An End to End Data Collection Architecture For IoT Devices in Smart Cities

Murat Kuzlu
Electrical Engineering Technology
Old Dominion University
Norfolk, VA, USA
mkuzlu@odu.edu

Halil Kalkavan
Department of Computer Science
Old Dominion University
Norfolk, VA, USA
hkalkavan@odu.edu

Oezguer Gueler
eKare, Inc.
Fairfax, VA, USA
oguler@ekareinc.com

Nasibeh Zohrabi
Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA, USA
zohrabin@vcu.edu

Patrick J. Martin
Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA, USA
martinp@vcu.edu

Sherif Abdelwahed
Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA, USA
sabdelwahed@vcu.edu

***Abstract*—A smart city environment utilizes different types of Internet of Things (IoT) devices, i.e., sensors and actuators, to collect and analyze millions of data to manage assets, resources, and services efficiently to improve the quality of life. However, the data collection process is one of the most challenging tasks for smart cities. To meet the requirements of smart city applications, a robust and efficient data collection process is needed to collect data from IoT devices then deliver it to data centers. This paper proposes an end-to-end data collection architecture from IoT devices for different smart city applications, such as smart buildings, transportation, water management, healthcare, and others, through advanced communication technologies and protocols and database management approaches. The proposed architecture includes five main components, (1) IoT Networks, (2) Server-side MQTT Interfacing, (3) Server-side Streaming, (4) NoSQL Database, and (5) Web Server. Finally, this paper provides a case study to demonstrate the effectiveness of the proposed architecture and discuss each design component in detail.**

***Index Terms*—Internet of Things (IoT), Smart cities, Data collection and streaming**

I. Introduction

Advancements in data communication and information technologies fostered smart devices connected things, called the Internet of Things (IoT), having embedded electronics and software. It is estimated that the total number of connected IoT (Internet of Things) devices will be more than 50 billion by 2022 [1]. The increase of IoT devices also promotes smart city application development, aiming to improve the quality of life in big cities by providing efficient and reliable services. With the availability of IoT technologies, cities can enable many smart applications easily and effectively for citizens. These applications include smart parking [2], smart lighting [3], waste management [4], traffic congestion [5], smart buildings [6], smart grid [7], and many more. A smart city needs a robust and efficient data exchange platform to contact the different systems, applications, types of data sources simultaneously. The data collection process from IoT devices is one of the biggest challenges in the smart city environment.

During the last decade, different smart applications along with the data collection solutions are discussed. Authors in [8] discuss possible solutions of data collection for smart transportation in the smart city environment. It indicates that the data collection process is complicated due to the high number of devices and volume of the data. Therefore, it needs a common standard interface and data models to store and analyze the collected data. Authors in [9] propose an open standard interoperable smart city platform, which allows sharing resources and creates a smart city data hub. It also demonstrates the standardized data collection and sharing for bus and vehicle information use cases. The study [10] presents a probability-based model addressing the data leakage and big data security problems in smart cities implemented in C++ in a simulated environment. In [11], a 5G-based smart city platform is proposed for data sharing, which supports an easy usage of the collected data and detects possible abnormal situations. Authors in [12] propose a data collection scheme to improve the collaborative mobile sinks in smart cities. Simulation results indicated that the proposed data scheme provides better performance than its peer in terms of data collection latency and the ratio of successful IoT object coverage. The proposed scheme can reach 96 percent of IoT devices while its peer can reach only 43 percent for the same collection duration. The study in [13] investigates the multi-source heterogeneous data collection approaches to be integrated with the smart city public information platform.

