



# **MASTER THESIS**

**In Order to Obtain the**

## **RESEARCH MASTER**

**in**

**Information Systems and Data Intelligence (ISDI)**

**Presented and defended by:**

**Ibrahim Tawbe**

**On Friday, September 27, 2019**

### **Title**

**CityPro; An Event-driven Collaborative Business Process**  
(Edge Centric Middleware for City Surveillance Platform)

### **Supervisor**

**Dr. Mohamed Dbouk**

### **Reviewers**

**Dr. Rami Tawil**

**Dr. Yehya Taher**

# Acknowledgment

First and foremost, I would like to thank God, for the continuous stream of blessings.

I would like to express my deep and sincere gratitude to my research supervisor, Prof. Mohamed Dbouk for the whole experience starting from the first class and through the research process. In addition to the scientific knowledge, I learned from Prof. Dbouk valuable soft skills for communication, leadership, teamwork, and critical thinking.

I would also like to acknowledge Dr. Rami Tawil and Dr. Yehya Taher as the reviewers of this thesis, and I am gratefully indebted to them for their very valuable comments on this thesis.

Completing this work would have been more difficult were it not for the support and understanding of my parents, wife, and my two kids.

I would also like to thank my employer and work manager for granting me the necessary time to carry on my course work and the thesis research.

Special thanks for Mariam Hakim (2018 master graduate) for her valuable reviews and comments.

## Abstract

Smarter Cities provides a better management for city services, reporting problems, and predicting future issues. One critical and important aspect is city protection. Dbouk, M., Mcheick, H. and Sbeity, I. (2017) proposed CityPro a “city-surveillance collaborative platform” that integrates multiple existing operational systems to provide city protection. The paper laid down the architecture of such a system. Their research highlighted the concept of “connectors” between different city systems and a central “core” system. However, this concept needs deeper investigation for its design and how it operates. We interchange the connector concept with a middleware approach. The need of a robust and unified middleware platform in the CityPro platform or more generally in the Smart City is not yet a solved problem. Traditional middleware that evolved in enterprise environments can’t handle smart cities complexities. This research provides a new middleware (architecture and framework) that connect existing city systems to the core system trying to solve issues such as heterogenous systems, event processing, real time data, scalability, availability, and interoperability while exploring edge computing capabilities. We called it Edge Centric Middleware for City Surveillance. We also provide an implementation for this middleware to serve as an open-source generic framework that can be extended and customized.

# Table of Contents

Acknowledgment.....	2
Abstract .....	3
Table of Contents .....	4
Table of Figures .....	5
1. Introduction .....	6
1.1 Context & Motivation .....	6
1.2 Problem Position.....	7
1.3 Intended Solution .....	8
2. Survey of Technologies & Solutions.....	9
2.1 Edge and Fog Computing.....	9
2.2 Middleware in the Context of Cities & Surveillance.....	9
2.3 Accessing Distributed Heterogeneous Database Systems .....	12
2.4 Message Broker .....	14
2.5 Complex Event Processing (CEP) .....	14
3. Proposed Solution: Edge Centric Middleware .....	15
3.1 Towards an Edge-Centric Middleware .....	15
3.2 General Architecture .....	15
3.3 Edge Device Software Framework.....	17
3.4 Data Flow in the Edge Centric Middleware .....	18
3.5 Key Design & Implementation Decisions and Techniques .....	21
4. Proof of Concept (PoC) – Telecom Test Case.....	26
5 Conclusion & Future Work.....	28
References.....	30

## Table of Figures

Figure 1 - CityPro General Architecture [1] .....	7
Figure 2 Smart city: Multiple applications create big data [4] .....	9
Figure 3 Civitas Platform [7] .....	10
Figure 4 InterSCity Platform [8] .....	11
Figure 5 - MUSYOP Architecture .....	12
Figure 6- Apache Spark SQL.....	13
Figure 7 - Proposed Solution General Architecture .....	15
Figure 8 - Edge Device Inner Components .....	17
Figure 9 - Two-Tier Architecture .....	18
Figure 10 - Three-Tier Architecture .....	18
Figure 11 Live Data Flow .....	19
Figure 12 On-Demand Data Flow .....	20
Figure 13 - Avro File - Schema Contract .....	22
Figure 14 - Apache Spark Architecture.....	23
Figure 15 – Siddhi [19] .....	23
Figure 16 - Siddhi Code Snippet from CEP module .....	24
Figure 17 - Edge Framework Report Template .....	24
Figure 18 RaspberryPi 3 B+.....	27
Figure 19 City Core Simple UI .....	28

# 1. Introduction

---

## 1.1. Context & Motivation

The amount of challenges a city faces is increasing daily. The Smart City paradigm approach these challenges by using city's resources efficiently and optimize services and management such as traffic control, electricity, water supplies, waste control, recycling, and public safety. It's worth to note that tackling these issues using technological advancements is not for "modern" cities only. Cities in developed countries can and should take measures following Smart City paradigm to solve challenges. Smart City is not a one-shot solution, developed countries can gradually build up their Smart City eco-system.

