CSCI 5253: Datacenter Scale Computing
Fall 2020

Title: thingsIO - IoT analytics application
Participants: Sanskar Katiyar, Siddhant Keshkar
Project (Github): https://github.com/sanskarkatiyar/thingsIO

1. Project: Motivation, Overview and Goals

We built a scalable service where users can transmit the measurements of their IoT devices in real-time, visualize it on a dashboard and perform common analytics operations on them. One of the main goals was to provide a seamless service that can monitor your IoT cluster (>= 1 sensor) on a single platform with practical rate-limits, in contrast to commercial platforms which enforce limits on the usage of various features. This service had to be scalable to support a large number of users and device measurements.

Since a device can incorporate multiple sensors, we wanted to provide the user the ability to dictate the exact format of measurements instead of conforming their setups to the platform. To offer a one-stop solution for both telemetry and analysis, we built a custom analytics application. Our analytics application supports common time-series data analysis (since that is the nature of IoT data, for most part).

In this report, we describe the behavior of our components and architecture, as well as the conscious choices that lead to their selection.

Below is the user flow of the application, to give you an idea about the end-user experience of the application, as well as a corresponding video detailing our service (https://youtu.be/oSRw8uqio6g):

User flow of the application

1. The user registers and logs-in to the application which creates credentials to authenticate their requests to the service.
2. The user then submits a schema for their device, essentially telling the service about their device's measurement format.
3. The device, then, sends data to the service using the user's API key.
4. The emerging locations of the data points and plots are updated live on the dashboard.
5. The user then specifies the analytics operations they require for their device data, and the application then displays this on the dashboard.

2. Components and Purpose

- Web framework: Flask
- Message Queue: RabbitMQ (pika)
- Containers: Docker
- Container Orchestration: Kubernetes
- Databases: Redis (user management, general), InfluxDB (time-series)
- Analytics: pandas, matplotlib, numpy, sci-kit learn
- Front-end: Bootstrap, JS, ChartJS, Leaflet (Overpass)
- Cloud environment: Google Cloud, GKE (Standard 3-node cluster)

Flask: We chose Flask primarily to develop our web application on *dashboard* and *rest*. We have used it because it has minimal dependencies on external Python libraries, is fast and easy to use; and has the ability to scale up to complex applications. It comes with a small drawback though - since, it has a singular source which means that it will handle every request in turns, one at a time. So if you are trying to serve multiple requests, it will take more time. There is also no good way to relay asynchronous updates.

RabbitMQ (pika): For our Messaging Service we chose RabbitMQ because messages are guaranteed delivery and managed through brokers. It is optimized for discrete message handling. RabbitMQ is not as fast as Apache Kafka - that's because Kafka batches messages and does not offer a robust routing and security feature set. However, we want to ingest each point as it comes so as to not tamper with the timestamps and relay the data to the user's dashboard and make it available for analytics as quickly as possible.

***Redis and InfluxDB*:** Our data storage pipeline for sensor data is heavily optimized for IoT data. Hence, we chose InfluxDB as our primary database for storing the time series data since it provides quick fetching and aggregation operations.The incoming data will generally be time-series data and will never be modified. It will, however, be read multiple times by the dashboard and analytics pods. Redis is used for storing user authentication data, schema data and the results for analytics jobs (*kind of over-worked Redis there*).

***Docker and Kubernetes*:** We used Docker for containerizing our applications (rest, dashboard, analytics, ingestor, logs) as well as our databases and messaging queue, so that they can run independently. Docker serves as a great tool for making

applications run exclusively. Most of them use a python-3-slim base image and all the docker images can be found here: https://hub.docker.com/u/sanskarkatiyar

Containerizing these applications is not enough. Kubernetes is used to manage these application pods and auto-scale them (horizontally) based on the number of requests and other metrics, such as memory usage, cpu usage, etc.

