Επεξεργασία Ροών Δεδομένων Πραγματικού Χρόνου

Κωνσταντίνος Παπαϊωάννου
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα, Ελλάδα
el17005@mail.ntua.gr

Παναγιώτης Ζευγολατάκος
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο
Αθήνα, Ελλάδα
el17804@mail.ntua.gr

***Abstract*—Στην εποχή των μεγάλων δεδομένων και του ολοένα και αναπτυσσόμενου διαδικτύου των πραγμάτων αναγκαία είναι η ανάπτυξη συστημάτων επεξεργασίας μεγάλου όγκου δεδομένων και παρουσίασής τους με ουσιαστικό τρόπο σε πραγματικό χρόνο. Στόχος είναι τα δεδομένα αυτά να προσφέρουν χρήσιμη πληροφορία για την υποστήριξη αποφάσεων και όχι μονο. Για αυτό το λόγο σε αυτήν την εργασία θα παρουσιαστεί ένα σύστημα επεξεργασίας ροών δεδομένων σε πραγματικό χρόνο με χρήση open source κατανεμημένων συστημάτων όπως RabbitMQ, Apache Spark Streaming, MongoDB και Grafana.**

***Index Terms*—big data, IoT, data-driven decision making, streaming data, rabbitmq, spark streaming, mongodb, grafana**

I. Εισαγωγή

Η παρούσα εργασία [1] εκπονήθηκε στα πλαίσια του μαθήματος "Ανάλυση και Σχεδιασμός Πληροφοριακών Συστημάτων" 9ου εξαμήνου. Σκοπός της εργασίας ήταν η προσομοίωση ενός live streaming συστήματος, το οποίο θα αποτελεί το πρωτότυπο ενός πραγματικού IoT συστήματος. Ουσιαστικά, δημιουργήθηκε ένα σύστημα που παράγει και επεξεργάζεται ροές δεδομένων σε πραγματικό χρόνο χρησιμοποιώντας επιλεγμένα open source εργαλεία. Αρχικά, παράγονται δεδομένα από εικονικούς αισθητήρες σε πραγματικό χρόνο, τα οποία στέλνονται σε έναν MQTT broker, το RabbitMQ [1]. Στη συνέχεια τα δεδομένα καταναλώνονται από το Spark Streaming [2] και μετά από την επεξεργασία τους αποθηκεύονται σε μία NoSQL βάση δεδομένων, τη MongoDB [3]. Για την αποθήκευσή τους γίνεται χρήση plugin για Timeseries δεδομένα [4] που στοχεύει στην αποδοτικότερη χρήση του αποθηκευτικού χώρου. Τέλος, τα αποθηκευμένα δεδομένα παρουσιάζονται σε dashboards με τη χρήση του εργαλείου Grafana [5].

[1] Rhea System [Source Code]

II. Ανάπτυξη Συστήματος

A. Σε επίπεδο VM

Για την ανάπτυξη του συστήματός χρησιμοποιήσαμε πόρους από την δημόσια υπηρεσία νέφους του ΕΔΥΤΕ ~okeanos-knossos. Στη διάθεσή μας είχαμε συνολικά τους εξής πόρους:

- 120 GB αποθηκευτικού χώρου σε σκληρό δίσκο
- 12 πυρήνες επεξεργασίας
- 16 GB μνήμης RAM
- 1 δημόσια διεύθυνση IPv4

Για την εκμετάλλευση των παραπάνω πόρων είχαμε στη διάθεσή μας έως 4 εικονικά μηχανήματα (VMs), τα οποία δημιουργήσαμε ως εξής:

- node-0: 2 CPUs, 4 GB RAM, 30 GB HDD, 1 IPv4
- node-1: 4 CPUs, 4 GB RAM, 30 GB HDD
- node-2: 4 CPUs, 4 GB RAM, 30 GB HDD
- node-3: 2 CPUs, 4 GB RAM, 30 GB HDD

Για την επικοινωνία μεταξύ κόμβων γίνεται χρήση ενός δικτύου LAN που περιλαμβάνει τους 4 κόμβους. Επειδή μόνο ένας κόμβος έχει public IPv4, για την επικοινωνία των υπόλοιπων κόμβων με το διαδίκτυο στήθηκε κατάλληλο NAT ώστε να αποκτούν πρόσβαση μέσω του node-0.

Αν και τα εικονικά μηχανήματα είναι βασισμένα σε image που τρέχει Ubuntu 16.04 (το πιο πρόσφατο διαθέσιμο), τα αναβαθμίσαμε έτσι ώστε να έχουν την

τελευταία διαθέσιμη LTS έκδοση (Ubuntu 20.04) και εγκαταστήσαμε επιπλέον docker για να τρέξουμε τις εφαρμογές μας σε containerized μορφή.