This paper presents an end-to-end data collection architecture for IoT devices in smart cities. The proposed architecture will cover all steps from data collection to data visualization. It is developed and designed based on the most up-to-date technologies and consists of five key components, namely, (1) IoT networks, (2) Server-side MQTT interfacing, (3) Server-side streaming, (4) NoSQL database, and (5) Web server. The

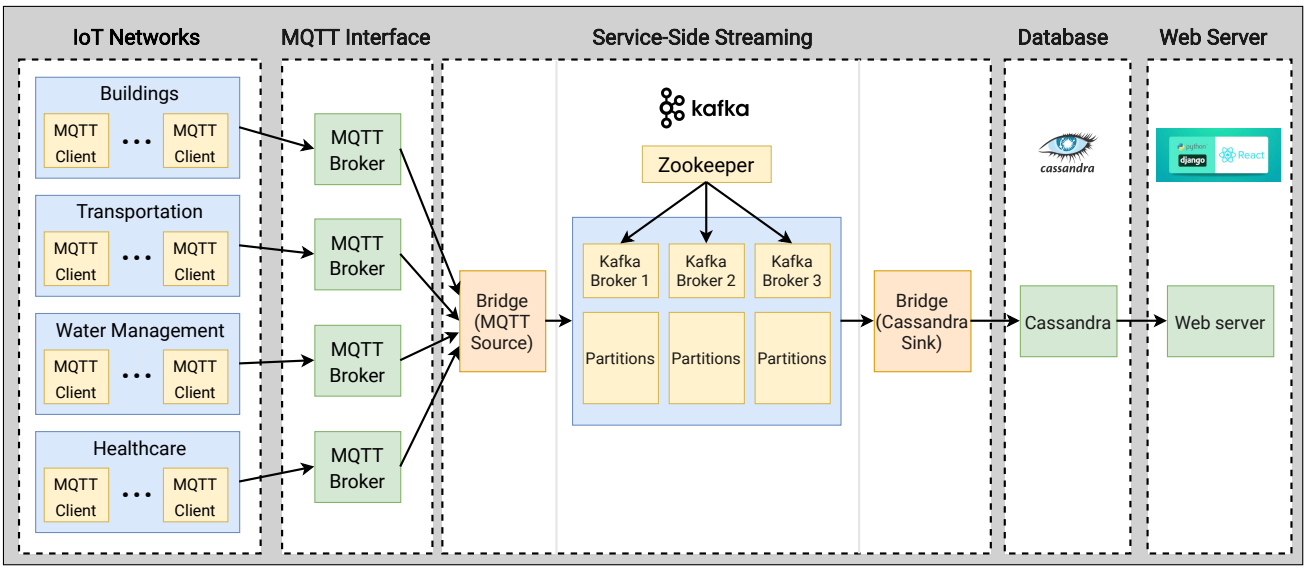

Fig. 1. End to end data collection process steps with main components

proposed end-to-end data collection architecture addresses the interoperability, scalability, and ease of use issues faced by conventional data collection architectures. Features and design considerations of each component of the proposed architecture are described in more detail as we move forward to the next sections of the paper.

The rest of the paper is organized as follows. Section II explains the main components of the proposed end to end data collection architecture. Section III presents a case study to demonstrate the effectiveness of the proposed architecture. Finally, the concluding remarks are given in Section IV.

II. An End to End Data Collection Architecture for Smart Cities

The proposed end-to-end data collection architecture for IoT devices in smart cities is illustrated in Fig. 1. The architecture covers five main parts, namely (1) IoT devices, (2) Server-side MQTT Interfacing, (3) Server-side Streaming, (4) NoSQL database, and (5) Web server. This section will discuss each component in further detail.

A. IoT Networks

Distributed, robust data collection is a key aspect to enable high level applications that incorporate IoT devices. In the proposed architecture, IoT devices, or *nodes*, connect to a communication network that supports the TCP/IP protocol. Nodes use the Message Queuing Telemetry Transport (MQTT) [14], a lightweight, standard application protocol, to send and receive messages across the smart city. MQTT uses brokers to maintain lists of active topics and provide that information to MQTT clients that wish to publish or subscribe to these topics. By using MQTT, we logically partition the various IoT networks based on their application domain, as shown in Fig. 1. Each domain defines a hierarchical topic structure that provides output (data, device status, etc.) and input (configuration, control, etc.) interfaces that can send or receive relevant messages. The payloads of these messages are formally structured using Protocol Buffers [15] to ensure data is structured correctly for subscribed devices and provide flexibility for device- and server-side algorithm implementation.

