Applied Distributed Systems
Final Project Report

Team Members:

Akhila Eshapula - akesha@iu.edu
Amita Ranade - asranade@iu.edu
Steve Mendis - smendis@iu.edu

Github:

We have uploaded the source code with github.iu.edu at following URL

URL: https://github.iu.edu/smendis/applied-distributed-systems-project/tree/main

Project Summary:

In this project, we have implemented a comprehensive data analysis system that generates random numbers using statistical models such as EMA-bollinger bands, ARIMA, and outlier detection. The generated data is then put into a MQTT queue and collected for analysis. K-means clustering is performed on the data, and the output generated is stored in a database. Finally, the results are displayed on a user interface, which shows the abnormal/normal status of the data using ReactJS.

Research Methodology and Findings:

We have explored three different messaging protocols: MQTT, Kafka, and CoAP. MQTT is a lightweight protocol designed for IoT applications with limited processing power and battery life. It uses a publish-subscribe model and is suitable for small-scale deployments. Kafka, on the other hand, is a distributed streaming platform designed for high-throughput, real-time data processing. It uses a publish-subscribe and queueing model and is suitable for big data applications. CoAP is a lightweight protocol intended for use in IoT devices with limited resources. It uses a publish-subscribe model and is designed to be simple and efficient.

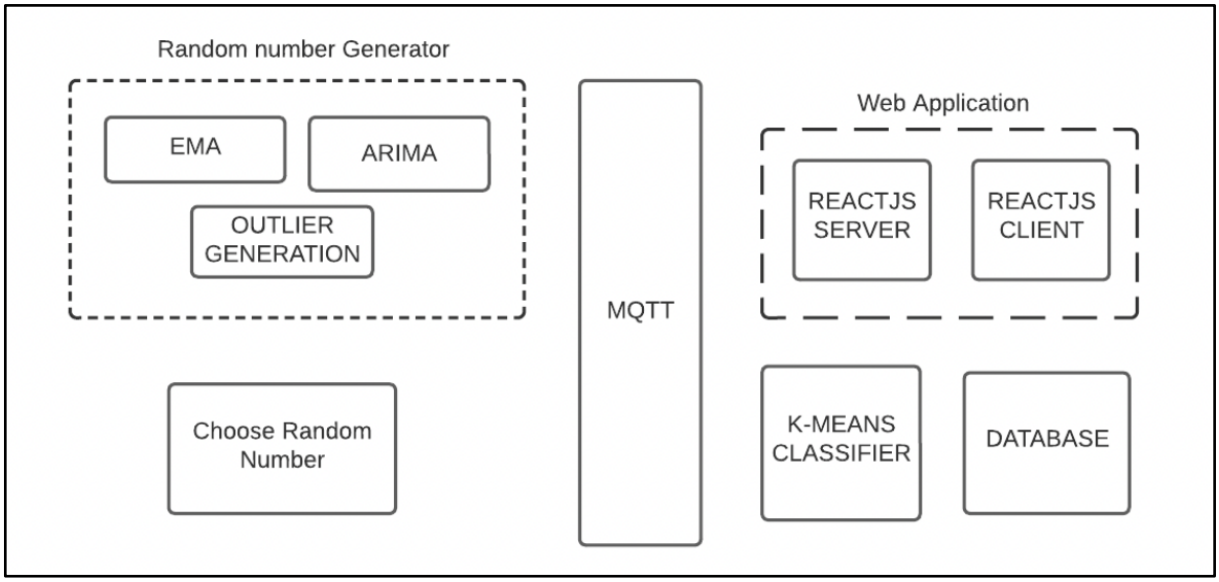
MQTT and CoAP are both publish-subscribe protocols, while Kafka uses a publish-subscribe and queueing model. MQTT and CoAP allow clients to subscribe to topics and receive messages from publishers, while Kafka allows producers to write messages to topics, and consumers can either subscribe to those topics or consume messages from a queue.

Overall, MQTT is suitable for small-scale IoT deployments, Kafka is suitable for big data applications, and CoAP is suitable for resource-constrained IoT devices. We have chosen MQTT for our project since we have chosen small dataset of a telemetry company.