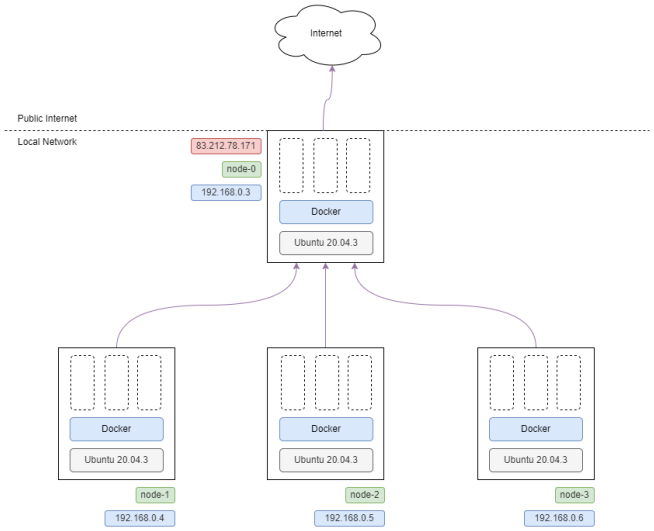


Figure 1. Ανάπτυξη συστήματος σε επίπεδο VM

B. Σε επίπεδο container

Για την πιο γρήγορη ανάπτυξη και τον ταχύτερο πειραματισμό αποφασίσαμε αρχικά να πακετάρουμε τις εφαρμογές μας σε containers. Χρησιμοποιήθηκαν κατα βάση images που παρέχονται από την bitnami (by VMWare) [6] καθώς συντηρούνται τακτικά και παρέχουν μεγαλύτερη ασφάλεια, κάνοντάς τα ιδανικά για σενάρια πραγματικού κόσμου.

Για εφαρμογές μας γραμμένες σε Python επιλέξαμε το **bitnami/python:3.8.10-prod** [7] ως base image και δημιουργήσαμε δύο νέα images, τα sensors και transmitters, που είναι υπεύθυνα για την εκτέλεση του πηγαίου κώδικα με τον κατάλληλο τρόπο.

Στους RabbitMQ brokers χρησιμοποιήσαμε αυτούσιο το **bitnami/rabbitmq:3.9** image [8] και φροντίσαμε να διατηρούμε στο δίσκο τα δεδομένα του broker σε περίπτωση σφάλματος, προσαρτώντας ένα volume κατάλληλα.

Στο Spark cluster που δημιουργήσαμε (spark master, 2 spark workers), καθώς και στους spark drivers που τρέχουν οι εφαρμογές χρησιμοποιήσαμε ως βάση το **bitnami/spark:3** [9] και το επεκτείναμε προσθέτοντας τα κατάλληλα dependencies (.jar files), απαραίτητα για την επικοινωνία με τη βάση δεδομένων MongoDB.

Στο container στο οποίο τρέχει η βάση δεδομένων MongoDB χρησιμοποιήσαμε το **bitnami/mongodb:5.0** [10] και φροντίσαμε ξανά να διατηρούμε τα δεδομένα στο δίσκο σε περίπτωση σφάλματος, προσαρτώντας ένα volume κατάλληλα.

Τέλος, για το Grafana επεκτείναμε κατάλληλα το base image **bitnami/grafana:8.3.3** [11] με την ενσωμάτωση ένος open-source MongoDB plugin [12] σε αυτό. Αυτή η ενέργεια είχε ως στόχο την υποστήριξη της βάσης δεδομένων ως datasource, καθώς δεν ήταν διαθέσιμη στην community έκδοση του Grafana.

Με βάση τις απαιτήσεις των εφαρμογών σε πόρους επιλέξαμε να σηκώσουμε τα εξής containers σε κάθε εικονικό μηχάνημα:

- Node-0
  - sensors - sensors
  - transmitters - transmitters
  - rabbitmq - bitnami/rabbitmq:3.9
  - grafana - grafana-mongodb
- Node-1:
  - sensors - sensors
  - transmitters - transmitters
  - rabbitmq - bitnami/rabbitmq:3.9
  - spark - spark-custom
  - spark-worker - spark-custom
  - spark-driver-late-temp - spark-driver-late
  - spark-driver-late-hum - spark-driver-late
- Node-2:
  - sensors - sensors
  - transmitters - transmitters
  - rabbitmq - bitnami/rabbitmq:3.9
  - spark-worker - spark-custom
  - spark-driver-temp - spark-driver
  - spark-driver-hum - spark-driver
- Node-3:
  - sensors - sensors
  - transmitters - transmitters
  - rabbitmq - bitnami/rabbitmq:3.9
  - mongodb - bitnami/mongodb:5.0

C. Σε επίπεδο services

Ξεκινώντας από την πηγή των δεδομένων, τους sensors, έχουμε στήσει ένα sensors service σε κάθε έναν από τους κόμβους. Κάθε τέτοιο service εκπροσωπεί μια περιοχή (area), A-B-C-D, δημιουργεί δεδομένα εκ μέρους της και τα στέλνει στον MQTT broker.