As a concrete example, we consider a building IoT network. All entities in this logical network connect to the server-side MQTT brokers using their MQTT client interface. The building topics are organized in a hierarchy like folders and files on a computer using the forward-slash ("/") as a delimiter. The structure of these topic names conveys meta-data about the device, such as its spatial location or its unique identifier. For example, if we consider an edge computing node that is deployed in a residential building with temperature, humidity, and CO2 sensors, the topic that provides the environmental data might be listed as:

```
/23220/B_001/F_004/7f8c19ec/environment
```

The top level of this topic structure is a US postal code, followed by a building identifier, `B_001`, and specific floor of that building, `F_004`. On this floor, there will likely be dozens of edge nodes equipped with sensors, so each must have a unique identifier, such as `7f8c19ec`. Finally, the `environment` topic indicates the topic output type. Accordingly, the `Environment` protocol buffer message would be defined to carry the temperature, humidity, and CO2 values as well as a sampling timestamp.

B. Server-Side MQTT Interfacing

As mentioned in the prior section, MQTT brokers must be deployed to queue received messages from publishers and transmit the messages to subscribers. To handle the multitude of MQTT clients in the smart city, the server-side MQTT interface requires horizontally scaled MQTT Brokers to manage the large number of topics as well as provide network resiliency.

This clustering mechanism may be provided by one of several available MQTT brokers, e.g., EMQX [16], VerneMQ [17].

MQTT Brokers support several security options, such as open, basic, or Transport Layer Security (TLS). We use TLS security and OpenSSL to create server keys and certification. For MQTT Clients, the authentication will be fully functional TLS with x.509 certificates. MQTT brokers are highly configurable, such as setting keep-alive message frequency or server certificates.

C. Server-side Streaming

Streaming is the core of the proposed end-to-end data collection architecture. Apache Kafka [18] is selected for the server-side streaming in the developed architecture, which is an open-source distributed event streaming platform. Apache Kafka is a publish-subscribe messaging system, which allows sending messages between processes, applications, and servers. It provides many advantages in terms of scalability, reliability, availability for messaging, storage, data integration, and stream processing. Apache Kafka architecture and interrelations between its components are comprehensively explained in [18]. The architectural components, which consist of Brokers, ZooKeepers, Producers, Consumers, Topics, Partitions, and Topic Replication Factor, are briefly explained here:

- *Kafka Brokers:* A Kafka broker is the primary server in a Kafka cluster. Typically, multiple brokers utilizing Apache ZooKeeper work together to create the Kafka cluster (three brokers are used in the developed architecture).
- *Apache ZooKeeper:* Kafka brokers use Apache ZooKeeper to manage and coordinate a Kafka cluster.
- *Kafka Producers:* A Kafka producer is the data source in Apache Kafka.
- *Kafka Consumers:* A Kafka consumer is responsible for reading data based on the topics and its subscription.
- *Consumer Group:* A Kafka consumer group is a group of consumers consuming one topic.
- *Kafka Topics:* A Kafka topic defines a logical category to which the data is stored. Any record has to be stored in a topic.
- *Kafka Partitions:* Partitions are the primary concurrency mechanism in Kafka. A topic is divided into partitions in a Kafka cluster, and those partitions are replicated through brokers.
- *Topic Replication Factor:* Topic replication is an essential factor to achieve a robust and reliable Kafka deployment. The replication factor used in the developed architecture is three.