Public safety is not considered a luxurious service, it's an essential challenge for any city. Cities are facing a variety of safety risks such as natural disasters, terrorists' attacks, crimes, vehicle accidents, etc. What triggers these risks is traditionally monitored by different government agencies for example weather monitoring agency is separated from the terror agency and local police forces. On the other hand, these risks put citizens in danger and coordinating the emergency procedures is critical to lower the losses. Furthermore, detecting and dealing with an emergency is not good enough, there is a need to predict and act before bad things happen. That's why they build and run models that simulate real world scenarios using machine learning and artificial intelligence. In addition to the models, early alarms sometime come from the correlation between different data sources.

Dbouk, M., Mcheick, H. and Sbeity, I. (2017) [1] proposed a collaborative platform for city protection, they called it "CityPro". CityPro is expected to collect data, analyze it, and support decision making for emergency cases. The approach combines and inter-operates pre-existing operational information systems such as security forces, airports, banks, telecom companies, hospitals... Its golden rule is: "To accumulate extreme benefits of existing operational systems with the least possible deployment efforts". Their research delivered the system architecture and highlights the potential of such a collaborative solution. In CityPro pre-existing operational systems are "data providers" that are connected to the CityPro Platform via "collaborative connectors". CityPro platform center is two parts: a federated big-data repository and a core system. The data repository is a federated star schema-like data model where it stores data coming from providers after applying summarization, integration, and enrichments. The core system hosts a computation environment for machine learning, monitoring, and complex event processing.

## 1.2. Problem Position

A Smart City is considered a heterogeneous and distributed information system. There is a need to collect, integrate, and correlate data from all these systems to deliver value. CityPro [1] tries to standardize the relation between the different systems with a centralized and supervised control and data repository architecture. It defines data providers as domain-specific independent (stand-alone) systems that coexist within the considered territory such as police departments, fire stations, banks, customs, and so on. In the context of CityPro these systems are data producers. The access to these data providers is through “collaboration-link” or a “connector”. CityPro [1] defines a connector as “Dedicated links that are materializing the collaborative inter-relationships between CityPro components. They mainly consist of ETL like dedicated data exchange automated protocols based on ‘adapter pattern’.” CityPro delivered the general architecture of the system, while we still need to investigate deeper in how to provide a standard and uniform access to these heterogeneous distributed systems with minimum effort at the data provider side.

