Επεξεργασία ροών δεδομένων σε πραγματικό χρόνο με χρήση των open source κατανεμημένων συστημάτων

Παπανικολαού Ιωάννης
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα, Ελλάδα
el18064@mail.ntua.gr

Ανδρεάς Χρυσοβαλάντης-Κωνσταντίνος
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα, Ελλάδα
el18102@mail.ntua.gr

Μανιάτης Ανδρέας
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα, Ελλάδα
el18070@mail.ntua.gr

***Abstract*— Τα Big Data αναφέρονται στον τεράστιο όγκο δομημένων και μη δομημένων δεδομένων που οι οργανισμοί είναι πλέον σε θέση να συλλέξουν και να αναλύσουν. Το IoT, ή το Internet of Things, αναφέρεται στο διασυνδεδεμένο δίκτυο φυσικών συσκευών που μπορούν να συλλέγουν και να μεταδίδουν δεδομένα. Το σύστημα IoT αναφέρεται στις τεχνολογίες και το λογισμικό που χρησιμοποιούνται για τη διαχείριση, ανάλυση και χρήση των δεδομένων που παράγονται από συσκευές IoT. Αυτές οι τεχνολογίες και το λογισμικό είναι ζωτικής σημασίας για την αξιοποίηση των τεράστιων ποσοτήτων δεδομένων που παράγονται από συσκευές IoT. Εμπνευσμένοι από αυτά, καλούμαστε να κατασκευάσουμε ένα σύστημα επεξεργασίας ροών δεδομένων σε πραγματικό χρόνο με χρήση των open source κατανεμημένων συστημάτων, όπως RabbitMQ, Apache Flink, OpenTSDB και Grafana.**

Keywords — IoT, streaming data, rabbitmq, apache flink, opentsdb, grafana

I. ΕΙΣΑΓΩΓΗ

Η παρούσα εργασία υλοποιήθηκε στο μάθημα Ανάλυση και Σχεδιασμός Πληροφοριακών Συστημάτων, μάθημα 9ου εξαμήνου και ροής Λ. Σκοπός της εργασίας είναι η δημιουργία ενός live streaming συστήματος που θα είναι prototype ενός πραγματικού IoT συστήματος, με τη χρήση open source εργαλείων. Συγκεκριμένα, δημιουργήθηκε ένα σύστημα στο οποίο στέλνονται εικονικά δεδομένα από αισθητήρες σε πραγματικό χρόνο σε έναν Message Broker, όπου είναι ο RabbitMQ [1]. Στην συνέχεια αυτά, επεξεργάζονται μέσω του Apache Flink [2] και έπειτα αποθηκεύονται σε μια σε μία Timeseries Βάση δεδομένων, την OpenTSDB [3]. Τέλος, τα δεδομένα παρουσιάζονται σε Dashboards με την χρήση του open source εργαλείου Grafana [4] και με την χρήση websockets παρουσιάζονται live.

II. GITHUB-LINK

https://github.com/john-papani/IoT-Live-streaming

III. ΑΝΑΠΤΥΞΗ ΣΥΣΤΗΜΑΤΟΣ

A. Βασικές Λεπτομέρειες

Για την ανάπτυξη του συστήματος χρησιμοποιήθηκαν αποκλειστικά τοπικοί πόροι, μιας και υλοποιήθηκε locally. Καθώς η ζωντανή παραγωγή δεδομένων ήταν πρακτικά αδύνατη, έχει δημιουργηθεί ένα αρχείο εικονικών δεδομένων διαστήματος εφτά ημέρων *(event.json),* το όποιο θα στέλνεται όπως θα εξηγηθεί στην συνέχεια.

B. Containers – Docker

Η εργασία βασίζεται υλοποιήθηκε με τη χρήση containers που προσφέρονται μέσω του Docker [5], καθιστώντας την εφαρμογή και το αρχικό της στήσιμο γρηγορότερο. Αρχικά, κατεβάζουμε το αντίστοιχο image και έπειτα δημιουργούμε το επιθυμητό container.

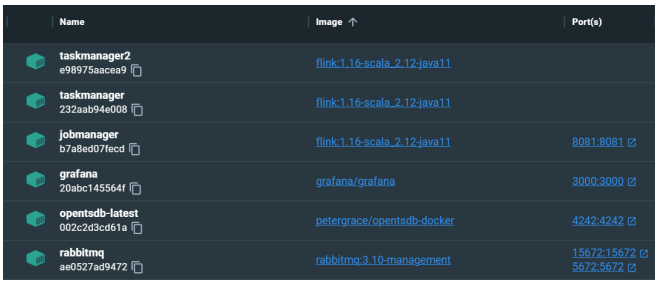
Ξεκινώντας, επιλέξαμε την έκδοση ***rabbitmq:3.10-management*** [6]**,** όπως αυτό παρέχεται από την επίσημη ιστοσελίδα του RabbitMQ.

Για το βασικό εργαλείο επεξεργασίας δεδομένων, επιλέξαμε το ***flink:1.16-scala_2.12-java11***, ακολουθώντας τις οδηγίες [7] που παρέχονται για την εγκατάσταση του container, αλλάζοντας κατάλληλα τις εντολές ώστε να κατεβάσουμε την επιθυμητή έκδοση.