Apache Kafka uses an immutable commit log with a very simplistic data structure, providing a persistent ordered data structure. A record cannot be directly deleted or modified, only appended onto the log after creating it. Each record is associated with a unique sequential ID known, i.e., offset, to retrieve data. A cluster architecture is used to be able to create and update a partitioned commit log for each topic, and all messages in the same partition are stored in the order based on arriving. Kafka's producer is responsible for writing to topics, while Kafka's consumer is reading from topics. Each record written by producers is added to the end of the topic commit logs. Then each record can be read by consumers based on the offset referring to specific locations in topic logs. A topic is distributed into one or more partitions to enable concurrency and provide high scalability and performance. The Kafka architecture can be customized based on the requirements by utilizing additional consumers as needed in a consumer group to access topic log partitions replicated across nodes.

```
import paho.mqtt.client as mqtt
from pykafka import KafkaClient
import time

mqtt_broker = "localhost"
mqtt_client = mqtt.Client("BridgeMQTT2Kafka")
mqtt_client.connect(mqtt_broker)

kafka_client = KafkaClient(hosts="localhost:9092")
kafka_topic = kafka_client.topics['temperature']
kafka_producer = kafka_topic.get_sync_producer()

def on_message(client, userdata, message):
    msg_payload = str(message.payload)
    print("Received MQTT message: ", msg_payload)
    kafka_producer.produce(msg_payload.encode('ascii'))
    print("KAFKA: Just published " + msg_payload + " to topic temperature")

mqtt_client.loop_start()
# subscribe to MQTT broker and get latest reading
mqtt_client.subscribe("temperature")
# if reading received trigger callback function which published data to Kafka
broker
mqtt_client.on_message = on_message
time.sleep(300)
mqtt_client.loop_end()
```

Fig. 2. MQTT-Kafka bridge is developed in Python

Kafka also offers several APIs, such as the Producer API, Consumer API, Streams API, and Connector API, to make deployments effortless and effective. In the developed architecture, three different frameworks, namely MQTT, Kafka and Cassandra, need to be linked to each other. These links called bridges in software development terminology are implemented in Python. Fig. 2 and Fig. 3 present the current implementation of the MQTT-Kafka and Kafka-Cassandra bridge.

MQTT and Kafka serve different purposes. MQTT connects hardware, such as a temperature sensor to a publisher-subscriber server to collect and distribute data, whereas Kafka provides a highly scalable and reliable publisher-subscriber framework. The Kafka connects the data from MQTT brokers/clients to a persistent data storage system, namely Cassandra. Kafka is an ideal candidate for scenarios that require high-performance, scalable data pipelines or data integration across

```
import time
from kafka import KafkaConsumer
from cassandra.cluster import Cluster
import json

class KafkaCassandraConnect:

   def __init__(self, kafka_ip, kafka_client_id, kafka_topic, cassandra_ip,
cassandra_dataCenter, cassandra_keySpace):

      self.kafka_consumer = KafkaConsumer(kafka_topic,
         bootstrap_servers=kafka_ip, group_id='group_01')
      self.cassandra_cluster = Cluster([cassandra_ip])
      self.cassandra_session =
self.cassandra_cluster.connect(cassandra_keySpace)

   def insert_to_devices(session, device_id, date_id, temperature1, value1):

      timestamp = int(time.time())
      session.execute(
      """
      INSERT INTO devices (id, device_id, date_id, temperature1, value1)
      VALUES (%s, %s, %s)
      """,
      (timestamp, device_id, date_id, temperature1, value1)
      )
      print("record added successfully to Cassandra")

   def run(self):
      for msg in self.kafka_consumer:
         print("message recieved from Kafka: " + msg.value.decode("utf-8"))
         obj = json.loads(msg.value.decode("utf-8"))

         self.insert_to_devices(self.cassandra_session, obj["device_id"], ,
obj["date_id"], obj["temperature1"], obj["value1"])
```

Fig. 3. Kafka-Cassandra bridge is developed in Python

multiple systems. Similarly, the MQTT protocol is utilized to collect data from IoT devices. For IoT use cases, especially for smart city applications where data is collected using sensors, a MQTT-Kafka-Cassandra pipeline is an adequate solution. IoT devices can connect to the MQTT broker via the MQTT protocol. MQTT broker receives and processes messages from a large number of IoT devices, and Kafka stores collected data and sends them to related entities for processing messages.