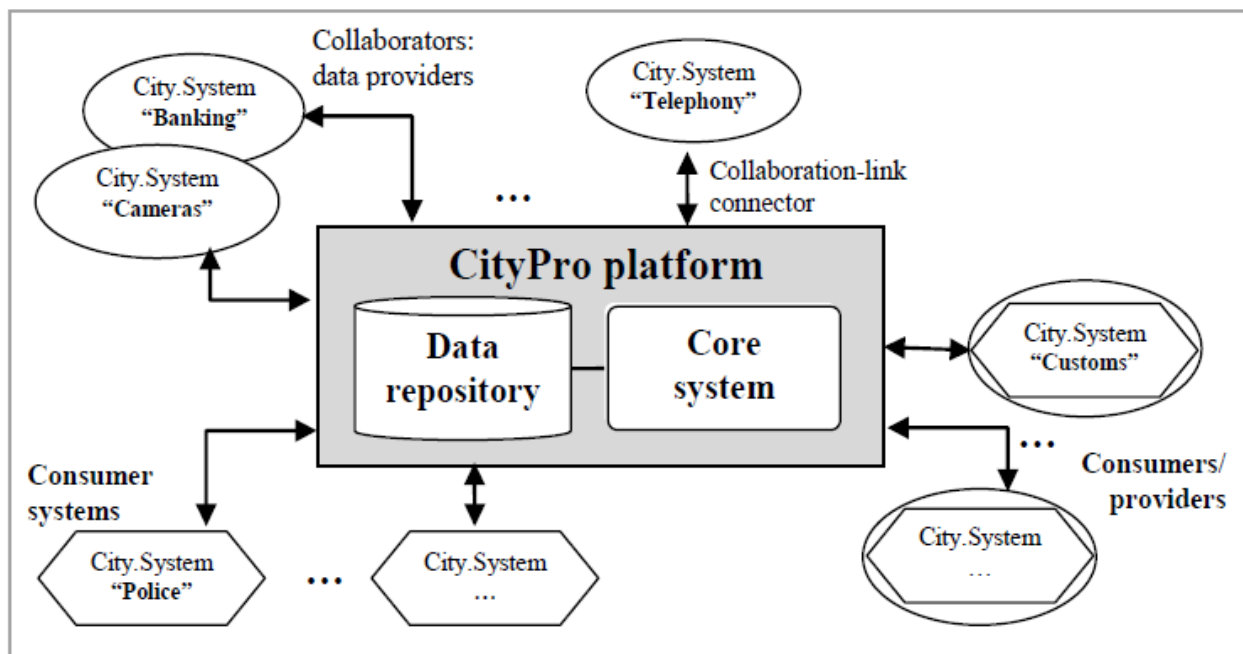


Figure 1 - CityPro General Architecture [1]

To address the need for city surveillance CityPro defines two data flows: periodically where data continually arrive from data providers and on-demand where the central system asks providers for instant detailed data. We need to define the standard protocols for this data exchange.

Such a highly distributed system produces gigantic data. A pure central system may struggle to handle all this data flow and deliver value, especially in the case of near real-time alerts. That's why it's tempting to use the distributed computation environment along the process of detecting anomalies and preparing the data for analysis.

Privacy and security are always an issue in any collaborative and network-based solution. In today's world the approach to this issue is a mix between political or government policies and technical implementations. At the technical side we should take these issues into consideration from early stages, starting from the design of the system.

### **1.3. Intended Solution**

We propose a new middleware approach that materialize the concept of connectors introduced in CityPro [1]. A connector middleware framework that:

- Provides a uniform and standard interface to the heterogenous distributed systems.
- Borrows some edge computing concepts to provide a computation resource for delegated tasks from the core central system to the distributed near-data-providers networks.
- Accompanied with well-defined and standard data flow and exchange protocols.
- Supports near real-time event processing and reporting. In addition to supporting the two modes of data exchange of CityPro: on-demand and periodically.
- Follows the concept of black box and security by obscurity to separate the framework private inner-workings, protocols, and communications from the data provider.
- Supports additional security layering by design and implementation such as memory and network encryption

We call this proposal "Edge-Centric Middleware"

This research delivers the architecture of the Edge Centric Middleware as well as an implementation. The implementation highlights and adopts highly rated open-source technologies that span different fields.



## 2. Survey of Technologies & Solutions

---

### 2.1. Edge and Fog Computing

W. Shi, J. Cao and Q. Zhang (2016) [2] refers to edge computing as the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services. Another technical trend is Fog Computing. This term is created by cisco and many interchanges it with edge computing. Cisco states that “Fog computing is a standard that defines how edge computing should work” [3].

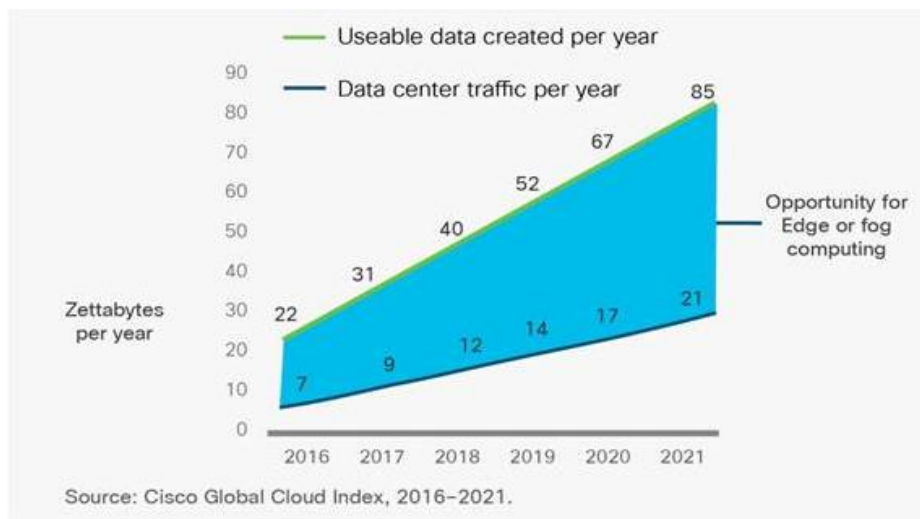


Figure 2 Smart city: Multiple applications create big data [4]