Για την αποθήκευση των δεδομένων, αξιοποιήθηκε το αρχείο image **petergrace/opentsdb-docker** [8]**.** Θα πρέπει

να πειράξουμε μία παράμετρο (configuration) εσωτερικά, απαραίτητη για την αποθήκευση των δεδομένων μετέπειτα. Για αυτό το σκοπό, ανοίγουμε ένα terminal για το συγκεκριμένο container, και βρίσκουμε το αρχείο *opentsdb.conf*. Το αρχείο βρίσκεται εσωτερικά του φακέλου */usr/local/share/opentsdb/etc/opentsdb.* Ανοίγουμε το αρχείο *(vi opentsdb.conf)* και προσθέτουμε στο τέλος το εξής: ***tsd.storage.fix_duplicates = true***. Τέλος κλείνουμε και αποθηκεύουμε το αρχείο *(:wq)* και επανεκκινούμε το container ώστε να ενημερωθεί το configuration. Με την 'ενεργοποίηση' αυτού του flag, επιλύονται πιθανά προβλήματα εγγραφής διπλότυπων τιμών, μιας και έτσι θα διατηρείται το πιο πρόσφατο σημείο δεδομένων.

Τέλος, για το Grafana χρησιμοποιήθηκε το **grafana/grafana** [9], καθώς και το *opentsdb plugin* που προσφέρει για την αυτόματη και ταχεία άντληση των δεδομένων από την βάση.



Σχήμα 1. Αναμενόμενο Αποτέλεσμα στο Docker Desktop

Να αναφέρουμε ότι όταν ανοίγουμε τις τοπικές ιστοσελίδες των docker containers, *http://localhost:{port:port}*, όπου το κάθε *port* φαίνεται στο Σχήμα 1, σε ορισμένες περιπτώσεις ζητάει στοιχεία σύνδεσης. Για το *rabbitmq – {15672:15672}*, το προεπιλεγμένο username και password είναι *guest* και *guest* και για το *Grafana – {3000:3000}*, το προεπιλεγμένο username και password είναι *admin* και *admin*.

C. Τοπικές Εφαρμογές

Πέρα από τα παραπάνω docker containers, είναι απαραίτητη και η εγκατάσταση τριών ακόμα εργαλείων, που προτιμήθηκε η τοπική – 'κλασική' εγκατάσταση τους.

Το πρώτο εργαλείο είναι η **Java** και πιο συγκεκριμένα η έκδοση **11** [10]. Επιλέχθηκε αυτή η έκδοση προκειμένου να είναι συμβατή με την έκδοση του Apache Flink.

Το δεύτερο εργαλείο είναι η **Python** [11]. Προτείνουμε την έκδοση **3.8.10,** χωρίς αυτό να σημαίνει ότι κάποια μικρότερη έκδοση δεν είναι αποδεκτή, μιας και η χρήση του εργαλείου είναι σε λίγα μόνο σημεία (*Python3.4+*). Ακόμα, πρέπει να εγκατασταθεί το πακέτο *pika (pip install pika*). Το Pika είναι μια client βιβλιοθήκη RabbitMQ για Python.

Τέλος, πρέπει να εγκατασταθεί το εργαλείο **Apache Maven – 3.8.6** [12]**,** το οποίο είναι υπεύθυνο για την δημιουργία ενός περιβάλλοντος java για την υλοποίηση των αρχείων του προγράμματος Flink. Πιο αναλυτικά, μέσω μίας εντολής χτίζεται ένα java project με τα απαραίτητα αρχεία και ένα αρχείο με τίτλο "*pom.xml*". Μέσω αυτό του αρχείο γίνεται η διαδικασία της κατασκευής και εξέλιξης του project πιο εύκολη και άμεση, καθώς αν θέλουμε να προσθέσουμε ένα dependency αυτό γίνεται μέσα από λίγες γραμμές, ακολουθώντας προφανώς κάθε φορά από εντολή compile (*mvn clean install).*

D. Οδηγίες Εγκατάστασης και Λειτουργίας της Εφαρμογής

Για αρχή κατεβάζουμε όλα τα αρχεία από το GitHub repository *(git clone)*. Στη συνέχεια βεβαιωνόμαστε πως έχουμε κατεβάσει στον υπολογιστή μας, το Docker Desktop (αν υπάρχει διαθέσιμο για το λειτουργικό σύστημα του υπολογιστή μας). Έπειτα, είτε κατεβάζουμε μεμονωμένα τα docker images που αναλύθηκαν παραπάνω, είτε τρέχουμε σε ένα terminal το αρχείο *downloadAllDocker.sh,* το οποίο κατεβάζει και δημιουργεί όλα τα απαιτούμενα container για το παρών project, και τέλος επιβεβαιώνουμε ότι έχουν εγκατασταθεί όλα σωστά. Μετά, ελέγχουμε ότι έχουν εγκατασταθεί και τα τοπικά εργαλεία με τις εξής εντολές: "*python --version", "java --version", "mvn -v",* βλέποντας έτσι και τις εκδόσεις τους.

Στην συνέχεια αφού έχουμε τελειώσει με την εγκατάσταση όλων των εργαλείων, προχωράμε με το το τρέξιμο του αρχείου *installMvnFlink.sh* στο εσωτερικό του φακέλου, με το όποιο 'χτίζεται' το Maven project και εγκαθίστανται εσωτερικά όλα τα αρχεία που χρειάζονται για το πρόγραμμα.