There are four different ways to connect MQTT and Kafka: (1) Build an MQTT bridge between MQTT Broker and Kafka, (2) Connect to Kafka via MQTT proxy, (3) Connect MQTT Broker to Kafka via Kafka Connect, and (4) Connect MQTT Broker to Kafka via MQTT Broker extension [19]. We chose the first option, i.e., build an MQTT bridge between MQTT Broker and Kafka. The bridge-based solution is also preferred as an alternative to using Kafka Connect and MQTT Broker extension needing additional installation and configuration, e.g., Confluent Platform Kafka installation. In the bridge-based solution, a custom application is developed in Python as a bridge between the MQTT and Kafka broker. It uses MQTT and Kafka client libraries to connect to the MQTT and Kafka brokers. The custom bridge is developed to forward the message to the Kafka broker.

Before building a bridge between the MQTT and Kafka broker in Python, which consumes MQTT messages and forwards them to Kafka, first, we set up fake IoT devices with Python for test purposes, as shown in Fig 4. In this experiment, the IoT device sends the room temperature messages to the MQTT broker.

```
import paho.mqtt.client as mqtt
from random import uniform
import time

mqtt_broker = "localhost"
mqtt_client = mqtt.Client("Temperature_Inside")
mqtt_client.connect(mqtt_broker)
mqtt_topic = “temperature”

while True:
   randNumber = uniform(70.0, 95.0)
   mqtt_client.publish(mqtt_topic, randNumber)
   print("MQTT: Just published " + str(randNumber) +
  " to topic temperature")
   time.sleep(3)
```

Fig. 4. Simulate random room temperature data published to MQTT broker

In the second step, MQTT-Kafka-bridge is developed in Python, as shown in Fig. 2. It consists of the following steps:

- Define MQTT client - BridgeMQTT2Kafka - and connect the client to the MQTT broker on localhost.
- Set up a Kafka client on localhost and standard port 9092.
- Define the Kafka topic as “temperature”, and create a Kafka producer.
- Start loop to receive continuous data.
- Subscribe to MQTT broker and get the latest reading.
- Once the MQTT client receives a message from the MQTT broker, a callback function named “on_message” is executed.
- In the callback function, the received MQTT message is displayed to the terminal before the same message is produced to a Kafka topic.

In the last step, the Kafka-Cassandra bridge is developed in Python. The following code snippet, presented in Fig. 3, shows the functionality.

- First, import relevant modules.
- Initialize a custom KafkaCassandraConnect object by defining the Kafka consumer, the Kafka topic, server IP,

and group ID. The initialization is concluded by defining the cluster and session.
- The Kafka consumer is run in a loop until the messages have been received.
- Each message is decoded into ”utf-8” and a JSON object is created.
- In the last step, the data is inserted into Cassandra utilizing a Cassandra “insert” query.

D. NoSQL Database

The persistent data storage part of the proposed architecture is the distributed database. In the developed architecture, time-series data collected from IoT devices is stored in Cassandra as it is a distributed database that allows data replication. Cassandra is an open-source NoSQL distributed database system suited for high frequency and high quantity data. Cassandra's main feature is reliability, i.e., storing data on multiple nodes with no single point of failure [20]. It consists of the following components: (1) *Node*: The basic component of Cassandra where data is stored, (2) *Data Center*: A collection of nodes, (3) *Cluster*: The collection of many data centers, (4) *Commit Log*: The place every write operation is logged, (5) *Mem-table*: Temporary table where data is hold after data is written in the commit log, (6) *SSTable*: Data is flushed to an SSTable disk file once Mem-table reaches a certain size.