Front-end: We wrote custom applications for front-end (dashboard, rest) and analytics, for time-series data. These applications can run independently, apart from the interaction with the messaging queue and databases. The dashboard application (run by Flask) handles the rendering of web pages. The view layer is constructed using Bootstrap for responsive components and behavior, and uses Leaflet (Tiles: OpenStreetMaps) and ChartJS libraries to display the maps and plots respectively.

These components allowed us to provide a better UX as well as delegate plots processing to the user's browser. The problem with Flask as a backend is that it needs to read the dataframe from InfluxDB and convert it into a format suitable for ChartJS and Leaflet. This also requires Flask to perform additional processing apart from storing the content of the dataframes in the app-context.

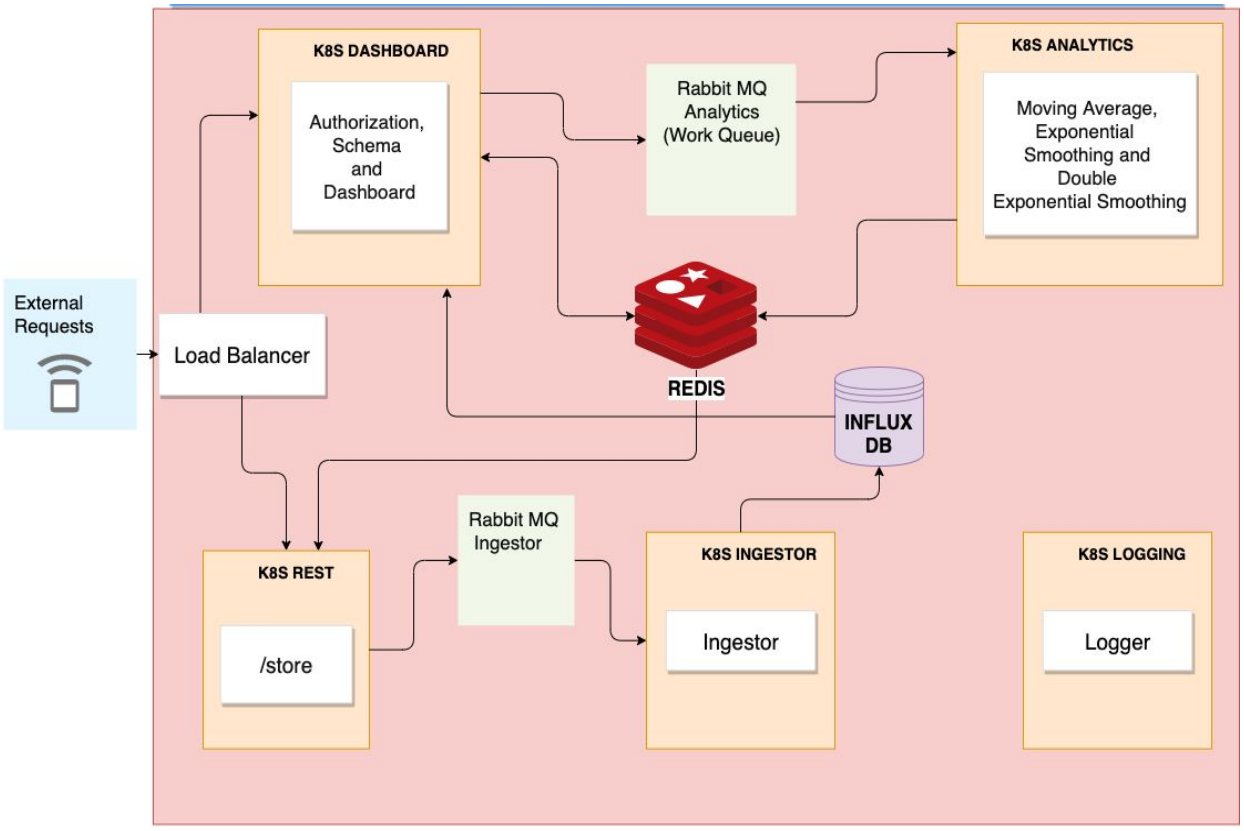
Analytics: To observe the time series IoT data, we created three different analytic operations namely - largely as a proof of concept for more complex operations - moving average, exponential smoothing and double exponential smoothing using the sci-kit learn metric methods. Scikit-learn is a free-to-use machine learning module built on SciPy to be used with Python. We chose it because it's a simple but effective data analysis tool. We have used Matplotlib, a plotting library for python to visualize the datasets and operations.