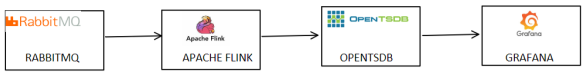
Συνολικά, τρέχουμε τις εξής εντολές:

- ***git clone https://github.com/john-papani/IoT-Live-streaming.git***
- ***cd IoT-Live-streaming***
- ***./downloadAllDocker.sh***
- ***./installMvnFlink.sh***

Στην τελική τώρα φάση, τρέχουμε σε δύο terminal ***παράλληλα*** το java αρχείο ***all_aggregation.java*** και το python αρχείο ***send.py***. Έπειτα από λίγα δευτερόλεπτα που θα έχει ολοκληρωθεί η αποστολή, η επεξεργασία και η αποθήκευση των δεδομένων, μπορούμε μέσα από το Grafana να δούμε τα διαγράμματα και τους πίνακες μας.

IV. ΠΕΡΙΓΡΑΦΗ ΣΥΣΤΗΜΑΤΟΣ – ΛΟΓΙΣΜΙΚΟ

A. Διάγραμμα Σύνδεσης Εργαλείων



Σχήμα 2. Βασικές Συνδέσεις Εργαλείων

B. Παραγωγή Δεδομένων

Επειδή όπως λογικό η κατασκευή ενός πραγματικού IoT συστήματος είναι αδύνατη και προκειμένου να δημιουργήσουμε ένα prototype απαιτείται να προσομοιάσουμε και την είσοδο δεδομένων. Για το σκοπό αυτό βασιστήκαμε στην

δημιουργία εικονικών δεδομένων για δέκα αισθητήρες *(TH1, TH2, HVAC1, HVAC2, MiAC1, MiAC2, Etot, Mov1, W1, Wtot),* με τιμές και περίοδο ανανέωσης όπως περιγράφεται στον Table 1.

Η παραγωγής δεδομένων είναι μια διαδικασία που πραγματοποιείται μία φορά στην αρχή, για την δημιουργία του αρχείου event.json, το όποιο θα περιέχει αυτά τα – τυχαία – δεδομένα. Τα δεδομένα αυτά παράχθηκαν από ένα script που φτιάχτηκε σε γλώσσα Python (*generator.py + fun.py)***.**

Αυτό το Python αρχείο βασίστηκε στις εξής συμβάσεις:

a. θα παράγει δεδομένα ανά 1 δευτερόλεπτο για κάθε αισθητήρα 15λεπτου

b. κάθε φορά θα παράγει δεδομένα για τους αισθητήρες των δεκαπενταλέπτων θα φροντίζουμε η τιμές να είναι μέσα στο επιθυμητό range.

c. Θα παράγει δεδομένα ανά 96 (24x4x1) δευτερόλεπτα για κάθε αισθητήρα ημέρας

d. κάθε φορά θα παράγει δεδομένα για τους αισθητήρες των ημερών που θα είναι αθροιστικές (να αυξάνονται με βάση το επιθυμητό range).

e. Θα παράγει 4 με 5 άσσους μέσα στην ημέρα (στα 96 δευτερόλεπτα) τυχαία.

TABLE 1. ΔΕΔΟΜΕΝΑ- ΑΙΣΘΗΤΗΡΕΣ

ΑΙΣΘΗΤΗΡΕΣ			
Όνομα	***Interval***	***Aggregation Method***	***Value Range***
TH1	15 min	Average	12-35^{oC}
TH2	15 min	Average	12-35^{oC}
HVAC1	15 min	Sum	0-100 Wh
HVAC2	15 min	Sum	0-200 Wh
MiAC1	15 min	Sum	0-150 Wh
MiAC2	15 min	Sum	0-200 Wh
Etot	1 day	Max	2600x24 ± 1000 Wh
Mov1	-	Sum/Count	-*
W1	15 min	Sum	0-1 lt
Wtot	1 day	Max	110 ± 10 lt

* να παραχθούν τουλάχιστον 4 με 5 άσοι την ημέρα τυχαία

Ετεροχρονισμένα δεδομένα

Μια πάγια τακτική, ειδικά σε πραγματικές συνθήκες *IoT* συστήματος, είναι η αποστολή ετεροχρονισμένων δεδομένων – Late Events. Κάτι τέτοιο συμβαίνει είτε για ενημέρωση λανθασμένων τιμών είτε για προσθήκη νέων σε περιπτώσεις όπου την τότε χρονίκη στιγμή υπήρχε αδυναμία λήψης της τιμής. Στο σύστημα μας λοιπόν επιλέχθηκε η παραγωγή ετεροχρονισμένων δεδομένων μόνο για τον αισθητήρα κατανάλωσης νερού W1 για τον οποίον δημιουργήσαμε ένα επιπλέον script-function που παράγει δεδομένα ανά 20 δευτερόλεπτα με timestamp που αντιστοιχεί σε *2 μέρες πίσω – Late Processed Events* – και ανά 120 δευτερόλεπτα timestamp που αντιστοιχεί σε *10 μέρες πίσω – Late Rejected Events*. Όλα τα Late Events, τα επεξεργαζόμαστε μέσω του Apache Flink, όπως θα δούμε και παρακάτω.