Cassandra stores data in tables where each table is organized in rows and columns similar to other databases. Tables are grouped in keyspaces. Each table has a defined primary key, which is divided into partition keys and clustering columns. Data can be efficiently retrieved by providing a range of clustering columns since they are retrieved sequentially from the disk. Each entry (or row) in a table must contain a partition key whose Murmur3 Hash is used to assign it to a particular node in the cluster. Cassandra uses the partition key to index the data. All rows that share a common partition key make a single data partition, the basic unit of data partitioning, storage, and retrieval in Cassandra.

The code example, given in Fig. 5, shows the Cassandra Query Language (CQL) statement to create the TempSensor (temperature sensor) table. The table consists of several columns with different data types. The primary key is formed by combining the device_id, date_id, and the time columns, out of which the device_id and date_id form the partition key, and the time forms the clustering column. The parenthesis around device_id and date_id groups them together to form the first element of the primary key, and hence it represents the partition key.

```
>> CREATE TABLE tempsensor (
device_id text,
date_id text,
time timestamp,
temp double,
PRIMARY KEY ((device_id, date_id), time));
```

Fig. 5. Create the TempSensor (temperature sensor) table

For a given device_id and given date_id, all rows belong to the same partition (node), and data is stored sequentially based on the clustering key, which is the time. Fig. 6 illustrates that each row has a unique set of primary keys composed of the device_id, date_id, and time columns. However, all rows shown belong to the same partition because they have the same partition key (device_id and date_id) and are stored as a wide row on a single node.

```
 device_id | date_id    | time                            | temp
-----------+------------+---------------------------------+------
     D0001 | 2021-03-31 | 2021-03-31 11:03:20.000000+0000 | 55.6
```

Fig. 6. Sample TempSensor table entry

E. Web Server

The web application or data visualization and control platform is the last part of the proposed architecture. The web application provides the visualization and user interface via a web browser in real-time. Collected data from IoT devices are stored as time-series data, i.e., Cassandra. The architecture also allows access to real-time as well as historical data. The web server fetches data through the Django API from the Cassandra database and sends the data to the frontend to visualize the data in dashboards using different charts. Django is a Python-based web server framework that allows rapid development and clean, pragmatic design. The web app utilizes React which is a Javascript library to build rapid user interfaces (UI). The web app integrates different React components, which fetch data from API requests from the Django API and displays data using Google Charts. In addition, Google Charts provides a library, which is compatible with ReactJS. The library can receive provided data, manipulate it, and visualize the data in various charts. The Google Charts library is predestined for web deployments. It is powerful, simple to use for easy development and integration within web platforms [21]. The collected data can be charted easily based on the selected chart object through the embedded libraries. The Google Charts renderer supports several chart types, such as Pie Charts, Column Charts, Multi-series Column Charts, Bar Charts, Multi-series Bar Charts, Line Charts, and Multi-series Line. All chart types use the DataTable class to sort, modify, and filter data and fetched directly from the Cassandra database.

III. Case Study and Demonstration

In the smart city, heterogenous multi-sources generate many data with different data formats, devices, systems, APIs, etc. The proposed architecture is designed and implemented using the most up-to-date protocols and technologies, i.e., IoT, MQTT, Kafka, Django, ReactJS, Google Charts, to collect, store and visualize from multi-sources. This section demonstrates the visualization of data collected from IoT devices in a simulated environment. The data is collected from virtual IoT devices with Python for demonstration purposes. In this experiment, two IoT devices are created to send different data values, i.e., temperature and humidity. The IoT devices

send data messages to the MQTT broker through the software bridge between the MQTT and Kafka broker written in Python. First, MQTT and Kafka clients are started after initializing the brokers. Then, the Kafka-Cassandra bridge is initialized by defining the Kafka consumer, the Kafka topic, server IP, and group ID. The Kafka consumer is responsible for reading data based on the topics and its subscription and runs in a loop until the messages have been received. The data decoded into "utf-8" and a JSON object is inserted into the Cassandra database utilizing a Cassandra "insert" query.

