF, κάθε window περιέχει 2 τιμές Value, την τωρινή και την προηγούμενη ημέρα. Το aggregation επομένως σπάει αυτές τις 2 τιμές σε 2 tasks. Τα tasks κάνουν το update με την αρχικοποιημένη τιμή 0 του buffer, επομένως η διαφορά που υπολογίζουν είναι η ίδια η τιμή. Γι' αυτό στο merge αφαιρούνται οι 2 buffer και εκεί υπολογίζεται η πραγματική διαφορά των 2 ημερών. Δεν είναι μία υλοποίηση που θα υπολόγιζε ένα κανονικό difference aggregation, αλλά δουλεύει στην προκειμένη περίπτωση λόγω της έξυπνης ομαδοποίησης των δεδομένων σε 2ήμερα με μονοήμερο slide.

Ειδικά για το aggregation της διαφοράς, επειδή γίνεται η ομαδοποίηση σε 2ήμερα, το dataframe που προκύπτει έχει και ένα entry για την διαφορά του επόμενου διημέρου, χωρίς ακόμα να έχει καταφθάσει η επόμενη ημέρα που χρειάζεται για να βρεθεί η σωστή διαφορά. Επομένως, η διαφορά του επόμενους διημέρου πρέπει να φιλτράρεται. Σε ένα πραγματικό σύστημα αυτό θα γινόταν απλώς με τον έλεγχο του current timestamp, ωστόσο αυτό δεν είναι εφικτό στην προσομοίωση και έτσι έγινε ένα φιλτράρισμα πάνω στην τιμή που υπολογίζει η UDAF. Αν αυτή είναι μικρότερη του 150, τότε αποστέλλεται, αλλιώς σημαίνει πως είναι απλώς η τιμή του ημερήσιου αισθητήρα της τωρινής ημέρας και είναι το entry του επόμενου διημέρου που δεν έχει υπολογιστεί πλήρως ακόμη, γι' αυτό και δεν αποστέλλεται.

Στη συνέχεια, για την αποστολή των δεδομένων στη βάση, δεν υπήρχε η δυνατότητα χρήσης ενός έτοιμου connector με MongoDB, αφού αυτός απαιτεί νεότερη έκδοση του Apache Spark (> 3.0.0). Επομένως, αποφασίστηκε να προστεθεί ουσιαστικά ένα στρώμα ενός Kafka broker που θα λαμβάνει τα aggregated δεδομένα και θα τα γράφει στη βάση με χρήση sink connectors. Έτσι, χρησιμοποιήθηκε για το writeStream το package για σύνδεση σε Kafka. Συγκεκριμένα στο writeStream επισημαίνονται το topic που θα γραφτεί το aggregation, η διεύθυνση που τρέχει ο Kafka broker και το path για τα checkpoints του query. Για την αποστολή των δεδομένων σε json μορφή στο Kafka χρησιμοποιήθηκε το outputMode update για τον αισθητήρα W1, για τα late rejected events και για τα aggregations διαφοράς, ενώ για όλα τα

υπόλοιπα χρησιμοποιήθηκε το append. Το append γράφει το ολοκληρωμένο aggregation όταν τελειώσει η συγκεκριμένα ημέρα, ενώ το update υπολογίζει και γράφει όποιο entry έχει ανανεωμένη τιμή. Δεν ήταν δυνατό να χρησιμοποιηθεί το append στο aggregation του W1, καθώς θα έπρεπε το query να περιμένει να περάσουν 2 ημέρες από την τωρινή ημερομηνία για να αποστείλει το aggregation, ώστε να μην χάσει κάποιο ετεροχρονισμένο δεδομένο μέχρι και 2 ημερών, πράγμα που θα χαλούσε τη live streaming μέτρηση. Γι' αυτό χρησιμοποιήθηκε το update mode, το οποίο ωστόσο στέλνει και ενδιάμεσες τιμές του aggregation για μία ημέρα.

Κάθε ένα από τα αρχεία Water.scala, Energy.scala, Temp.scala υλοποιούν την παραπάνω διαδικασία για τα αντίστοιχα topics/μετρήσεις με πολλαπλά queries.