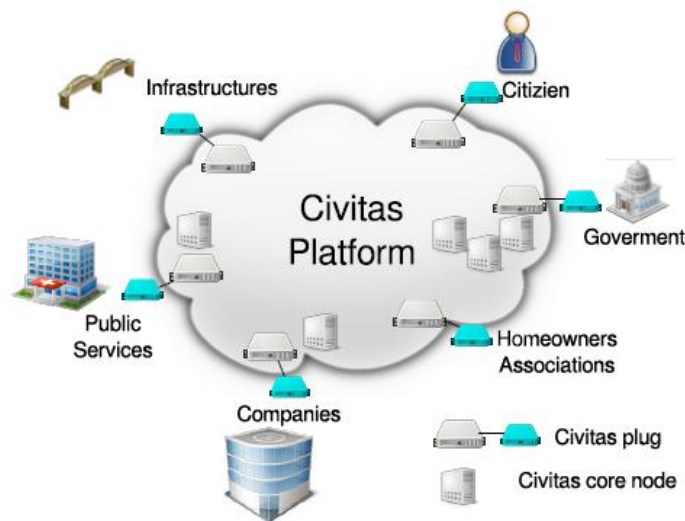
### 2.2. Middleware in the Context of Cities & Surveillance

Before edge computing and smart city era, “Embedded Middleware on Distributed Smart Cameras” [5] designed and implemented a middleware for distributed embedded image processing on a network of smart cameras. They embedded the middleware in the camera device. In our case CityPro, instead of the camera or device there is a complete information system – the data provider. Unlike the embedded middleware used in [5] where they interchange data between cameras, in CityPro data partners don’t interact with each other.

Furthermore, in CityPro an embedded middleware can't handle big-data, scalability, availability, and other processing tasks that's why we evolved the concept of embedded to the edge device.

A research by M. Fazio et al (2012) [6] tackled heterogeneity in terms of communication technologies and sensors devices by abstracting the sensing infrastructure with the collected data. They used the Sensor Web Enablement standard specifications which plays the role of discovering and managing sensors via the Web. Once again, the research focused on collecting data from sensors and IoT devices, unlike the case of CityPro with distributed data partners.

F. J. Villanueva et al (2013) [7] highlighted that traditional middleware technologies are designed for enterprise environments and can't address issues of heterogeneity and scalability in a Smart City. They connected different entities such as citizens, governmental institutions, and companies to the "Civitas" platform which is considered as the core of the IT infrastructure in the Smart City. The connection is through a device called "Civitas Plug", these devices can be smartphones, company servers, residential gateways...



*Figure 3 Civitas Platform [7]*

Another key point in the Civitas platform is the "Core Nodes". They are servers that host different kind of services to the city entities. They consider that heterogeneity and interoperability is solved by the distributed object-oriented middleware, where each entity is an object with a set of defined standard interfaces. It also supports event-based communication through publish-subscribe pattern.

We share the idea of the plug device, but this device can't alone solve the availability and scalability issue, that's why we added a broker layer between our "edge device" and the "core system". Moreover, we needed a deeper study of the inner-design and inner-workings of the plug device.

A. de M. Del Esposito et al (2017) [8] noted that there is no agreed middleware platform for Smart Cities platforms. They listed three factors for this challenge: security and privacy policies, the lack of scientific and practical validation, and the "extensive use of development of non-opensource software" which is causing inter-operability issues and limits the collaboration among researchers. They applied the micro-service paradigm to provide a modular and scalable middleware. InterSCity microservices architecture is shown in Figure 4. The abstraction is through "city resource" a logical concept that resembles a physical entity such as cars, traffic lights, etc.

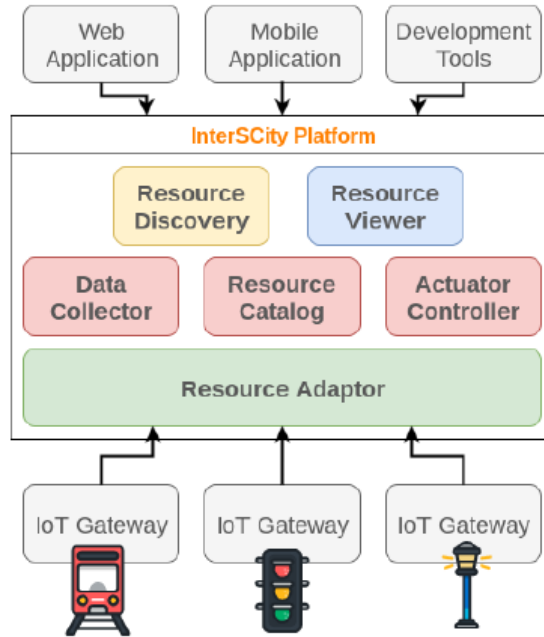


Figure 4 InterSCity Platform [8]

Each resource has attributes and functions to provide data and receive commands. For communication protocols, microservices use synchronous HTTP Rest API and asynchronous message bus using RabbitMQ. Their work adopts many open source projects such as PostgreSQL and Redis. In CityPro context the distributed existing information systems are more data producers that service providers. Furthermore, to some extent, middleware and micro-services solves different problems. A micro-services architecture takes the whole application (Smart City) and de-couple it into independent services.