Όταν τα δεδομένα φτάσουν στο exchange τους δρομολογούνται με βάση το topic τους στην κατάλληλη ουρά (queue) του broker. Ο RabbitMQ broker που έχει στηθεί σε κάθε κόμβο θα μπορούσε να αποτελέσει το δεύτερο αυτοτελές service που κάνει διαθέσιμα τα queues που δημιουργούνται. Παρόλα αυτά τα queues

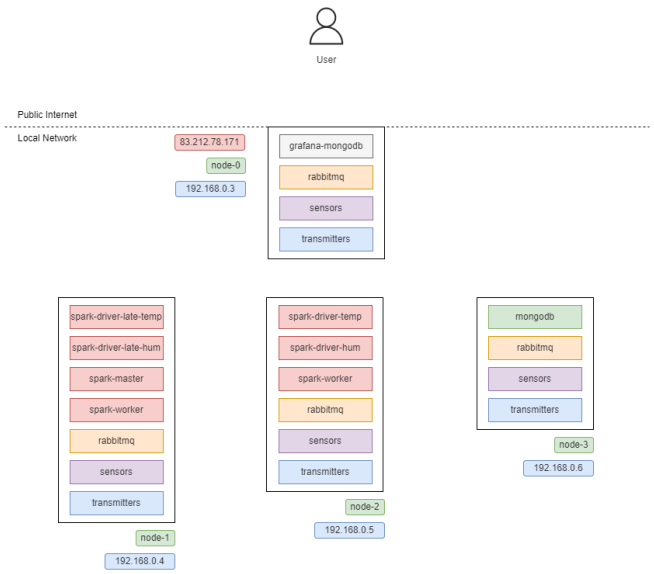


Figure 2. Ανάπτυξη συστήματος σε επίπεδο container

πρέπει να μετατραπούν σε κατάλληλα string streams, ώστε να είναι δυνατόν να καταναλωθούν από τους spark executors στη συνέχεια.

Για αυτό το λόγο έχουμε δημιουργήσει ένα νέο service, που το ονομάζουμε transmitters service και είναι υπεύθυνο για αυτή τη δουλειά. Συγκεκριμένα λειτουργεί από τη μια ως consumer για τις ουρές του broker, ενώ από την άλλη λειτουργεί ως server που εξυπηρετεί πελάτες που θέλουν να λάβουν τα περιεχόμενα του queue. Αυτό που κάνει στην ουσία είναι να παρακολουθεί τις ουρές για νέα δεδομένα και στη συνέχεια να τα κάνει διαθέσιμα στους clients σε συγκεκριμένα ports ως string streams.

Εδώ, αφού καταφέραμε να ομαδοποιούμε και να κάνουμε διαθέσιμα σε clients τα δεδομένα των sensors, στήσαμε ένα spark standalone cluster που αποτελείται από πολλαπλά containers (1 spark-master, 2 spark-worker) και φιλοξενείται σε δύο κόμβους. Σε αυτό το spark service υποβάλλονται οι εφαρμογές που είναι υπεύθυνες για την κατανάλωση και την επεξεργασία των δεδομένων από τους αισθητήρες. Για να γίνει αυτό η κάθε εφαρμογή (application) συντονίζεται από τους spark-drivers που τρέχουν συμπληρωματικά με το spark cluster. Συνολικά τρέχουν παράλληλα 4 εφαρμογές στο spark cluster, ενώ οι αντίστοιχοι spark-drivers είναι μοιρασμένοι στους node-1 και node-2.

Αφού τα δεδομένα υποστούν επεξεργασία, οι spark executors που εκτελούν τις εφαρμογές φροντίζουν να τα αποθηκεύουν στη βάση δεδομένων. Στην περίπτωση μας έχουμε στήσει ένα mongodb service στον node-3. Περιλαμβάνει κανονικά και time-series collections που στοχεύουν στην αποθήκευση διαφορετικών τύπων δεδομένων.

Το τελικό service που στήσαμε φιλοξενεί το Grafana. Είναι υπεύθυνο να παρακολουθεί τη βάση δεδομένων για καινούργια δεδομένα και να τα απεικονίζει στα γραφήματα και τους πίνακες που έχουμε ορίσει.