C. *Αποστολή Δεδομένων – RabbitMQ*

Το RabbitMQ είναι ένας message broker – ένας μεσάζων για ανταλλαγή μηνυμάτων. Δίνει στις εφαρμογές μια κοινή πλατφόρμα για την αποστολή και λήψη μηνυμάτων και στα μηνύματά ένα ασφαλές μέρος για να ζήσουν μέχρι να ληφθούν. Συγκεκριμένα, όσο αναφορά την λειτουργία του, απαιτείται να υπάρχει ένας αποστολέας, ένα κανάλι που θα διέρχονται τα μηνύματα *(queue)* και έναν παραλήπτη.

send.py

Αυτό το αρχείο υλοποιεί έναν αποστολέα – Sender, δηλαδή μια διεργασία υπεύθυνη για την αποστολή των δεδομένων από το αρχείο – event.json – στο RabbitMQ broker. Πιο συγκεκριμένα έχοντας ορίσει εσωτερικά του αρχείο send.py τις κατάλληλες παραμέτρους για τον broker δημιουργήται ένα κανάλι (channel - queue) – με όνομα *dataQueue* - στο οποίο αποστέλονται αυτούσια όλα τα δεδομένα, μέσω της συνάρτησης *basic_publish*.

Βασική παρατήρηση όσον αφορά τα δεδομένα είναι ότι οι **ημερομηνίες/ωρες** αποστέλονται σε μορφή **epoch-unix.**

Πιο συγκεκριμένα, σε θέματα κώδικα, αρχικά μέσω του python πακέτου pika, δημιουργούμε ένα connection του υπολογιστή με τον RabbitMQ server, ορίζοντας ταυτόχρονα και το queue που θα χρησιμοποιηθεί. Στην συνέχεια, αποστέλλουμε τα δεδομένα που βρίσκονται στο ήδη έτοιμο αρχείο και με την ολοκλήρωση της αποστολής τερματίζουμε την σύνδεση *(connection.close()).* Ο παραλήπτης στην παρούσα εφαρμογή δεν υφίσταται σαν μεμονωμένο python αρχείο, αλλά βρίσκεται εσωτερίκα στο Apache Flink, ως πηγή δεδομένων (data source).

D. *Επεργασία Δεδομένων*

all_aggregations.java

Το αρχείο αυτό είναι το βασικότερο όλου του συστήματος μιας και αυτό είναι υπεύθυνο τόσο για την αποστολή των raw δεδομένων, όσο και των επεξεργασμένων δεδομένων στην βάση δεδομένων μας.

Μια χρήσιμη έννοια πρώτου προχωρήσουμε στην ανάλυση της επεξεργασίας των δεδομένων είναι η χρήση *Windows.* Επειδή τα stream που καλούμαστε να διαχειριστούμε είναι απεριόριστου μεγέθους, μιας και θεωρητικά δεν ξέρουμε πότε και αν τελειώνουν ποτέ, είναι αδύνατον να εφαρμοστούν κλασικές μέθοδοι τύπου *group_by.* Αυτό το πρόβλημα επιλύει η μέθοδος των *Windows*, μιας και χωρίζει το stream σε 'buckets' πεπερασμένου μεγέθους, πάνω στα οποία μπορούμε να εφαρμόσουμε υπολογισμούς. Βέβαια πριν από αυτό, το Flink πρέπει να γνωρίζει τις χρονικές σημάνσεις συμβάντων, που σημαίνει ότι κάθε στοιχείο στο stream πρέπει να έχει τη χρονική σήμανση συμβάντος (δηλαδή ένα ξεχωριστό timestamp). Αυτό γίνεται στην περίπτωση μας με την εξαγωγή της χρονικής σφραγίδας από το πεδίο του *timeday* χρησιμοποιώντας ένα *TimestampAssigner*

(*BoundedOutOfOrdernessTimestampExtractor*). Έτσι, το Flink μπορεί εύκολα να χωρίσει το κύριο stream ανάλογα με τις παραμέτρους που του έχουμε ορίσει (*Time.days(1)),* 'κατηγοριοποιώντας' τα δεδομένα ανά ημέρα, μιας και όλα τα aggregation απαιτούν ημερήσια ομαδοποίηση.

Πιο αναλυτικά, σε ένα αρχικό Datastream *stream* διαβάζονται όλα τα δεδομένα από το queue που έχει ήδη γεμίσει από το αρχείο send.py προηγουμένως. Έπειτα μέσα ένα φιλτράρισμα κρατάμε σ' ένα νέο Datastream *rawData*, τα 'καθαρά' δεδομένα χωρίς δηλαδή τα late event όπου στα περισσότερα aggregations δεν υπάρχουν, πέρα από αυτό του W1 που το συνυπολογίζουμε. Έτσι για τον σκοπό αυτό φτιάξαμε ένα ακόμα Datastream – *streamWithTwoDaysLateEvents* – στο οποίο περιέχονται οι τιμές όχι μόνο οι τιμές του rawData stream, αλλά και τα *Late Processed Events* τα οποία χρειάζεται να τα λάβουμε υπόψη μας για τα daily aggregation του W1.