## 2.3. Accessing Distributed Heterogeneous Database Systems

Today's computing and analysis techniques are tempting to integrate different Information Systems to provide additional insights. On the other hand, many international enterprises are providing services that span different subjects in different locations. In CityPro project which tries to maximize the benefit of the distributed operating systems, there is a clear case for this challenge.

L. Wang and R. Ranjan (2015) [9] highlights the need to “combine and analyze the distributed data along with contextual factors”. The authors list the current solutions and technologies in the cloud computing infrastructure stack (Azure, Amazon, private stacks) in three main domains: managing distributed clusters, distributed data processing models (such as MapReduce), and the data management service across datacenters which “integrated different cloud data storage services by providing a transparent interface” such as Simple Cloud API , PDC@KTH’s proxy service, Open Grid Services Architecture Data Access and Integration OGSA-DAI).

Z. Liu et al (2014) [10] provided “a federated approach - a mediator server - that allows users to query access to multiple heterogeneous data sources” relational database, Triplestore, NoSQL database, and XML with a management layer using SPARQL and mapping different databases to RDF.

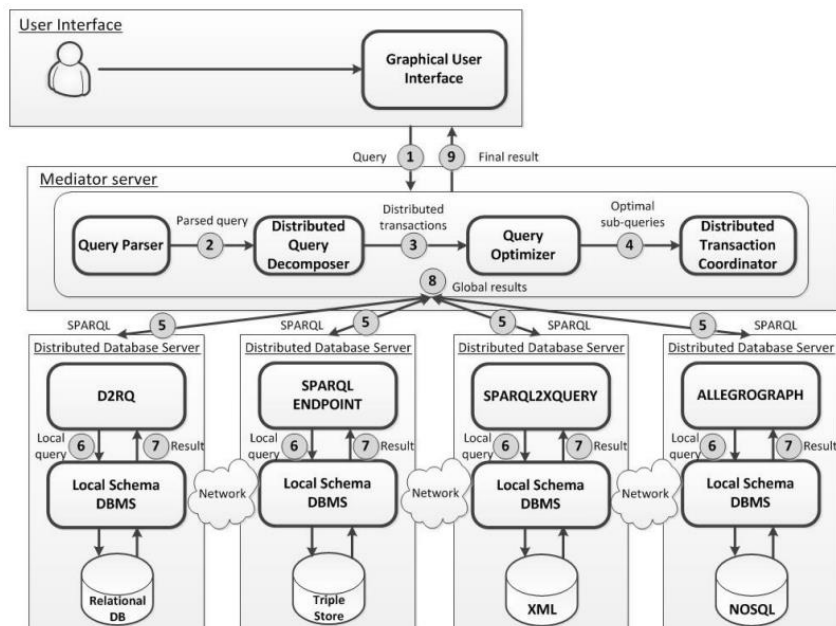
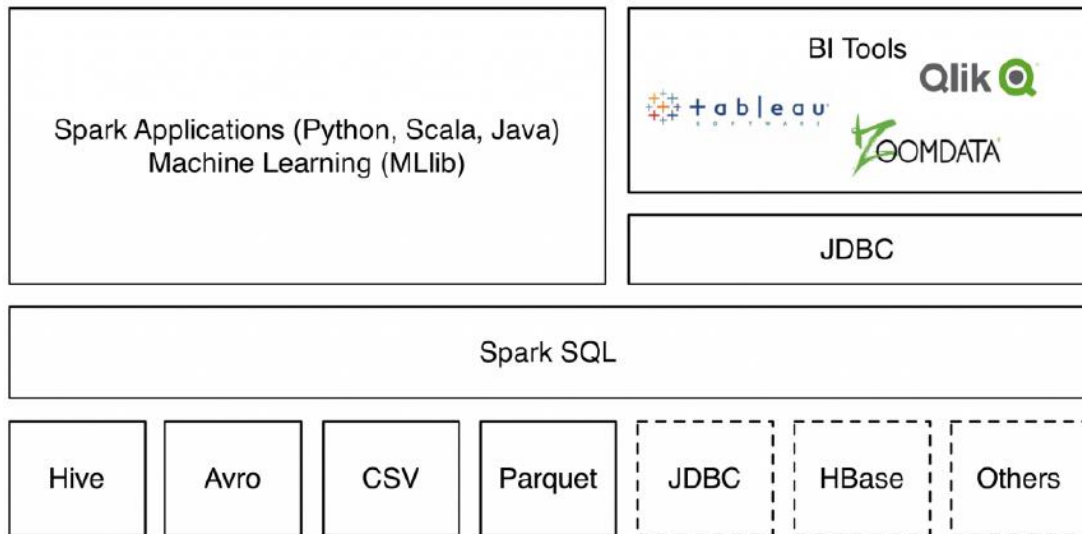