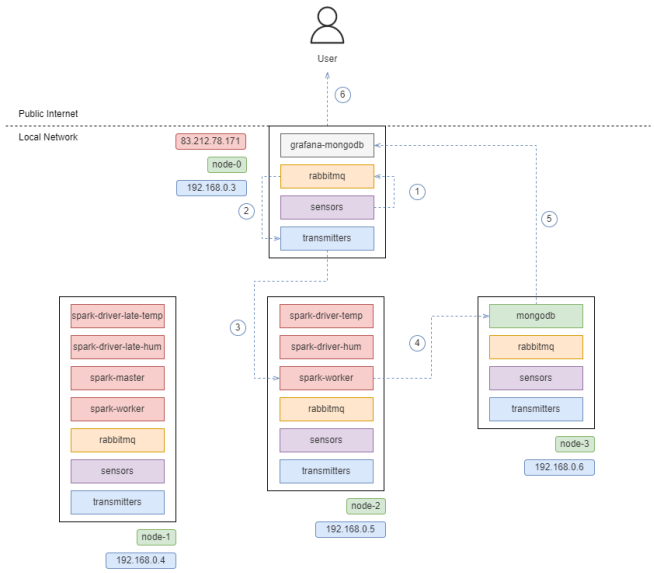


Figure 3. Ροή δεδομένων σε επίπεδο services

D. Αποφάσεις για καλύτερη αξιοποίηση των πόρων

1. Αποφασίσαμε να δημιουργήσουμε 4 διαφορετικά εικονικά μηχανήματα με τους πόρους που μας δόθηκαν. Οι κόμβοι node-0 και node-3 εκτελούν εργασίες όχι και τόσο υπολογιστικά βαριές και για αυτό τους αναθέσαμε μόνο 2 πυρήνες. Αντίθετα, οι κόμβοι node-1 και node-2 όπου τρέχει το spark έχουν μεγαλύτερο υπολογιστικό φορτίο και για αυτό αναθέσαμε 4 πυρήνες στον καθένα.
2. Χρησιμοποιήσαμε 8 αισθητήρες που στέλνουν δεδομένα. Οι 4 εξ αυτών στέλνουν δεδομένα θερμοκρασίας, ενώ οι άλλοι 4 στέλνουν δεδομένα υγρασίας. Η κάθε περιοχή εξυπηρετεί μόνο 2 αισθητήρες, έναν από κάθε είδος. Ας σημειωθεί ότι θα μπορούσαν να χρησιμοποιηθούν και περισσότεροι αλλά επιλέχθηκε ο σχετικά μικρός αριθμός για καλύτερη επιτήρηση της λειτουργίας των αισθητήρων.
3. Συνολικά σε όλο το σύστημα υπάρχουν 16 streams από τα οποία κάποιος μπορεί να λάβει δεδομένα, 8 για δεδομένα θερμοκρασίας και 8 για δεδομένα υγρασίας. Σε κάθε περιοχή αντιστοιχούν 4 streams (ontime-temp, late-temp, ontime-hum, late-hum). Δεν δημιουργήθηκαν περισσότερα streams γιατί ήταν αδύνατο για το περιορισμέ-

νο spark cluster που στήσαμε να παρακολουθούνται όλα τα streams ταυτόχρονα.

4. Στο spark cluster που στήσαμε έχουμε δυο workers με 12 worker cores και 3G worker memory ο καθένας. Αυτές οι τιμές είναι οι ελάχιστες δυνατές για να τρέξει ολόκληρο το σύστημα με 16 streams δεδομένων. Κάθε εφαρμογή χρησιμοποιεί 4 cores για την ανάγνωση των 4 streams και 2 cores για την επεξεργασία των δεδομένων. Επιπλέον, χρησιμοποιείται 1.5G memory ανά εφαρμογή, ενώ έχουμε περιορίσει τον κάθε driver ώστε να χρησιμοποιεί το πολύ 512m memory.

III. Περιγραφή Συστήματος - Λογισμικό

A. Rhea python package

Οι sensors έχουν εγκατεστημένο το rhea [13], το οποίο είναι ένα python package με βοηθητικά εργαλεία τα οποία υλοποιούν ορισμένες χρήσιμες λειτουργίες. Ας δούμε αναλυτικά τα αρχεία που περιέχει:

sender.py

Αυτό το αρχείο υλοποιεί έναν Sender, δηλαδή τη διεργασία που είναι υπεύθυνη για την αποστολή πληροφοριών στο RabbitMQ broker και διατηρεί οποιαδήποτε πληροφορία είναι απαραίτητη για αυτόν. Υποστηρίζει τις παρακάτω λειτουργίες:

- Δημιουργία σύνδεσης με έναν broker χρησιμοποιώντας τις απαραίτητες παραμέτρους και credentials.
- Διακοπή σύνδεσης με έναν broker, αφού προηγηθεί κλείσιμο όσων καναλιών έχουν δημιουργηθεί
- Δημιουργία channel που επικοινωνεί με επιλεγμένο exchange του broker.
- Αποστολή μηνύματος στο κανάλι ενός broker.

broker.py

Αυτό το αρχείο λειτουργεί ως interface για τον RabbitMQ broker, δηλαδή είναι υπεύθυνο για την αρχικοποίηση και διαχείριση του RabbitMQ broker. Υποστηρίζει τις παρακάτω λειτουργίες:

- Δημιουργία σύνδεσης και channel για επικοινωνία με τον RabbitMQ server.
- Διακοπή σύνδεσης, αφού προηγηθεί κλείσιμο του καναλιού έχει δημιουργηθεί.
- Δημιουργία exchange, δηλαδή ενός 'middleman' που αποφασίζει σε ποιο queue να στείλει το μήνυμα που έχει λάβει βάσει του τύπου ανταλλαγής (exchange type).
- Δημιουργία queue, δηλαδή μιας ουράς που διατηρεί τα μηνύματα που στέλνονται ακόμα και σε περίπτωση αποτυχίας του συστήματος.
- Σύνδεση υπάρχοντος queue με επιλεγμένο exchange του Broker.

area_manager.py

Αυτό το αρχείο υλοποιεί έναν AreaManager, δηλαδή τη διεργασία που είναι υπεύθυνη για τη δημιουργία και διαχείριση μιας περιοχής. Σε κάθε περιοχή αντιστοιχεί ένας Broker που αρχικοποιείται μαζί με τον AreaManager. Οι λειτουργίες που υποστηρίζει είναι οι εξής:

- Παροχή interface για χειρισμό του Broker μέσω αυτού, κάνοντας χρήση wrapper functions.
- Δημιουργία όσων Sender θελήσει ο χρήστης και καταχώρησή τους στην περιοχή.
- Στήσιμο ολόκληρης περιοχής με αισθητήρες που στέλνουν δεδομένα σε τακτά χρονικά διαστήματα, αλλά και μη σειριακά στο χρόνο.
- Διακοπή σύνδεσης του Broker, είτε αφού προηγηθεί κλείσιμο των καναλιών όλων των Senders, ή όχι.
- Δημιουργία της απαραίτητης τιμής ενός αισθητήρα βάσει του τύπου του, καθώς και αποστολή της.