Η εξεργασία των δεδομένων λοιπόν χωρίζεται στις εξής κατηγορίες:

1) Raw data

Για τα *raw data* έχουμε δημιουργήσει *10 SingleOutputStreamOperator* (όσοι και οι αισθητήρες μέτρισης), όπου σε κάθε γράμμη κάθε stream υπάρχει ένα *Tuple* με την ημερομηνία/ωρα και την τιμή της εκάστοτε μέτρησης (π.χ θερμακρασίας). Τα δεδομένα αυτά αποθηκευονται κάθε χρονική στιγμή που παράγονται καθώς δεν επιδέχονται περαιτέρω επεξεργασία. Τα stream των raw data ξεκινάνε με τον χαρακτηριστικο "only" και κατάληξη το όνομα του αισθητήρα (π.χ *onlyTh1, onlyMiac1, onlyW1 κτλ).* Οφείλουμε να επισημάνουμε πως τα εννιά από τα δέκα νέα streams προήρθαν από το rawData stream, ενώ μόνο το *onlyW1* προήρθε από το *streamWithTwoDaysLateEvents*, για τους προφανείς λόγους.

2) Aggregated Data

Αρχικά έχουμε ορίσει ένα νέο stream, τύπου *WindowedStream* με όνομα *oneDayWindowedStream*, όπου όπως εύκολα γίνεται κατανοητό από το όνομα χωρίζονται τα δεδομένα κατάλληλα ανά ημέρα με την χρήση της μεθόδου που αναλύθηκε προηγομένως *(timeWindow – assignTimestampsAndWatermarks).* Έτσι πάλι, μέσα από 10 νέα *SingleOutputStreamOperator* υλοποιούνται τα aggregation για κάθε αισθητήρα με βάση το *oneDayWindowedStream*, εκτός από αυτό του W1 όπου βασιστηκάμε στο αρχικό stream που περιέχει και τα Late events. Έτσι σε κάθε γράμμη κάθε τέτοιου stream υπάρχει ένα *Tuple* με την ημερομηνία/ώρα και την εκάστοτε τιμή. Τα δεδομένα αυτά αποθηκεύονται μετά το τέλος κάθε ημέρας και συγκεκριμένα στη πρώτη καταγραφή της επόμενης μέρας. Τα stream των aggregation data ξεκινάνε με το είδος του aggregation και κατάληξη το όνομα του αισθητήρα (π.χ *avgTh1, sumMiac1, sumW1 κτλ).* Επισημαίνουμε ότι αυτό με βάση το πώς έχουμε κατασκευάσει τα εικονικά δεδομένα, είναι απαραίτητο να επεξεργαστούν και τα δεδομένα που παίρνουν νέα τίμη μία φορά την ημέρα. Ομοίως με πριν, το aggregation για τον αισθητήρα W1 βασίστηκε στο *streamWithTwoDaysLateEvents*.

Πέρα από τα βασικά aggregation , υπάρχουν και τα aggregation που αφορούν υπολογισμό "διαφορά" για τους αισθητήρες ενημέρωσης δεδομένων ανά ημέρα (Etot, Wtot) – *AggDayDiff[y]*. Για το σκοπό αυτών των ημερίσιων aggregations έχουμε δύο DataStreams, όπου σε κάθε γράμμη κάθε stream υπάρχει, πάλι, ένα *Tuple* με την ημερομηνία/ώρα και τη τιμή του εκάστοτε aggragetion. Αυτά τα Datastream ξεκινάνε με τον χαρακτηριστικο "diff" και κατάληξη το aggregation και το όνομα του αισθητήρα (*diffMaxEtot, diffMaxWtot).*

3) Επεξεργασία Ετεροχρονισμένων Δεδομένων

Όπως είδαμε παραπάνω τα ετεροχρονισμένα χωρίζονται σε δύο 'κατηγορίες', στα Late Rejected Events και στα Late Processed Events. Όσον αφορά τα πρώτα που παράγονται ανά δέκα ημέρες η επεξεργασία είναι η ακόλουθη. Δημιουργούμε ένα νέο Datastream, με ονομασία *tenDaysLateStream,* το οποίο βασιζόμενο στο αρχικό stream ελέγχει κάθε φορά αν η τιμή της ημερομηνίας/ώρας κάθε γραμμής (row of stream) είναι σε μεγαλύτερο διάστημα των επτά ημερών από την προηγούμενη της. Αν ισχύει αυτή η συνθήκη, τότε προσθέτουμε αυτό το tuple στο νέο stream, αποτελούμενο από την unix ημερομηνία και την τιμή του αισθητήρα W1. Τέλος, για λόγους πληρότητας, δημιουργήσαμε και ένα Datastream για τα Late Processed Events, με όνομα *twoDaysLateStream,* ακολουθώντας την ίδια διαδικασία με προηγουμένως με την μόνη διαφορά ότι η σύγκριση αφορά αποκλειστικά δύο ημέρες.