Figure 5 - MUSYOP Architecture

One more interesting solution for accessing different datasets is Apache Spark. Apache Spark is a “unified analytics engine for large-scale data processing” [11]. The unified part is baked into the Spark SQL module.



*Figure 6- Apache Spark SQL*

Spark SQL Layer is built on top of two interfaces Data Frame API and Data Source API which supports schema understanding, reading data with filters, and writing custom aggregations. Custom drivers for different database engines will use these interfaces to support those functionalities. Currently drivers for most database engines (MySQL, MongoDB, HBase, Cassandra, HDFS ...) are already implemented and ready for production use.

There are two important sides for this topic: distribution and heterogeneity. For the heterogeneity part there are two trends: use ontology-based solutions or build custom interface layer. It’s also important to note that whatever the integration solution. SQL is the preferred language to query these distributed datasets. The SQL layer is used for the unified access with the added benefit that it can easily integrate with upper layer tools and technologies such as BI tools. Often suggested solutions tackle the whole process from accessing the data to integrating it, but my focus is on providing the interface to different database systems and the integration part will be solved in later phases of CityPro.

## 2.4. Message Broker

A Message Oriented Middleware (MOM) is responsible for sending and receiving data encapsulated in messages between different distributed systems. A MOM can be with a broker or broker-less. TIBCO Inc. [12] defines a message broker as a discrete service that provides data marshaling, routing, persistence, and delivery to all appropriate consumers.

We will highlight different technologies and researches with respect to important features we are interested in persistent cache, high availability and fault tolerance, scalability, with added value features such as message formats optimizations for binary messages and compression. First we consider current production-grade technologies. The state of art in persistence is to use a journaling file system write-ahead commit log backed by operating system page cache, this is implemented in Apache Kafka [13] and Apache ActiveMQ Artemis [14] or delegate this to a database that uses this implementation. For high availability and fault tolerance, a replication set of 3 brokers is recommended for production. Scalability is done horizontally by adding more nodes as brokers, but a consensus and management service is needed to keep track of nodes and data index in the cluster. One of the well-known and heavily used software that implements this functionality is Apache Zookeeper [15] which itself can be replicated. Research wise, N.-L. Tran, S. Skhiri and E. Zimányi (2011) introduced “EQS: an Elastic and Scalable Message Queue for the Cloud” [16]. They discuss automatic scaling and load balancing in message queues to optimize the throughput along systems by layering additional components for monitoring, rules, and scaling management.

## 2.5. Complex Event Processing (CEP)

With various systems and sensors generating and sending data, there is a need to detect interesting patterns along the data streams. CEP paradigm has an opposing concept to regular databases. Instead of executing a query on a dataset, the data is executed on a well-defined query. This technology has been deployed and heavily used in the financial sector especially for fraud detection.

D. Luckham and B. Frasca [17] introduced the concept back in 1998. It was a hot topic again in the research community in 2006-2009 where it was discussed in the context of big-data and adding machine learning for prediction of events. Many commercial and open-source CEP systems are available such as Apache Flink [18], Siddhi.io [19], and Esper [20].

## 3. Proposed Solution: Edge Centric Middleware

---

### 3.1. Towards an Edge-Centric Middleware

This research is based on CityPro [1], a collaborative and inter-operable system architecture with a centralized supervised control. As stated in section 1.2, CityPro is connected to data providers via “connectors”.

Traditional middleware doesn’t fit to play the role of CityPro’s connectors. Additional criteria and functionalities are required as mentioned in section 1.2. Moreover, reviewing the traditional middleware in the edge computing era opens a door for more options and facilities. At the general phase we integrated two well-known concepts: edge device and a broker into one approach. This help us to utilize more features while decoupling the roles of each part.

Applying our work to CityPro, it provides the middleware between data providers such as (banks, customs, hospitals...) the existing information systems and the “Core” system of the city. Although we started by challenges raised by CityPro, we designed and implemented our approach to be generic, opensource, and customizable so it can be applicable in many scenarios.

### 3.2. General Architecture

There are two main parts for the Edge Centric Middleware: the edge device and the broker as shown in Figure 8.