B. Sensors - RabbitMQ Broker

Για την αποστολή μηνυμάτων από τους sensors στον RabbitMQ broker, χρησιμοποιούνται τα εργαλεία που παρέχει το rhea υπο τη μορφή του παρακάτω python script:

setup_area_min.py

Το script αυτό αρχικοποιεί έναν AreaManager που αντιπροσωπεύει μια από τις 4 περιοχές A-B-C-D και μέσω αυτού στέλνονται δεδομένα από τους sensors στον Broker της περιοχής. Συγκεκριμένα, δημιουργούνται τα queues:

- ontime_temperature
- late_temperature
- ontime_humidity
- late_humidity

δηλαδή 2 queues για κάθε είδος sensor, ένα για δεδομένα σειριακά στο χρόνο και ένα για τυχαία δεδομένα μη σειριακά στο χρόνο. Αφού δημιουργηθούν και οι απαραίτητοι Senders, δηλαδή αισθητήρες, ξεκινάει η διαδικασία αποστολής των δεδομένων. Στέλνονται με διαφορετικό topic και το exchange του RabbitMQ broker που τα δέχεται, τα διαχωρίζει σε ένα από τα παραπάνω queues.

Η επιλογή ορισμένων χρήσιμων παραμέτρων για την περιοχή γίνεται μέσω μεταβλητών περιβάλλοντος. Αυτό επιτρέπει στην εφαρμογή να χρησιμοποιείται εύκολα σε containerized μορφή. Οι μεταβλητές μπορούν να οριστούν κατά τη δημιουργία του container και να χρησιμοποιηθούν από το python script.

Παράδειγμα μεταβλητών περιβάλλοντος:

```
environment:
  - AREA=A
  - BROKER_IP=localhost
  - BROKER_PORT=5672
  - BROKER_USER=rheaAdmin
  - BROKER_PASSW=adminPassword
  - SEND_INTERVAL=900
  - LATE_PCT=0.0333
---
```

C. RabbitMQ Broker - Transmitters

Όπως αναφέρθηκε παραπάνω, δημιουργήσαμε ένα νέο service το οποίο ονομάσαμε transmitters, προκειμένου να μετατραπούν τα παραπάνω queues σε κατάλληλα string streams και να είναι δυνατόν να καταναλωθούν από τους spark executors.

Αρχικά, δημιουργούμε ένα socket στο οποίο κάνουμε listen, περιμένοντας να συνδεθεί κάποιος client σε αυτό. Όταν η σύνδεση γίνει αποδεκτή, τότε δημιουργούνται η σύνδεση και το κανάλι με τον broker και ξεκινάμε το consumption των μηνυμάτων που έχουν σταλεί/στέλνονται. Από αυτό αντιλαμβανόμαστε πως πρώτα πρέπει να εκκινήσουν οι sensors για να δημιουργηθούν οι brokers και έπειτα οι transmitters.

Το consumption των μηνυμάτων χρησιμοποιεί μια callback συνάρτηση η οποία κάνει format το μήνυμα αντικαθιστώντας τα single-quotes με double-quotes και προσθέτοντας νέα γραμμή, εφόσον το Spark διαβάζει γραμμή-γραμμή, διατηρώντας το UTF-8 encoding. Στη συνέχεια στέλνει το μήνυμα στο socket, και άρα στον remote client που είναι συνδεδεμένος.

Το σύστημα επιτρέπει γενικά την αποσύνδεση ενός client και την επανασύνδεσή του στο stream, όποτε αυτός το επιθυμεί. Επιπλέον, μπορούμε να σηκώσουμε αρκετά transmitters ταυτόχρονα αν κάνουμε χρήση του αρχείου run.py. Αυτό μας επιτρέπει ορίζοντας τις κατάλληλες μεταβλητές περιβάλλοντος να σηκώσουμε τα text streams στα ports που επιθυμούμε.

Παράδειγμα μεταβλητών περιβάλλοντος:

```
---
environment:
  - BROKER_IP=192.168.0.3
  - BROKER_PORT=5672
  - BROKER_USER=rheaAdmin
  - BROKER_PASSW=adminPassword
  - QUEUES=["ontime_temp", "late_temp"]
  - QUEUE_PORTS=["9000", "9001"]
---
```

(Αρχεία: transmitter.py, run.py)

D. Transmitters - Spark

Για την λειτουργία του Spark, το αρχικοποιούμε με το configuration που εξηγήθηκε παραπάνω και για κάθε γνωστό broker δημιουργούμε socket text streams που συνδέονται στα transmitters χρησιμοποιώντας τη διεύθυνση και το port που τους αντιστοιχούν.

Στη συνέχεια, τα socket text streams δίνουν DStreams (λειτουργικότητα του Spark Streaming), τα οποία συγχωνεύουμε τελικά σε ένα DStream. Ακολουθεί η μετατροπή του μηνύματος από μορφή json σε μορφή (key, value), ώστε να μπορεί να γίνει η απαραίτητη επεξεργασία. Το json μήνυμα περιέχει:

- το ”sender_no” που είναι τύπου int και αντιπροσωπεύει τον αύξων αριθμό του αποστολέα
- το ”value” που είναι τύπου float και αντιπροσωπεύει τη μέτρηση του αισθητήρα
- το ”created_timestamp” που είναι τύπου time και αντιπροσωπεύει τη χρονική στιγμή που έχει δημιουργηθεί το μήνυμα και
- το ”event_timestamp” που είναι επίσης τύπου time και αντιπροσωπεύει τη χρονική στιγμή που στάλθηκε το μήνυμα (διαφοροποιείται από το “created_timestamp” στα late events).