We transformed operations such that they resulted in a list of images (plots); these were stored as a binary image for ease of implementation of the view layer, but the network usage would have been much lower in case of relaying the binary dataframe along with the supplemental information.

3. Architecture and Interaction between components

External requests sent to the gateway are redirected between endpoints `/auth` (`/account` if logged-in), and `/store`. If the request is meant to register or login for the user it is redirected to the `/auth` service endpoint. If the request is meant to store a datapoint then it is redirected to the `/store` service endpoint and if the user wants to view the data, perform analytics tasks then the request will be redirected to the `/account` service endpoint. These endpoints refer to `dashboard` application pods managed by Kubernetes. We have also used the standard `redis`, `rabbitmq` and

`influxdb` deployments and services for this application. We have exposed the `dashboard` and `rest` service using the Kubernetes ingress controller. We have a service endpoint for analytics applications, also managed by Kubernetes, but is inaccessible to the user directly.



As soon as a new user registers, their username and password hash gets stored into the redis database (db=0), and is fetched for login authorization. It also creates a new API_KEY unique to the user which is stored along with the username in the redis database (db=1). When the user submits a schema for their device, it stores the API_KEY and the corresponding schema into the redis database (db=3) as a key-value pair.

The Ingestor messaging queue is used to line up our `/store` requests at the ingestion site. At the end of this queue, an ingestor verifies the point against the schema of the application, if there exists a valid one, then inserts it into InfluxDB. When a new analytics operation request is created by the user in the dashboard, it creates a new JOB_ID and stores it along with the API_KEY into the redis database (db=4). The Analytics queue is a work queue which is set to receive analytics job requests from the dashboard waiting to be consumed by the Analytics Pod. Once the job request is

processed the resulting plotted images are stored into the redis database (db=5) along with the respective JOB_ID.

We describe these Kubernetes deployments as follows:

a. **Rest**: Responsible for taking data point storage requests for a user and verifying if the user exists then pushing it onto the Ingestion Messaging Queue.
b. **Ingestor**: Responsible for consuming valid data point storage requests from the Ingestion Messaging Queue and storing them into the Influx database.
c. **Dashboard**: Responsible for providing a front-end interface for the user to display plots and location for the data, submit schema for their device(s), submit analytics jobs and provide results to analytics operations.
d. **Analytics**: Responsible for retrieving analytics jobs from the work queue to process them and store them in the redis database. The unique thing is that it directly receives the data with the request, since dataframes can be compactly packed.
e. **Logging**: Responsible for logging all info/error data along with the host-ip, type and timestamps, across different deployments.

All of our queues are durable to ensure that the failure/restart of the broker does not result in lost measurements and job requests.

4. Testing and Debugging

To test our platform, particularly the ingestion, we wrote a custom batch request simulator. This simulator sends about ~50 requests per second per host, to numerous custom-created API keys (accounts) on the platform. This provided us with a rough overview of different rate-limiting mechanisms we may want to enforce for a huge load of traffic.

We also created a logger (pod) to log info and error data along with the host-ip, type and timestamps, across different pods. This was a great tool, which helped us debug numerous issues with respect to our messaging queue and locate exact error sequences.

We also leveraged the GKE monitoring service to observe any key resource changes for each of our deployments. Using the stress-test with the batch simulator, we were able to identify the load each of our pods was taking under various situations as well as identify the resource usage. This helped us vastly in setting our resource requests and limits for each deployment phase.