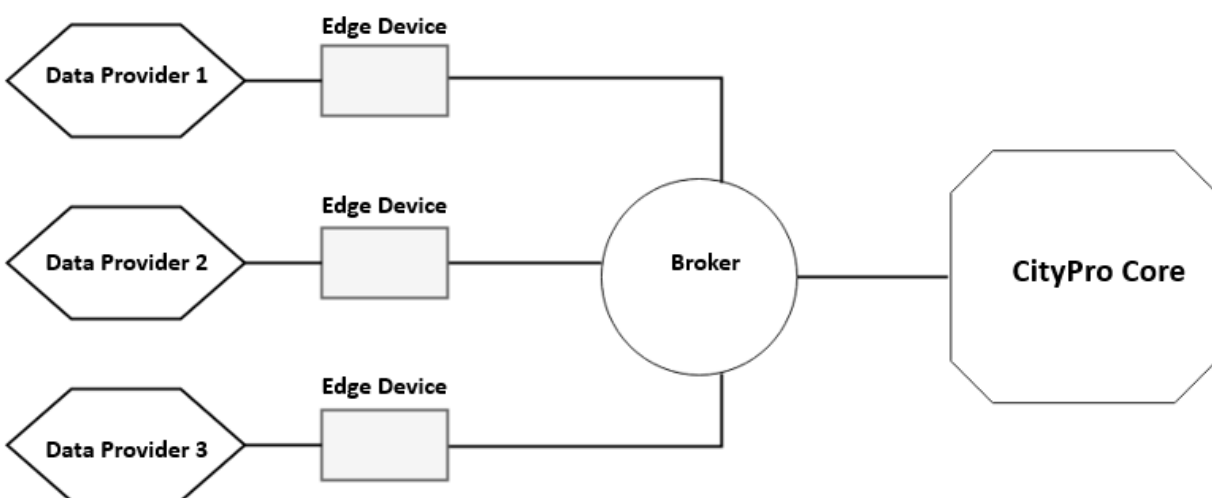


Figure 7 - Proposed Solution General Architecture

### 3.2.1. The Edge Device

It's a software and hardware package deployed at the edge network of the data provider and connected to the existing system via computer network communications. This package can scale up from a simple computer board such as Raspberry-Pi [21] to a rack of servers. The main roles of the edge device:

- Provide an interface to access different databases at rest at the provider side according to a schema contract required by the government agency
- Consume live data from the provider side according to a schema contract
- Provide the required computation resources to host and execute data summary and ETL-like operations
- Send batches of data according to the configured time interval and schema contract
- Detect and propagate real-time alerts specified by a defined list of triggers
- Enabler for the confidentiality and integrity of the data and business rules in question

In the next sections, we will propose the key design principals and the inner components of the edge device that enable it to fulfill the above roles.

### 3.2.2. The Broker

It's the second part, between the network of distributed edge devices and the core system. Only our certified edge devices can connect to the broker, this enhance the privacy and quality of data flowing through the middleware. From the broker point of view edge devices are data producers and the "Core System" is a data consumer. The broker roles are:

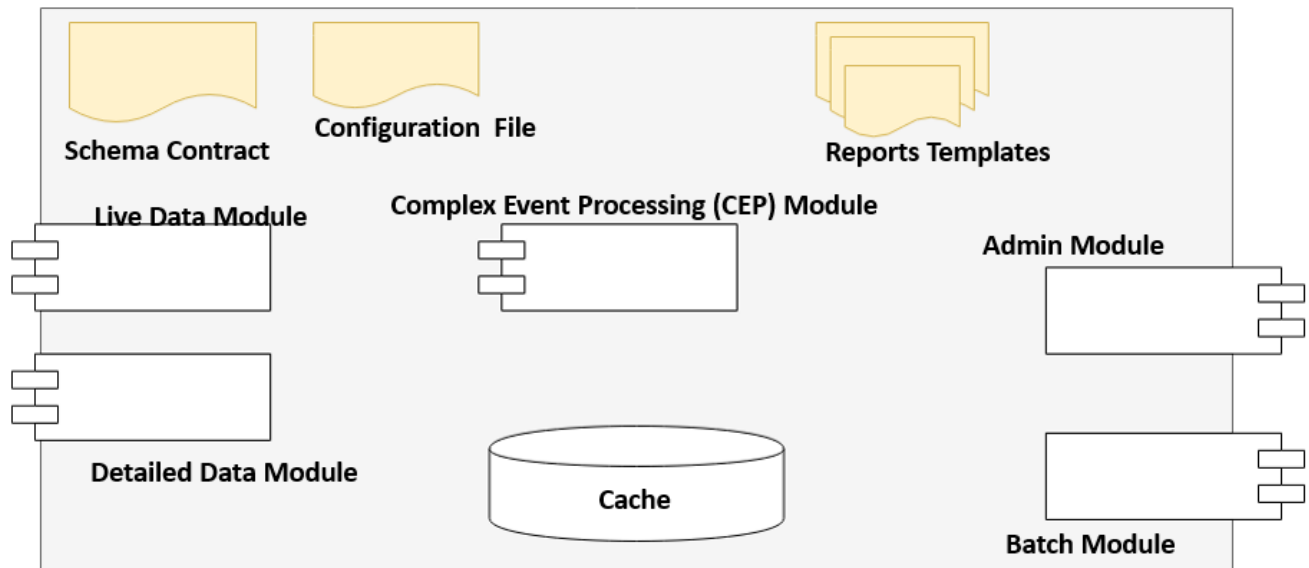
- A message queue that supports high volume and speed of data
- Support publish-subscribe pattern
- Support high availability in case of systems failures and high traffic
- Support horizontal scalability to handle existing and new systems