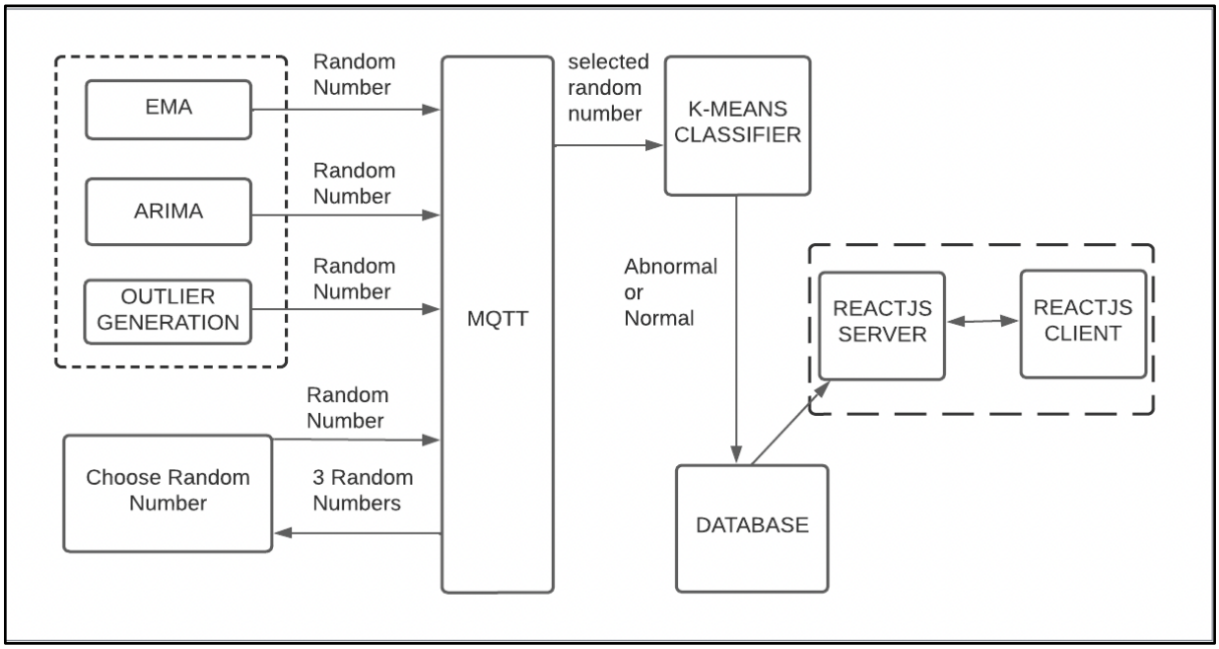
Technical Description:

FLOW:

EMA-bollinger bands, ARIMA, and outlier detection for generating random numbers
MQTT queue for collecting the generated data
K-means clustering for analyzing the data
Database for storing the output of the analysis
ReactJS for displaying the abnormal/normal status of the data on a user interface
Docker for containerizing the entire system



DATAFLOW DIAGRAM:

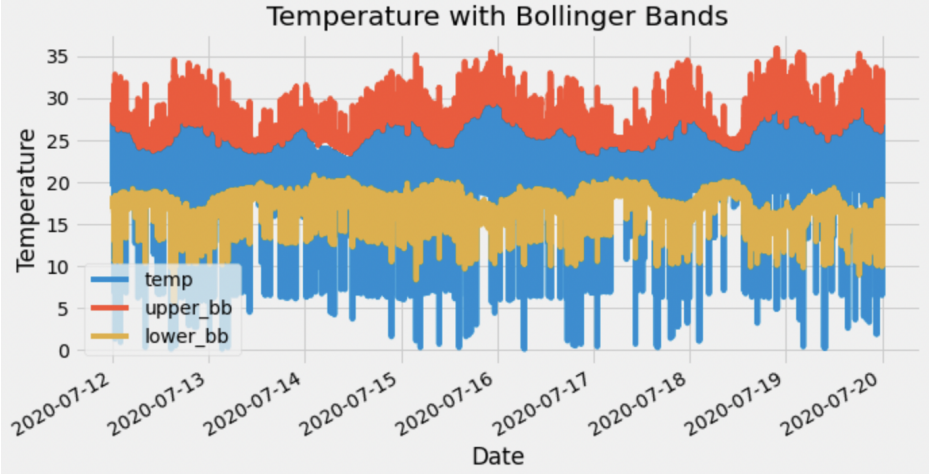


DOCKER

We have created 9 containers and distributed our project. All the containers run using docker compost for orchestration.

EMA

The EMA code implements a statistical technique called Exponential Moving Average (EMA) to generate random numbers. The code reads in a dataset of IoT telemetry data and selects the timestamp and temperature columns. It then calculates the Simple Moving Average (SMA) and Standard Deviation (STD) of the temperature data using a rolling window of 20 data points. Using the SMA and STD, it calculates the upper and lower Bollinger Bands.



The code generates a random number between the upper and lower Bollinger Bands and adds uniform noise to it. The resulting random number is then published to a MQTT topic using Paho MQTT client library. The code waits for 60 seconds before generating the next random number. The published data can be used for further analysis.

ARIMA

In this part of the project, we implemented an ARIMA (Autoregressive Integrated Moving Average) analysis on the IoT telemetry data using PySpark and Statsmodels. We started by loading the dataset into a Spark DataFrame, then we converted it to a Pandas DataFrame to perform the ARIMA analysis.