Ως key θέτουμε το ”sender_no” και ως value θέτουμε το tuple (value, created_timestamp) στα ontime events και το tuple (value, created_timestamp, event_timestamp) στα late events. Πρέπει όμως πρώτα να επεξεργαστούμε τα timestamp, και να τα φέρουμε στη μορφή που θέλουμε ώστε να είναι δεκτά από το Time-series plugin της MongoDB.

Έπειτα, ανάλογα με το αν το event είναι ontime ή late, συμβαίνει ένα από τα παρακάτω:

- Ontime event: δημιουργούμε 4 διαφορετικά DStream, βασισμένα στο DStream που έχει δημιουργηθεί παραπάνω, ώστε να γίνει η απαραίτητη επεξεργασία (aggregation). Πιο συγκεκριμένα, κάνουμε τα απαραίτητα reduce ώστε να καταλήξουμε με DStreams για MIN, MAX, SUM και AVG, δηλαδή DStreams που να περιέχουν την ελάχιστη τιμή, τη μέγιστη τιμή, το άθροισμα τιμών και το μέσο όρο τιμών όλων των μετρήσεων που λήφθηκαν εντός ενός χρονικού παραθύρου 3600 δευτερολέπτων (μιας ώρας).
- Late event: δε γίνεται κάποια παραπάνω επεξεργασία, απλά διατηρείται το merged DStream με χρονικό παράθυρο επίσης 3600 δευτερολέπτων.

(Αρχεία: ontime_consumer.py, late_consumer.py)

E. Spark - MongoDB

Αφού τα αρχεία υποστούν επεξεργασία από το Spark, για κάθε RDD που προκύπτει από ένα DStream καλείται η συνάρτηση που τα αποθηκεύει στη βάση δεδομένων. Συγκεκριμένα, το RDD μετατρέπεται σε DataFrame σύμφωνα με ένα schema και έπειτα αποθηκεύεται στο collection που του αντιστοιχεί. Αν είναι late event, αποθηκεύεται σε ξεχωριστό collection λαμβάνοντας

ταυτόχρονα υπόψιν και το είδος του αισθητήρα, ενώ αν είναι ontime event αποθηκεύεται βάσει του αισθητήρα και του είδους του aggregation που έχει γίνει. Δηλαδή υπάρχουν τα παρακάτω collections:

Ontime events collections:

- Temperature
  1. Max: tempMax
  2. Min: tempMin
  3. Sum: tempSum
  4. Avg: tempAvg
- Humidity
  1. Max: humMax
  2. Min: humMin
  3. Sum: humSum
  4. Avg: humAvg

Late events collections:

- lateTemp
- lateHum

Εκτός αυτού, το schema που χρησιμοποιούν τα ontime events είναι διαφορετικό από το schema που χρησιμοποιούν τα late events:

- Ontime events:

```
{ "metadata": {
    "sender_no": integer
    "type": string
  },
  "timestamp": timestamp,
  "value": double
}
```

- Late events:

```
{ "sender_no": integer,
  "area": string,
  "value": double,
  "event_timestamp": timestamp,
  "created_timestamp": timestamp
}
```

Παρατηρούμε πως τα late events, αντικαθιστούν το πεδίο “timestamp” με “event_timestamp” και “created_timestamp”, εφόσον όπως αναφέρθηκε, είναι διαφορετικά μεταξύ τους.

(Αρχεία: ontime_consumer.py, late_consumer.py)

F. MongoDB - Grafana

Όσον αφορά την επικοινωνία MongoDB με Grafana αξίζει να αναφέρουμε ότι χρειάστηκε να χρησιμοποιήσουμε κατάλληλα ένα open source plugin. Αυτό το plugin προσφέρει ένα είδος proxy που μετατρέπει το Grafana Data Source API σε κατάλληλα queries που επικοινωνούν με τη βάση δεδομένων για να λάβουν τα δεδομένα που ζητούνται.

Για να πάρουμε την πληροφορία που χρειάζεται συμβουλευτήκαμε το περιορισμένο documentation και το επεκτείναμε κατάλληλα. Αρχικά, δημιουργήσαμε panels στο Grafana, τα οποία έχουν ως Data Source τη MongoDB, και προσθέσαμε queries για τα δεδομένα τόσο σε Time-series όσο και Table μορφή. Τα 4 panels που προβάλλονται είναι τα εξής:

- Temperature
- Temperature Late Events
- Humidity
- Humidity Late Events

Τα ontime panels έχουν 4 queries (MIN, MAX, SUM, AVG) για Time-series δεδομένα όπως το παρακάτω:

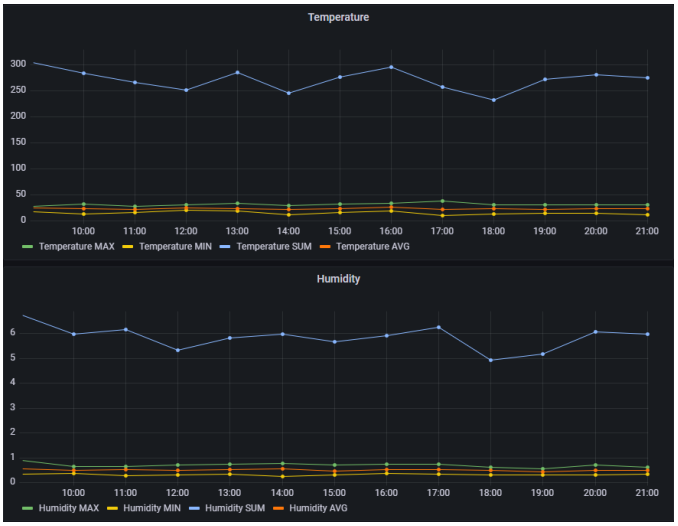
```
db.tempMax.aggregate(
[
    {
        "$project": {
            "name": "Temperature MAX",
            "value": "$value",
            "ts": "$timestamp",
            "_id":0
        }
    }
])
```

Τα late events panels έχουν 1 query για Table δεδομένα.

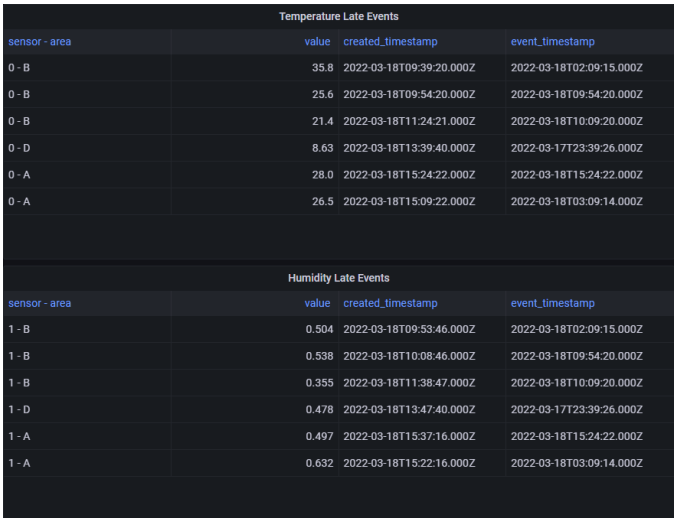
```
db.lateTemp.aggregate(
[{  "$match" :  {
        "created_timestamp" : {
            "$gte" : "$from",
            "$lt" : "$to"
            }
        }
    },
    {"$project": {
      "sensor - area" : {
      "$concat" : [
          {"$substr":
            ["$sender_no",0,-1]
          },
          " - ",
          "$area" ]
      },
      "value": "$value",
      "created": "$created_timestamp",
      "event": "$event_timestamp",
      "_id":0
    }
}])
```

G. Grafana - User

Ο χρήστης (admin) μπορεί να δει τα αποτελέσματα του συστήματος σε 4 panels, τα 2 που περιέχουν ontime events σε μορφή γραφικής παράστασης και τα 2 που περιέχουν late events σε μορφή πίνακα:



Σχήμα 4. Grafana Time-series Panels Overview



Temperature Late Events

sensor - area	value	created_timestamp	event_timestamp
0 - B	35.8	2022-03-18T09:39:20.000Z	2022-03-18T02:09:15.000Z
0 - B	25.6	2022-03-18T09:54:20.000Z	2022-03-18T09:54:20.000Z
0 - B	21.4	2022-03-18T11:24:21.000Z	2022-03-18T10:09:20.000Z
0 - D	8.63	2022-03-18T13:39:40.000Z	2022-03-17T23:39:26.000Z
0 - A	28.0	2022-03-18T15:24:22.000Z	2022-03-18T15:24:22.000Z
0 - A	26.5	2022-03-18T15:09:22.000Z	2022-03-18T03:09:14.000Z

Humidity Late Events

sensor - area	value	created_timestamp	event_timestamp
1 - B	0.504	2022-03-18T09:53:46.000Z	2022-03-18T02:09:15.000Z
1 - B	0.538	2022-03-18T10:08:46.000Z	2022-03-18T09:54:20.000Z
1 - B	0.355	2022-03-18T11:38:47.000Z	2022-03-18T10:09:20.000Z
1 - D	0.478	2022-03-18T13:47:40.000Z	2022-03-17T23:39:26.000Z
1 - A	0.497	2022-03-18T15:37:16.000Z	2022-03-18T15:24:22.000Z
1 - A	0.632	2022-03-18T15:22:16.000Z	2022-03-18T03:09:14.000Z