5. Capabilities, Limitations and Future Work

Capabilities

We have enabled automatic horizontal scaling on our ingestor, analytics and dashboard deployments via GKE. This will allow us to support oscillating traffic without much intervention. An example scenario would be that the users will tend to use the analytics and dashboards during the day, where these pods can easily scale. Ingestors may also receive batch requests/updates if a device has an intermittent connection and thus, will require autoscaling, as well. This can be handled by the platform with ease.

We also started with a minimum of two replicas for our messaging queue since this is a crucial component of what ties the system together. This redundancy helps the remaining system to function well, even if something goes wrong in one of the sections.

As highlighted earlier, the ingestion and fetching pipelines are heavily optimized for time-series data, due to which query execution and fetching is much faster.

Below, we also highlight the capabilities of our platform by comparing it to the popular commercial IoT platform Thingspeak (https://thingspeak.com/):

1. Currently, we allow an arbitrary number of sensors (fields) per account to be set up on our platform. Thingspeak, on the other hand, limits it to 8 fields (maximum).

2. We are also primed for both location and numeric data, whereas Thingspeak can only store and aggregate numeric data.

3. Thingspeak is heavily rate limited: 1 request per 15 seconds. Thanks to our design of the ingestor and a durable request queue, we can adjust to heavy influx of data without limiting it to more than a second.

4. Open source: we plan to publish this as an open-source project, so organizations can monitor their IoT device data without paying for a premium Thingspeak setup, as well as locally customizing and managing their IoT clusters.

Limitations and Future work

Our application can do with a number of improvements and these will be on our horizon for future work, as well. Thus, each statement below covers our reaction to the limitation as well.

1. A major limitation of our platform is that it currently supports only HTTP requests. Our goal will be to support protocols like MQTT, which is popular with IoT

devices and is much more efficient for transferring data over low bandwidth connections.

2. In its current state, the application relies heavily on Redis serving as a general-purpose database. This is not a good idea (and was done primarily for ease), especially when one needs to store results of analytics operations, that may involve binary data. A better idea would be to use another database such as PostgreSQL. Our plan is to continue using Redis for quick lookups of valid API key and schemas at the ingestor, as well as set up the streaming data queue for live updates on the dashboard while delegating the low frequency fetch-update options like credentials and analytics results go to PostgreSQL.

3. The analytics package currently does not offer as many operations as we would have liked. Currently, it may seem odd to keep a separate analytics deployment but with more complex operations (location, network analysis) and deep learning models, the analytics pod(s) will be better off running independently of the dashboard application to avoid the network overheads and allow dashboard users to explore the telemetry data without interruption.

4. We would like to improve the front-end. (i) Currently, the dashboard does not offer a ton to explore the data (change time-zones, window sizes, etc.) - and we would like to improve the UX. (ii) At the same time, avoid double parsing the dataframes between Flask and ChartJS - as otherwise the app stores the dataframe in memory within the app-context. (iii) One of the other potential improvements, here, involves relaying live-updates to the user's dashboard using SocketIO or Server-sent events. We would also like to implement running analytics operations with asynchronous event updates.

5. Finally, we would like to (i) extend support for multi-dimensional measurement such as data from an accelerometer, gyroscope, etc. and (ii) write a custom header which can offer batch sending of the data to the service for poorly connected devices.

Project (Github): https://github.com/sanskarkatiyar/thingsIO

References:

1. Prof. Grunwald's lecture material
2. Docker Documentation: https://docs.docker.com/
3. Kubernetes Documentation: https://kubernetes.io/docs/home/
4. Bootstrap Documentation: https://getbootstrap.com

5. Leaflet Documentation: https://leafletjs.com/reference-1.7.1.html
6. Flask Documentation: https://flask.palletsprojects.com/en/1.1.x/
7. InfluxDB Documentation: https://docs.influxdata.com/influxdb/v2.0/
8. Redis (Python) Documentation: https://github.com/andymccurdy/redis-py
9. Pika Documentation: https://pika.readthedocs.io/en/stable/index.html
10. Analytics:https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775
11. StackOverflow and Open Github Issues