Τέλος, για το aggregation του leak του νερού που υλοποιήθηκε στο αρχείο Leak.scala χρησιμοποιήθηκε ελαφρώς διαφορετική προσέγγιση. Εδώ, λαμβάνονται τα ήδη υπολογισμένα aggregations της ημερήσιας διαφοράς του Wtot και του αθροίσματος του W1 από τα Kafka topics στα οποία έχουν γραφτεί ήδη από Water.scala, γίνεται το ανάλογο καθάρισμα και στη συνέχεια γίνεται ένα outer join σε δεδομένα της ίδιας ημέρας. Για να γίνει αυτό, πρέπει οπωσδήποτε τα dataframes να έχουν watermark, επομένως προστέθηκε τυπικά ένα watermark 5 δευτερολέπτων. Το join γίνεται πάνω στο DateTime πεδίο και μετά απλώς υπολογίζεται η αφαίρεση των 2 values, γίνεται ένα round σε 2 ψηφία και το dataframe τελικά αποστέλλεται και αυτό σε διαφορετικό topic στον Kafka broker. Η ίδια διαδικασία θα ακολουθούνταν και για το leak aggregation της ενέργειας, ωστόσο δεν υλοποιήθηκε.

Δ. Live Streaming – Database Gateway – Confluent Kafka:

Για να γίνει η αποθήκευση των δεδομένων στη MongoDB, όπως προαναφέρθηκε, χρησιμοποιήθηκε ένας Kafka Broker ώστε να μεταφερθούν τα δεδομένα πρώτα σε αυτόν και μετά με τη χρήση των MongoDB Sink Connectors να περαστούν στη βάση.

Αρχικά, για να σεταριστεί το Kafka σε windows μέσω του Confluent, αφού αυτό παρέχει τους sink connectors, απαιτείται περιβάλλον WSL2. Ακολουθήθηκαν τα βήματα για την εγκατάστασή του [10] και μετά έγινε η εγκατάσταση του Confluent [11]. Στην πόρτα 9021 τρέχει το Control Center του Confluent, το οποίο είναι ένα γραφικό interface για το σετάρισμα του Kafka broker και των connectors.

Ωστόσο, για την επικοινωνία του Spark με τον Kafka broker του Confluent που τρέχει σε WSL2 χρειάζεται ένα mapping στις πόρτες του localhost του WSL2 και του host μηχανήματος που τρέχει Windows 10. Γι' αυτό χρησιμοποιήθηκε το powershell script που βρέθηκε σε github repo με κάποιες αλλαγές για το mapping. Αυτό το script είναι το wsl2-port-mapper.ps1 που βρίσκεται στο github repo της εργασίας. Έτσι, το Spark μπορεί πλέον να «δει» την πόρτα 9092 που τρέχει το Kafka στο WSL2 [12].

Με τη χρήση του γραφικού interface δημιουργήθηκαν τα ανάλογα topics στον Kafka broker για να λαμβάνονται τα aggregated δεδομένα από το Spark. Συγκεκριμένα δημιουργήθηκαν τα topics: agg_etot, agg_wtot, aggday_hvac1, aggday_hvac2, aggday_miac1, aggday_miac2, aggday_mov, aggday_th1, aggday_th2, aggday_w1, late_rejected και leak_water.

Επομένως, το Spark στέλνει τα aggregated δεδομένα στο αντίστοιχο topic. Ξανά, αυτή ίσως δεν είναι η καλύτερη πρακτική, καθώς μια ομαδοποίηση στα topics θα βελτίωνε το σύστημα, ωστόσο τα topics είναι λίγα και δεν αντιμετωπίζεται κάποιο πρόβλημα στο περιβάλλον προσομοίωσης.

Για να επιτευχθεί η σύνδεση με τη βάση και η μεταφορά των δεδομένων σε αυτή, αρχικά έγινε η εγκατάσταση του MongoDB Connector (Source and Sink) με την εντολή *confluent-hub install mongodb/kafka-connect-mongodb:1.9.1* και στη συνέχεια από το tab Connect του UI που παρέχει το Confluent δημιουργήθηκε ένας connector για κάθε topic που προαναφέρθηκε. Συγκεκριμένα, στο πεδίο topics ορίζεται το topic στο οποίο θα ακούει ο connector και θα μεταφέρει νέα δεδομένα από αυτό προς τη βάση, στο πεδίο Value converter class χρησιμοποιείται η κλάση

org.apache.kafka.connect.storage.StringConverter

η οποία μετατρέπει το μήνυμα που φτάνει στο topic σε κανονική json μορφή, στο πεδίο MongoDB Connection URI προστέθηκε το URI για το connection με την MongoDB που βρίσκεται στο Atlas, στο πεδίο MongoDB database name σημειώνεται το όνομα της βάσης που θα χρησιμοποιηθεί και στο πεδίο collection name το όνομα του collection που θα δημιουργηθεί και αφού τα δεδομένα θα αποθηκευτούν σε timeseries μορφή, στο πεδίο "The field used for time" τίθεται το όνομα του πεδίου του μηνύματος που φτάνει στο Kafka που περιέχει την ημερομήνια (timestamp) και τέλος τίθεται το πεδίο "Convert the field to a BSON datetime type" σε true, για να γίνει η μετατροπή σε σωστό type για τη βάση [13]. Τα .properties αρχεία για τους connectors βρίσκονται στον φάκελο Confluent Kafka του github repo.