We used the ARIMA(p, d, q) model, where p represents the autoregressive term, d represents the integrated term, and q represents the moving average term. In our case, we used an order of (1,0,1), which means we used one autoregressive term and one moving average term, but no integrated term.

SARIMAX Results

Dep. Variable:	temp	No. Observations:	405158
Model:	ARIMA(1, 0, 1)	Log Likelihood	-959490.231
Date:	Thu, 20 Apr 2023	AIC	1918988.463
Time:	00:29:31	BIC	1919032.111
Sample:	0	HQIC	1919000.928
	- 405158		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
const	22.4543	0.003	7560.740	0.000	22.448	22.460
ar.L1	0.0614	0.006	10.626	0.000	0.050	0.073
ma.L1	-0.3597	0.005	-73.894	0.000	-0.369	-0.350
sigma2	6.6758	0.012	535.598	0.000	6.651	6.700

Ljung-Box (L1) (Q):	18.54	Jarque-Bera (JB):	58135.88
Prob(Q):	0.00	Prob(JB):	0.00
Heteroskedasticity (H):	1.65	Skew:	0.61
Prob(H) (two-sided):	0.00	Kurtosis:	4.40

After training the ARIMA model on the dataset, we used it to forecast the next value in the time series. We then appended the forecast to the original Spark DataFrame, converted it back to a Pandas DataFrame, and published the forecasted value to an MQTT queue.

Finally, we set up a loop to continuously perform the ARIMA analysis and append the forecasted values to the original DataFrame every three minutes. This allowed us to continuously update the dataset with new values and generate real-time forecasts using ARIMA.

OUTLIER GENERATION:

The code generates outliers for a given set of temperature data by calculating the mean and standard deviation of the data and then randomly generating lower and higher outliers based on this distribution. The generated outliers are published to a designated MQTT topic using the Paho MQTT client library. The code continuously runs in an infinite loop, generating new outliers every 3 minutes and publishing them to the designated MQTT topic.

RANDOM NUMBER:

In this section, we are using the Paho MQTT library to subscribe to three different topics: forecast/arima, forecast/bollinger, and forecast/outlier. We store the received values in a dictionary called values and use a threading.Event object called process_event to synchronize the processing. Whenever a new message is received, the on_message callback function is called, which stores the received value in the values dictionary and checks if all three values have been received. Once all three values have been received, the process_event object is set, allowing the processing to proceed.

In the main loop, we wait for the process_event object to be set, indicating that all three values have been received. We then process the received values by selecting a random number from the list of values based on their probabilities. Finally, we publish this random number to the forecast/random topic using the Paho MQTT library.

Overall, this code is feeding in data from three different topics related to exponential moving averages, autoregressive integrated moving averages, and outlier generation, and is using a probabilistic approach to select a random value from this data to publish to another MQTT topic.

MQTT QUEUE:

We have a Python code for MQTT (Message Queuing Telemetry Transport) client that uses the Paho MQTT library. Our code consists of a publisher and a subscriber. As a publisher, we publish messages to the MQTT broker on a specified topic with a payload. As a subscriber, we subscribe to a topic and wait for any messages published on that topic. When a message is received, it triggers the callback function specified in the code. We have subscribed to four different topics, and each topic has its own callback function.

The callback functions in our subscriber code print the received messages and their respective topics. We use a while True loop to keep the MQTT client running indefinitely. To prevent the code from consuming too much processing power, we use the time.sleep() function to add a delay between message publications.

KMEANS:

Here, we have included several modules from the PySpark machine learning library to train a K-Means clustering model on a given dataset. The dataset used is 'iot_telemetry_data.csv', which contains temperature readings. The model is used to predict if a new temperature value is normal or abnormal. The code includes a function to train the K-Means model, a function to use the trained model to predict if a temperature reading is normal or abnormal, and a main function that connects to a message broker and receives random temperature values to be passed to the K-Means model for prediction. Finally, the predicted category ('Normal' or 'Abnormal') is sent to store in a database.

REACTJS:

We used ReactJS to build a server that connects to a PostgreSQL database using the Node.js library, pg. We defined a number of endpoints using the Express framework, which enables the server to listen for incoming requests and respond with the appropriate data. The endpoints include retrieving all data from the "abnormal_table", deleting a specific entry from the table and adding it to "abnormal_history", and retrieving all data from the "abnormal_history" table. The code uses pool.query to perform queries on the database and handle multiple client connections at the same time.

References:

https://www.investopedia.com/terms/e/ema.asp
https://towardsdatascience.com/an-introduction-to-time-series-forecasting-with-arima-in-python-21394f45c3a0
https://www.tutorialspoint.com/mqtt/index.htm
https://en.wikipedia.org/wiki/MQTT
https://docker-curriculum.com/
https://www.tutorialspoint.com/docker/index.htm
https://www.youtube.com/channel/UC7yfnfvEUlXUIfm8rGLwZdA