Σχήμα 5. Grafana Table Panels Overview

IV. Αποτελέσματα - Συμπεράσματα

Όπως μπορεί κάποιος εύκολα να αντιληφθεί, ξεκινώντας από την παραγωγή δεδομένων σε επίπεδο αισθητήρων καταφέραμε να παρουσιάσουμε με ουσιαστικό τρόπο χρήσιμη πληροφορία σε διαγράμματα, και όλα αυτά σε πραγματικό χρόνο! Το σενάριο που προσπαθήσαμε να εξομοιώσουμε ήταν ένα data center που έχει 4 περιοχές από όπου συλλέγουμε δεδομένα από αισθητήρες θερμοκρασίας και υγρασίας. Στη συνέχεια, επεξεργαζόμαστε κατάλληλα τα δεδομένα και δίνουμε μια εικόνα της συνολικής κατάστασης του data center σε ωριαία βάση. Εκτός αυτού, με την παρουσίαση καθυστερημένων μετρήσεων μπορούμε να εντοπίσουμε σε ποια περιοχή του data center και σε ποια συγκεκριμένη χρονική στιγμή δημιουργήθηκαν προβλήματα. Αυτό μας δίνει τη δυνατότητα να εντοπίσουμε τυχόν βλάβες εγκαίρως ή να κατανοήσουμε άλλα φαινόμενα που συνέβησαν στο data center εκείνη την χρονική περίοδο.

Αυτό το σενάριο πρφανώς είναι πολύ περιορισμένο σε σχέση με μια περίπτωση ενός πραγματικού data center, όπου υπάρχουν χιλιάδες αισθητήρες και η συνολική έκταση που καλύπτουν είναι τεράστια. Παρόλα αυτά, ο τρόπος με τον οποίο σχεδιάσαμε το σύστημα κανει μια απόπειρα να λάβει υπόψιν σενάρια μεγαλύτερης κλίμακας. Αρχικά, αναπτύσσοντας εφαρμογές σε containerized μορφή και δίνοντας τη δυνατότητα να δημιουργηθούν πολλαπλές περιοχές με πολλαπλούς αισθητήρες εξομειώνει αισθήτηρες και gateways ενός πραγματικού IoT συστήματος. Συμπληρωματικά, στήνοντας ένα spark cluster όπου τρέχουν παράλληλα πολλά applications υπό περιορισμό πόρων και το οποίο είναι εύκολα επεκτάσιμο με προσθήκη νέων workers, λαμβάνουμε υπόψιν την ευελιξία κλιμακωσιμότητας που απαιτούν τα σύγχρονα συστήματα. Τέλος, η βάση δεδομένων σε κάθε περίπτωση μπορεί να υποστηρίξει μεγαλύτερα σενάρια λειτουργίας με την προσθήκη replicas.

Ολοκληρώνοντας, είναι αναγκαίο να γίνει μια αναφορά σε πιθανές επεκτάσεις του συστήματος που αφορούν όχι τόσο την ύπαρξη περισσότερων υπολογιστικών πόρων, αλλά τη φύση του συστήματος. Μια ενδιαφέρουσα επέκταση θα ήταν η ανάπτυξη του σε περιβάλλον Kubernetes [14], σε αντίθεση με το ”ψευδό” K8s που δημιουργήσαμε στην υλοποίησή μας. Αυτή η επέκταση θα πρόσεφερε μεταξύ άλλων τη δυνατότητα αυτοματοποιημένης κλιμακωσιμότητας και ανάνηψης από σφαλματα. Μια άλλη επέκταση θα μπορούσε να σχετίζεται με την εξυπηρέτηση πολλαπλών consumer για ένα συγκεκριμένο stream δεδομένων. Για παράδειγμα, ένα ενδιάμεσο σύστημα Pub/Sub [15] θα επέτρεπε την επεξεργασία των ροών δεδομένων από πολλαπλούς consumers με μεγαλύτερη ευελιξία. Τέλος, καθώς κατά την ανάπτυξη του συστήματος αντιμετωπίσαμε δυσκολίες στον ακριβή προσδιορισμό των πόρων που απαιτούνται για να τρέξει ικανοποιητικά, η ενσωμάτωση ενός άλλου εξωτερικού συστήματος παρακολούθησης της χρήσης των υπάρχοντων πόρων (cpu usage, memory usage, network metrics etc.) [16] θα αποτελούσε μια χρήσιμη επέκταση της όλης λειτουργίας.

Αναφορές

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

[2] *Spark Streaming Programming Guide*. https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[3] *MongoDB: The Application Data Platform.* https://www.mongodb.com/.

[4] *MongoDB Time Series Collections.* https://docs.mongodb.com/manual/core/timeseries-collections/.

[5] *Grafana: The open observability platform.* https://grafana.com/.

[6] *Bitnami By VMWare.* https://hub.docker.com/u/bitnami.

[7] *Bitnami/python image*. https://hub.docker.com/r/bitnami/python.

[8] *Bitnami/rabbitmq image.* https://hub.docker.com/r/bitnami/rabbitmq.

[9] *Bitnami/spark image.* https://hub.docker.com/r/bitnami/spark.

[10] *Bitnami/mongodb image*. https://hub.docker.com/r/bitnami/mongodb.

[11] *Bitnami/grafana image*. https://hub.docker.com/r/bitnami/grafana.

[12] *MongoDB datasource for Grafana.* https://github.com/JamesOsgood/mongodb-grafana.

[13] *Rhea Python package.* https://github.com/panoszvg/NTUA-Analysis-and-Design-of-Information-Systems/tree/main/rhea-pkg.

[14] *Kubernetes, Production-Grade Container Orchestration.* https://kubernetes.io/.

[15] *What is Pub/Sub?.* https://cloud.google.com/pubsub/docs/overview.

[16] *Prometheus - Monitoring system & time series database.* https://prometheus.io/.