Για τα μη επεξεργασμένα (raw) δεδομένα δημιουργήθηκαν 3 topics, ένα για την ενέργεια, ένα για το νερό, ένα για την θερμοκρασία/κίνηση. Τα δεδομένα φτάνουν στα topics μέσω του MQTT-Kafka bridge που υλοποιήθηκε (αρχείο mqtt_kafka_bridge.py στον φάκελο Bridge του repo) και με αντίστοιχους connectors περνιούνται σε ξεχωριστή βάση με όνομα Raw.

Για την γέφυρα μεταξύ MQTT και Kafka το script απλώς δημιουργεί clients που κάνουν subscribe στα topics του Mosquitto και με τη χρήση του on_message callback, όταν έρθει ένα νέο δεδομένο το κάνουν publish στο αντίστοιχο Kafka topic με τη χρήση ενός Kafka Client. Ωστόσο, σε αυτό το σημείο τα δεδομένα αποστέλλονται σε JSON μορφή και επίσης προστίθεται και το αναγνωριστικό της κάθε μέτρησης για να είναι το payload συμβατό με τον connector και τη βάση [14].

E. Data Storage Layer

Για την αποθήκευση των δεδομένων χρησιμοποιήθηκε το MongoDB, ωστόσο για μεγαλύτερη ευκολία και ευελιξία, καθώς και κλιμακωσιμότητα του συστήματος, η βάση δεν υλοποιήθηκε τοπικά στο μηχάνημα, αλλά έγινε χρήση της δωρεάν έκδοσης του MongoDB Atlas, που παρέχει τη βάση στο Cloud [15].

Για αρχή, δημιουργήθηκε λογαριασμός στο Atlas και δημιουργήθηκε ένα απλό cluster. Για τη σύνδεση στο Atlas

έγινε ένα μικρό configuration με βάση την αποδοχή της δικτυακής κίνησης από οποιαδήποτε διεύθυνση IP, δηλαδή προστέθηκε ένας κανόνας allow στο 0.0.0.0/0. Αυτό, σε ένα πραγματικό σύστημα μπορεί να δημιουργήσει κενό ασφάλειας, ωστόσο σε περιβάλλον προσομοίωσης δεν είναι προτεραιότητα. Το URI για την πρόσβαση από τους connectors αλλά και αργότερα από το Grafana απαρτίζεται και από τα username, password του λογαριασμού που δημιουργήθηκε καθώς και από το namespace της τοποθεσίας της βάσης. Με χρήση του MongoDBCompass που παρέχει ένα φιλικό γραφικό interface γίνεται μια γρήγορη απεικόνιση της βάσης τοπικά.

Για την αποθήκευση των δεδομένων, δημιουργήθηκαν 4 διαφορετικές βάσεις, μία Energy για τα aggregations της ενέργειας, μία Water για τα aggregations του νερού, μία Temp για τα aggregations της θερμοκρασίας και μία Raw για τα μη επεξεργασμένα δεδομένα. Επίσης, το κάθε aggregation γράφεται σε ξεχωριστό, δικό του collection. Σε ένα πραγματικό σύστημα, θα ήταν καλύτερη πρακτική να γίνει μια ομαδοποίηση στα collections.

Να σημειωθεί, πως λόγω της επιλογής του update outputMode στο aggregation του W1 στο Spark, το collection του μπορεί να περιέχει πολλά entries για την ίδια ημερομηνία, καθώς μπορεί να περιέχει και ενδιάμεσες τιμές στην ημέρα, ενώ δεν υπάρχει δυνατότητα για update στο ίδιο id ενός entry σε timeseries collections. Ωστόσο, αυτό δεν είναι απαραίτητα κακό και μπορεί να χειριστεί αναλόγως και στο στρώμα απεικόνισης με το σωστό query στη βάση [16].

ΣΤ. Presentation Layer

Για το στρώμα απεικόνισης δεδομένων χρησιμοποιήθηκε το εργαλείο ανοιχτού κώδικα Grafana. Αντίστοιχα με τη βάση, για το Grafana χρησιμοποιήθηκε η Cloud έκδοση που παρέχεται.