In the last step, the web application using Django fetches data from the Cassandra database and sends the data to the frontend to visualize the data in charts, implemented using Google Charts. The developed web interface supports storing and viewing historical data for analysis. Historical data can be viewed using the 'Charts' page. For example, the temperature and humidity data are available as charts on the 'Dashboard' page. Charts are placed in the center of the page, and the data points to choose from appear on the right in a widget. Below the data points widgets are the 'Auto Update' and 'Export Data' widgets. Clicking on 'Auto Update' enables live streaming, and data is retrieved and updated every minute. The charts also show a color-coded legend to identify the data points. 'Export Data' widget exports the time-series data in the time and value format for all the data points mentioned in the Data Points widget. Fig. 7 shows an sample chart for the temperature and humidity data in a day, i.e., 24-hour. Data is obtained from the Cassandra database using the database driver developed for the proposed architecture. This driver provides a secure connection to the Cassandra database fetching the necessary data using its simplified API.

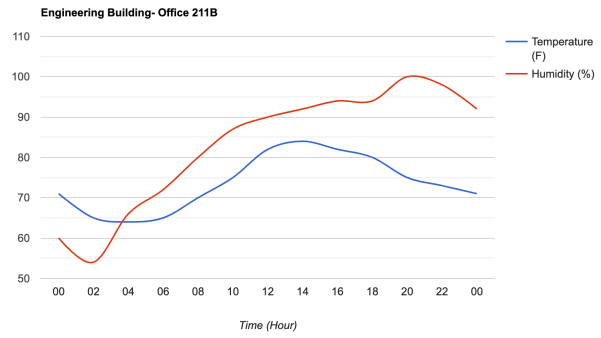

Fig. 7. A chart for temperature and humidity data in a day

IV. Conclusion

With the availability of IoT devices and advanced broadband communication technologies, cities become a digital ecosystem fed from multi-sources, i.e., systems, applications, data sources, etc. The data collected from those heterogeneous multi-sources is one of the biggest challenges due to the lack of common standard interfaces, data models, high volume of data collected. The proposed end-to-end data collection architecture for IoT devices in smart cities addresses the interoperability, scalability, and ease of use issues faced by conventional data collection architectures. It is developed based on the most up-to-date technologies and consists of several steps to meet requirements, namely, (1) IoT Networks, (2) Server-side MQTT Interfacing, (3) Server-side Streaming, (4) NoSQL Database, and (5) Web Server. Enabling the IoT devices hosted in each node is the first step of the data collection process. MQTT protocol is selected to collect data from IoT devices, widely used in the industry due to its simplicity and low bandwidth requirement for messaging. Each IoT device acts as an MQTT client. MQTT broker is the second step to queue the received messages from the publisher and the subscribers. The Mosquitto broker is selected to be used in the proposed architecture, including the broker/server itself and MQTT client libraries. Streaming is the most critical step in the data collection process. Apache Kafka, as a scalable, reliable, and elastic real-time platform for messaging, is selected to achieve the server-side streaming task. In the architecture, time-series data collected from IoT devices is stored in Cassandra. It is a distributed and an open-source NoSQL database system. The last step is the web server built on Django with React and various other application interfaces to provide a robust web platform for users. Web server using Django APIs fetches data from Cassandra and visualizes the collected data by using Google Charts.

This study demonstrates an end-to-end data collection architecture for IoT devices in smart cities using the most up-to-date technologies. As future work, the proposed data collection architecture will be evaluated in terms of performance, i.e., availability, latency, and scalability.

Acknowledgment

This work is supported by the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation, and workforce development in Virginia. For more information about CCI, visit cyberinitiative.org.

References

[1] "IoT Connections to Grow 140% to Hit 50 Billion by 2022," https://www.juniperresearch.com/press/iot-connections-to-grow-140pc-to-50-billion-2022, Accessed May 2021.