eachrow.java

Προηγουμένως σκόπιμα παραλείψαμε να αναφέρουμε πως κατά την διάρκεια κατανάλωσης των δεδομένων, από το queue προς την Apache Flink, χρειάζεται όταν ορίζουμε την πηγή (addSource) να ορίσουμε και ένα deserialization σχήμα (DeserializationSchema). Το deserialization σχήμα περιγράφει τον τρόπο μετατροπής των μηνυμάτων byte που παραδίδονται από την πηγή δεδομένων σε τύπους δεδομένων (αντικείμενα Java) που υποβάλλονται σε επεξεργασία από το Flink. Έτσι εμείς προτιμήσαμε να φτιάξουμε ένα (new) custom DeserializationSchema, το οποίο μετατρέπει τα bytes εισόδου με βάση τις μεταβλητές που έχουμε ορίσει στο eachrow.java αρχείο. Προφανώς αυτές οι μεταβλητές συμπίπτουν με αυτές που υπάρχουν στο αρχείο event.json, με τα αρχικά δεδομένα. Αυτές οι μεταβλητές είναι οι εξής: (int) th1, (int) th2, (int) hvac1, (int) hvac2, (int) miac1, (int) miac2, (int) etot, (int) mov1, (double) w1, (int) wtot.

E. Αποθήκευση Δεδομένων στην Βαση

Η αποστολή και η αποθήκευση των δεδομένων στην

Timeseries Βάση δεδομένων, την OpenTSDB γίνεται εντός του αρχείο *all_aggregations.java*, μόλις ολοκληρώνεται η επεξεργασία των δεδομένων. Πιο αναλυτικά για τον σκοπό αυτό έχουν δημιουργηθεί οκτώ (8) λίστες, τέσσερις (4) που περιέχουν τα stream των raw, όλων των ειδών aggregated δεδομένων και τα Lates Events(*rawDataList*, *aggList aggDayList* και *lateEventList* αντίστοιχα) και τέσσερις (4) που περιέχουν ονομασίες αισθητήρων που θα χρησιμοποιηθούν ως χαρακτηριστικό στην βάση δεδομένων (*sensors*, *aggSensors* ,*aggDaySensors, lateEventSensors*). Έπειτα, μέσα από τέσσερα for-loop συνδυάζονται ανά δύο οι λίστες και αποστέλλονται τα επεξεργασμένα και μη δεδομένα, με την χρήση της συνάρτησης *writeToOpenTSDB().* Η συνάρτηση αυτή αντλώντας τα στοιχεία που δέχεται ως ορίσματα και δημιουργώντας το κατάλληλο μήνυμα (String *msg)* στέλνει μέσω API ένα *Post Request,* με την χρήση της κλάσης *RestClient* και αποθηκεύει τα δεδομένα στην βάση.

RestClient.java

Μέσα σε αυτήν την κλάση, η συνάρτηση *publishToOpenTSDB()* διαχειρίζεται τις παραμέτρους, *(protocol, host, port, path, message)* και εκτελεί το HTTP Post Request επιστρέφοντας την απάντηση στην μεταβλητή *res.* Αν η απάντηση επιστρέφει τιμή 200-299 (στην περίπτωση μας θα πρέπει να επιστρέψει 204) τότε η είναι αποδεκτό, αν επιστρέψει τιμές μεγαλύτερες από 300, τότε η αποστολή των δεδομένων δεν ήταν επιτυχές.

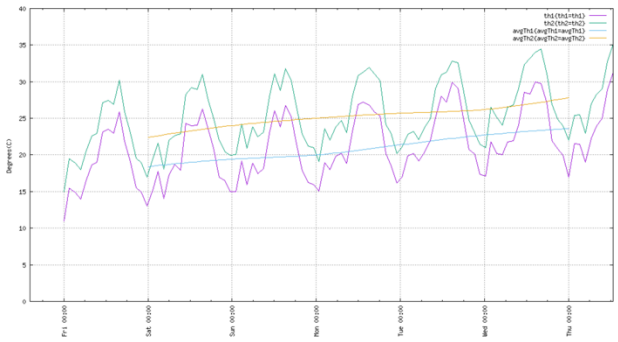
F. Λίγα λογια για την βάση δεδομένων – OpenTSDB

OpenTSDB είναι ένα από τα πιο διαδεδομένα συστήματα αποθήκευσης δεδομένων σε στιγμιότυπα (time-series), μιας και λειτουργεί με την αποθήκευση δεδομένων σε στιγμιότυπα σε μια βάση δεδομένων HBase. Η Apache HBase είναι μία βάση δεδομένων Hadoop, ένας κατανεμημένος, επεκτάσιμος, μεγάλος χώρος αποθήκευσης δεδομένων. Πιο συγκεκριμένα και όσον αφορά την αποθήκευση μέσω *API Post,* το μήνυμα που θα σταλθεί πρέπει να έχει συγκεκριμένη σύνταξη, όπως παρουσιάζεται παρακάτω:

```
{
  "metric": "th1"
  "timestamp": 1731628800000,
  "value": 18,
  "tags": {
    "th1": "th1"
  }
}
```

Όπως φαίνεται, έχει τέσσερα βασικά πεδία: metrics, timestamp, value, tags. Για τους σκοπούς της παρούσας εργασίας, στο πεδίο metrics εμφανίζεται το όνομα του sensor, όπως και στα δύο πεδία του tag. Στο timestamp τοποθετείται η Unix ημερομηνία και στο value η τιμή της θερμοκρασίας. Έτσι είναι εύκολο να εντοπίσουμε τα διάφορα raw/aggregated stream data που έχουν δημιουργηθεί μέσω Flink, αφού κάθε ένα αποθηκεύεται με ένα χαρακτηριστικό όνομα – μητρική *(metric)* στην βάση.