Για την απεικόνιση των δεδομένων από τη βάση χρειάζεται ένα plugin το οποίο παρέχεται από τα enterprise plugins του Grafana, το MongoDB Datasource Plugin. Ωστόσο, το Grafana παρέχει 14ήμερη δωρεάν πρόσβαση στην enterprise έκδοση και επομένως στα πλαίσια της εργασίας ήταν η κατάλληλη επιλογή. Αρχικά, αφού εγκαταστάθηκε το plugin μέσω του γραφικού interface του Grafana, δημιουργήθηκε ένα DataSource, όπου το μόνο που χρειάστηκε από configurations ήταν να οριστεί το connection URI της βάσης στο Atlas.

Δημιουργήθηκε ένα Dashboard με ονομασία IoT Sensor Application και σε αυτό δημιουργήθηκαν 6 διαφορετικά panels για απεικόνιση των δεδομένων στο διάστημα μεταξύ των ημερών 01-01-2023 και 06-01-2023:

- Ένα panel όπου απεικονίζει ένα bar chart για την μέση τιμή του αισθητήρα TH2 ανά ημέρα, το query που χρησιμοποιήθηκε είναι: *Temp.Aggregated_TH2.find({})* για να επιστρέψει όλα τα δεδομένα και επιλέχθηκε να απεικονιστεί.
- Ένα panel με bar chart για τα μη επεξεργασμένα δεδομένα του αισθητήρα TH2, το query που χρησιμοποιήθηκε είναι το: *Raw.Temp.find({Sensor: "TH2"})*.
- Ένα panel με το table των ετεροχρονισμένων δεδομένων που απορρίφθηκαν (10 ημέρες), το query που χρησιμοποιήθηκε είναι το: *Water.Late_Events_Rejected.find({})*.
- Ένα panel με το table για τα μη επεξεργασμένα δεδομένα του αισθητήρα W1, το query που χρησιμοποιήθηκε είναι το *Raw.Water.find({Sensor: "W1"})*.
- Ένα panel με timeseries απεικόνιση για την ημερήσια διαφορά του Etot (Etot aggregation), το query που χρησιμοποιήθηκε είναι το: Energy.Aggregated_Etot.find({}).
- Ένα panel με table για την ημερήσια διαρροή του νερού, το query που χρησιμοποιήθηκε είναι το Water.Leak.find({}).

IV. Περιγραφη των αποτελεσματων και των συμπερασματων της εργασιας

Το πρωτότυπο σύστημα που υλοποιήθηκε, σε γενικές γραμμές, αντικατοπτρίζει επιτυχημένα ένα πραγματικό σύστημα IoT ενός οικιακού περιβάλλοντος. Με τις επιλογές και τα εργαλεία που αξιοποιήθηκαν δημιουργήθηκε ένα σύστημα, το οποίο είναι κλιμακώσιμο, σχετικά γρήγορο και πολυεπίπεδο.

Με τη χρήση διαφορετικών brokers επιτυγχάνεται κλιμακωσιμότητα, με τη χρήση των διαφορετικών υπηρεσιών που προσφέρει το πρωτόκολλο MQTT εξασφαλίζεται η ασφαλής και εγγυημένη μετάδοση, καθώς και η ταχύτητα για την ζωντανή μετάδοση και επεξεργασία των δεδομένων, ενώ με το μοντέλο publish/subscribe τα μηνύματα διαδίδονται εύκολα και γρήγορα στα επόμενα στρώματα του συστήματος. Ωστόσο, ένα συμπέρασμα εδώ θα ήταν ότι είναι καλύτερη πρακτική να ομαδοποιηθούν κάποια δεδομένα σε λιγότερα topics και να μοιράζονται στους clients από το ίδιο topic.