[2] I. Aydin, M. Karakose, and E. Karakose, "A navigation and reservation based smart parking platform using genetic optimization for smart cities," in *2017 5th International Istanbul Smart Grid and Cities Congress and Fair (ICSG)*. IEEE, 2017, pp. 120–124.

[3] E. Bingöl, M. Kuzlu, and M. Pipattanasompom, "A lora-based smart streetlighting system for smart cities," in *2019 7th international Istanbul smart grids and cities congress and fair (ICSG)*. IEEE, 2019, pp. 66–70.

[4] S. S. Chaudhari and V. Y. Bhole, "Solid waste collection as a service using iot-solution for smart cities," in *2018 International Conference on Smart City and Emerging Technology (ICSCET)*. IEEE, 2018, pp. 1–5.

[5] R. Goudar and H. Megha, "Next generation intelligent traffic management system and analysis for smart cities," in *2017 International Conference On Smart Technologies for Smart Nation (SmartTechCon)*. IEEE, 2017, pp. 999–1003.

[6] A. Nugur, M. Pipattanasomporn, M. Kuzlu, and S. Rahman, "Design and development of an iot gateway for smart building applications," *IEEE Internet of Things Journal*, 2018.

[7] M. Kuzlu, M. M. Rahman, M. Pipattanasomporn, and S. Rahman, "Internet-based communication platform for residential dr programmes," *IET Networks*, vol. 6, no. 2, pp. 25–31, 2017.

[8] S. N. Shukla and T. A. Champaneria, "Survey of various data collection ways for smart transportation domain of smart city," in *2017 international conference on i-smac (iot in social, mobile, analytics and cloud)(i-smac)*. IEEE, 2017, pp. 681–685.

[9] H. M. Nguyen, J. Byun, K. Kwon, J. Han, W. Yoon, N. Lee, H. Kim, N. Pham, D. Kim *et al.*, "Oliot-opencity: Open standard interoperable smart city platform," in *2018 IEEE International Smart Cities Conference (ISC2)*. IEEE, 2018, pp. 1–8.

[10] V. Dattana, K. Gupta, and A. Kush, "A probability based model for big data security in smart city," in *2019 4th MEC International Conference on Big Data and Smart City (ICBDSC)*. IEEE, 2019, pp. 1–6.

[11] J. Kim, S. Jang, D. Jee, E. Ko, S. H. Choi, and M. K. Han, "5G based smartcity convergence service platform for data sharing," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2020, pp. 1522–1524.

[12] F. Chehbour, Z. Doukha, S. Moussaoui, and M. Guerroumi, "Congestion aware data collection with mobile sinks in smart city," in *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, 2020, pp. 1–7.

[13] S. Lehner, J. Horstmann, and J. Schulz-Stellenfleth, "Terrasar-x for oceanography mission overview," in *IGARSS 2004. 2004 IEEE International Geoscience and Remote Sensing Symposium*, vol. 5. IEEE, 2004, pp. 3303–3306.

[14] "Message queuing telemetry transport," https://mqtt.org/mqtt-specification/, accessed May 2021.

[15] "Protocol buffers," https://developers.google.com/protocol-buffers, Accessed April 2021.

[16] "Emqx," https://www.emqx.io/products/broker, Accessed May 2021.

[17] "Vernemq," https://github.com/vernemq/vernemq, Accessed May 2021.

[18] "Apache Kafka Architecture: A Complete Guide," https://www.instaclustr.com/apache-kafka-architecture, Accessed May 2021.

[19] "MQTT and Kafka," https://medium.com/python-point/mqtt-and-kafka-8e470eff606b, Accessed May 2021.

[20] "Apache cassandra, open source nosql database," https://https://cassandra.apache.org/, Accessed May 2021.

[21] "What is google charts?" https://www.w3schools.com/whatis/whatis-google-charts.asp/, Accessed May 2021.