Τέλος, μία πρώτη ματιά των δεδομένων εμφανίζεται στην ιστοσελίδα του OpenTSDB (*localhost:4242),* επιλέγοντας κάθε φορά μόνο το χρονικό διάστημα και το όνομα της μετρικής.



Σχήμα 3. OpenTSDB

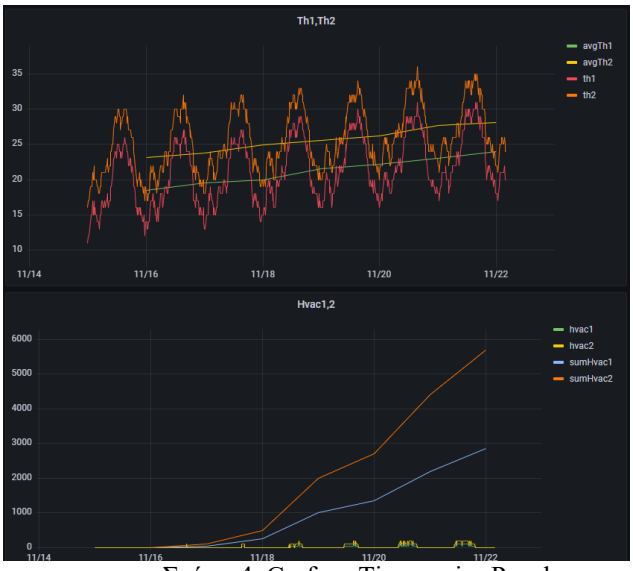
G. Παρουσίαση Δεδομένων – Grafana

Το Grafana είναι μια open-source διαδραστική πλατφόρμα οπτικοποίησης δεδομένων, που αναπτύχθηκε από την Grafana Labs, η οποία επιτρέπει στους χρήστες να βλέπουν τα δεδομένα τους μέσω γραφημάτων, πινάκων και άλλου είδους διαγραμμάτων.

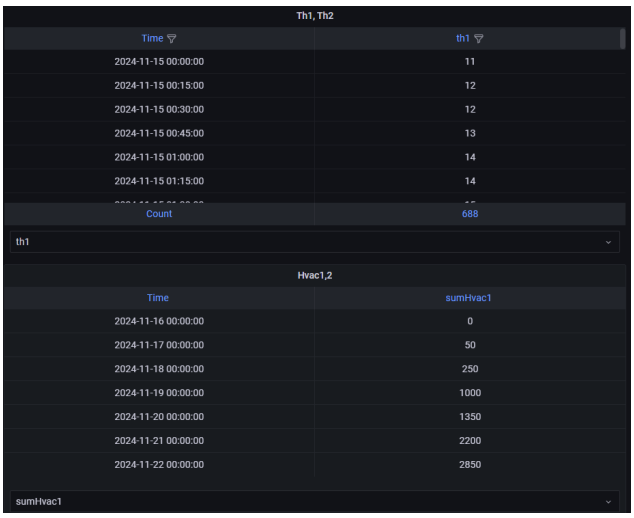
Πιο αναλυτικά, από την ιστοσελίδα του Grafana (*localhost:3000)* επιλέγουμε αρχικά το plugin που μας επιτρέπει την απευθείας και άμεση επικοινωνία της εφαρμογής με την βάση δεδομένων - *OpenTSDB Data Source - Native Plugin.*[14] Στην συνέχεια, το παραμετροποιούμε κατάλληλα, ορίζοντας το *URL = http://host.docker.internal:4242* και αφήνοντας τις υπόλοιπες επιλογές όπως έχουν. Τέλος, φτιάχνουμε ένα νέο Dashboard, προσθέτοντας τα επιθυμητά διαγράμματα και πίνακες, ορίζοντας προφανώς ως DataSource σε κάθε περίπτωση αυτό που ορίσαμε προηγουμένως.

Συνολικά υλοποιήθηκαν 16 panels στο Grafana απεικονίζοντας διαφορετικές μετρικές για όλους τους αισθητήρες δεδομένων, καθώς και των διάφορων aggregations. Αξιοποιήθηκαν Time-series panels [Σχήμα 4], Table panels [Σχήμα 5], Stat [Σχήμα 6], Gauge [Σχήμα 7] και Bar Gauge.

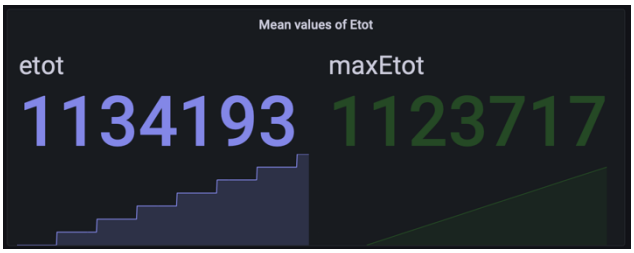
Ενδεικτικές φωτογραφίες των παραπάνω panels που υλοποιήθηκαν στο Dashboard του Grafana παρατίθενται παρακάτω.