Το στρώμα ζωντανής επεξεργασίας είναι το bottleneck του συστήματος, καθώς είναι η πιο χρονοβόρα διαδικασία όπως φαίνεται. Επίσης, το Apache Spark δεν είναι ακόμα εντελώς συμβατό με μηνύματα τύπου MQTT και επομένως δημιουργούνται μερικά προβλήματα συμβατότητας με το πρωτόκολλο αλλά και με την ταχύτητά του. Τα αποτελέσματα του συγκεκριμένου στρώματος ήταν πως ειδικά σε τοπικό υπολογιστή η επεξεργασία πολλών δεδομένων μαζί είναι βαριά διαδικασία, καθώς όταν τρέχουν όλα τα queries ταυτόχρονα τα aggregations χρειάζονται κατά μέσο όρο περίπου ένα λεπτό για να παραχθούν, χρόνος ο οποίος είναι σημαντικός σε περιβάλλον προσομοίωσης που τα δεδομένα καταφθάνουν πολύ πιο γρήγορα. Ωστόσο, σε ένα πραγματικό σύστημα, πολλά από τα προβλήματα θα λύνονταν ευκολότερα, αφού η διαδικασία υπολογισμού θα γινόταν σε στάδια μέσα στη μέρα και το τελικό ημερήσιο aggregation θα υπολογιζόταν αμέσως στο τέλος της ημέρας χωρίς προβλήματα και ακόμα και σε περίπτωση καθυστέρησης μερικών λεπτών η live streaming φύση του συστήματος δεν θα χανόταν. Η καλύτερη υλοποίηση σε πραγματικό σύστημα θα ήταν μια clustered υλοποίηση, ώστε η δουλειά να μοιράζεται σε διαφορετικά nodes.

Επιπλέον, χρειάστηκε να προστεθεί ένα στρώμα Kafka broker, λόγω ασυμβατοτήτων των εκδόσεων του Spark, πράγμα που δημιουργεί περισσότερη καθυστέρηση στο σύστημα και αχρείαστη πολυπλοκότητα.

Από την άλλη, η χρήση μιας timeseries βάσης, η οποία είναι κατάλληλη για IoT δεδομένα προσθέτει μεγάλη ταχύτητα στα queries που χρειάζονται από το presentation layer, αλλά και εξοικονομεί χώρο στη βάση, πράγμα που καθιστά το σύστημα κλιμακώσιμο και μειώνει το κόστος σε πραγματικά σενάρια.

Η απεικόνιση των δεδομένων με το Grafana ήταν αρκετά ξεκάθαρη διαδικασία, καθώς σαν εργαλείο προσέφερε πολλά έτοιμα plugins και πολύ φιλικό διαδραστικό περιβάλλον για την δημιουργία γραφημάτων και πινάκων, πράγμα που σε ένα πραγματικό σύστημα θα διευκόλυνε έναν χρήστη να ελέγχει και να διαχειρίζεται τις καταναλώσεις του σπιτιού του και να βλέπει τα δεδομένα σε πραγματικό χρόνο.

Επίσης, ιδιαίτερη σημασία πρέπει να δοθεί πως τα στρώματα της βάσης και της απεικόνισης έγιναν σε cloud, όπου η διαδικασία σεταρίσματος ήταν αρκετά εύκολη και γρήγορη και δίνει μια πραγματική υπόσταση στο σύστημα, το οποίο θα ήταν υλοποιημένο σε κατανεμημένο clustered περιβάλλον.

Τελικά, το σύστημα IoT Live Streaming που υλοποιήθηκε έχει αρκετά καλές επιδόσεις, με τα ημερήσια aggregations να χρειάζονται το πολύ 1 λεπτό για να μεταφερθούν και να απεικονιστούν και τα μη επεξεργασμένα δεδομένα να φτάνουν αμέσως, ενώ το σύστημα είναι επιπρόσθετα κλιμακώσιμο και βελτιστοποιημένο, χωρίς μεγάλο κόστος για την υλοποίησή του.

V. Παραπομπες

[1] https://mqtt.org

[2] https://www.educba.com/rabbitmq-vs-mqtt/

[3] https://www.confluent.io/hub/mongodb/kafka-connect-mongodb

[4] http://www.steves-internet-guide.com/into-mqtt-python-client/

[5] http://www.steves-internet-guide.com/mqtt-which-qos-to-use/

[6] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[7] https://bahir.apache.org/docs/spark/current/spark-sql-streaming-mqtt/

[8] https://github.com/cdarlint/winutils

[9] https://medium.com/@atulverma838/udaf-user-defined-aggregate-function-in-spark-f081ef7fe404

[10] https://pureinfotech.com/install-windows-subsystem-linux-2-windows-10/

[11] https://www.confluent.io/blog/set-up-and-run-kafka-on-windows-and-wsl-2/

[12] https://gist.github.com/grishagin/e45f5a1044c87321c6f4c9cc54004ba8

[13] https://www.youtube.com/watch?v=_6NuTTQdDn4

[14] https://medium.com/python-point/mqtt-and-kafka-8e470eff606b

[15] https://www.mongodb.com/atlas/database

[16] https://www.mongodb.com/docs/manual/core/timeseries/timeseries-limitations/