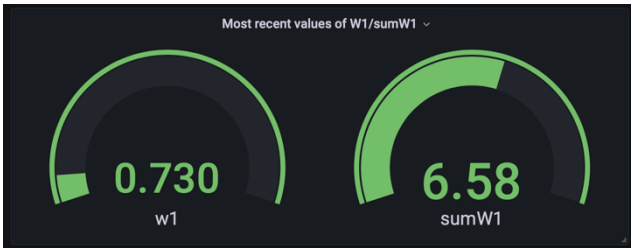
Σχήμα 4. Grafana Time-series Panels



Σχήμα 5. Grafana Table Panels



Σχήμα 6. Grafana Stat Panel for Etot & maxEtot



Σχήμα 7. Grafana Gauge Panel

Ολόκληρο το Dashboard μπορεί να βρεθεί στο αρχείο *IoT-Dashboard.json*. Για να προβληθεί το Dashboard στο Grafana, αρκεί να μέσω του *menu* να επιλέξουμε + *Import* και έπειτα *Upload JSON*.

V. ΑΠΟΤΕΛΕΣΜΑΤΑ – ΣΥΜΠΕΡΑΣΜΑΤΑ

Όπως είναι λογικό, έχοντας ολοκληρώσει όλα τα προηγούμενα έχουμε καταφέρει να υλοποιήσουμε πλήρως μία πιστή αντιγραφή ενός IoT συστήματος αποτελούμενο από δέκα αισθητήρες. Προφανώς, η υλοποίηση ενός πραγματικού IoT συστήματος είναι κάτι πιο περίπλοκο, τόσο σε θέμα ποσότητας δεδομένων μιας και ένα πραγματικό σύστημα έχει πολλαπλάσιο αριθμό αισθητήρων, όσο και σε θέμα υπολογιστικών πόρων για την επεξεργασία των μεγαλύτερων αρχείων δεδομένων. Βέβαια, με την παραγωγή των Late Events καταφέραμε να προσεγγίσουμε τις πραγματικές συνθήκες, μιας και καθημερινά στέλνονται Late event data προκειμένου να διορθωθούν παλιές τιμές.

Πιο αναλυτικά, καταφέραμε να οπτικοποιήσουμε τα σειριακά δεδομένα εισόδου τόσο σε διαγράμματα όσο και σε πίνακες τόσο για τα raw data, όσο και για τα aggregated. Μπορούμε, λοιπόν, να παρατηρήσουμε ποιά ήταν η μέση τιμή της θερμοκρασίας για μία ολόκληρη μέρα ή ποιο είναι το άθροισμα κατανάλωσης νερού με βάση έναν αισθητήρα. Επιπλέον, λόγω των δυνατοτήτων που προσφέρει το εργαλείο Grafana καθίσταται δυνατό (με την χρήση πραγματικών δεδομένων μεγάλης διάρκειας) να επεκτείνουμε τις παραπάνω λειτουργίες. Ενδεικτικά μπορούμε μέσω της συνδυαστικής παρατήρησης μετρήσεων να προβλέψουμε μελλοντικές ανάγκες του «έξυπνου» σπιτιού. Ένα τέτοιο παράδειγμα είναι η παρατήρηση της κατανάλωσης ενέργειας, συναρτήσει της μεταβολής της θερμοκρασίας, μέσω της οποίας μπορούμε να προβλέψουμε με μεγαλύτερη ακρίβεια τις ενεργειακές ανάγκες του σπιτιού μέσα στον χρόνο.

Ένα γενικότερο σχόλιο όσο αναφορά τα aggregations είναι ότι φαίνεται να έχουν υπολογιστεί σωστά μιας και τα διαγράμματα το επιβεβαιώνουν. Χαρακτηριστικό παράδειγμα είναι για τους αισθητήρες θερμοκρασίας (th1, th2), όπου το η ευθεία του average φαίνεται να βρίσκεται στην αναμενόμενη – σωστή θέση.

REFERENCES

[1] RabbitMQ is the most widely deployed open source message broker. https://www.rabbitmq.com

[2] Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. https://nightlies.apache.org/flink/flink-docs-release-1.16/

[3] OpenTSDB – The Scalable Time Series Database. http://opentsdb.net/

[4] Grafana is the open source analytics & monitoring solution for every database. https://grafana.com

[5] Docker is a platform designed to help developers build, share, and run modern applications. We handle the tedious setup, so you can focus on the code. https://www.docker.com/

[6] RabbitMQ - Docker Image https://hub.docker.com/_/rabbitmq

[7] Apach Flink - Docker Setup. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/deployment/resource-providers/standalone/docker/

[8] OpenTSDB – Docker Image. https://hub.docker.com/r/petergrace/opentsdb-docker/

[9] Grafana – Run Grafana Docker image. https://grafana.com/docs/grafana/latest/setup-grafana/installation/docker/

[10] Python is a programming language that lets you work quickly and integrate systems more effectively. https://www.python.org/

[11] Java Intallation . https://www.oracle.com/java/technologies/downloads/#java11

[12] Apache Maven is a software project management and comprehension tool. https://maven.apache.org/

[13] OpenTSDB provides an HTTP based application programming interface to enable integration with external systems. http://opentsdb.net/docs/build/html/api_http/index.html

[14] OpenTSDB plugin for Grafana. https://grafana.com/grafana/plugins/opentsdb/