While we invested more on the edge device part to propose a new framework, we opted to rely on existing technologies for the broker side. In addition to realizing the above criteria, where traditional message queues support producer and consumer, some brokers support also a logic-processing endpoint such as Apache Kafka's [13] Processing API which can be used for data integration at this stage before consuming the data.



### 3.3. Edge Device Software Framework

The edge device framework is a software & hardware package. In this section we will describe the software stack of the edge device. We decomposed the framework into subcomponents with decoupled functionalities. We used a file-based configuration for some settings and standard communication protocols for inter-component communications and for the outer interfaces whether with the data provider or the core system.



*Figure 8 - Edge Device Inner Components*

- **Schema Contract:** States the required (selected) fields and fields' types from the data provider.
- **Live Data Module:** Consumes live data from the provider and validates the raw data according to the schema contract.
- **Complex Event Processing Module:** Works on a stream of data and filter events that match the required query. The query uses standard language SQL. Any matching result should be sent to the broker immediately.
- **Detailed Data Module:** Provides an interface to query heterogeneous databases and files. It also generates dynamic reports of the results using the reports templates. This module provides a unified standard query interface language SQL.
- **Cache:** Stores temp data between batch intervals. It's optimized for high-performance sequential operations.
- **Batch Module:** Queries the cache according to the defined time interval and sends them to the broker.
- **Admin Module:** Receives commands and direct queries from the core system and replies with the result.

### 3.4. Data Flow in the Edge Centric Middleware

Edge Centric Middleware supports three data flow modes:

1. Data is collected from the provider, prepared, and then sent in batches to the core system
2. A defined event pattern can trigger sending data near real-time to the core system
3. The core system requests detailed data on a specific subject. The request is fulfilled by the edge device which sends back a reply message

#### 3.4.1. Provider Live Data Source

The first two modes operate while getting live data from the provider. To achieve this, we studied two paradigms at the abstract level.

1. First Case: Two-Tier Systems:

In this case: We consider the database as the source of “live” data. So, we must detect and forward any data changes at the database level and forward the changes to a live data stream.

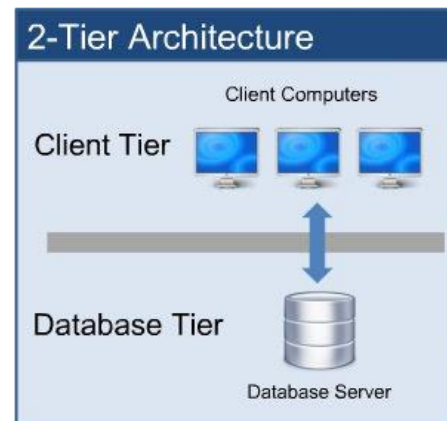


Figure 9 - Two-Tier Architecture

2. Second Case: Three-Tier Systems:

In 3-Tier architecture we can stream data live from the business logic layer or we can use the database layer in a similar way to the first case

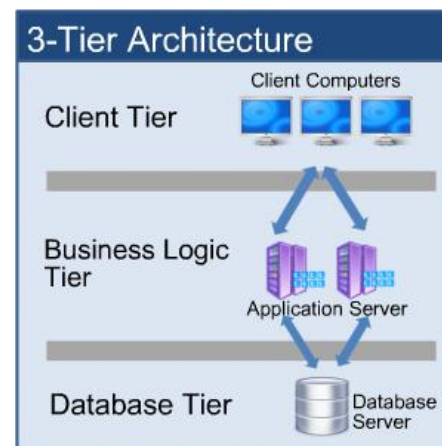


Figure 10 - Three-Tier Architecture

### 3.4.2. Live Dataflow

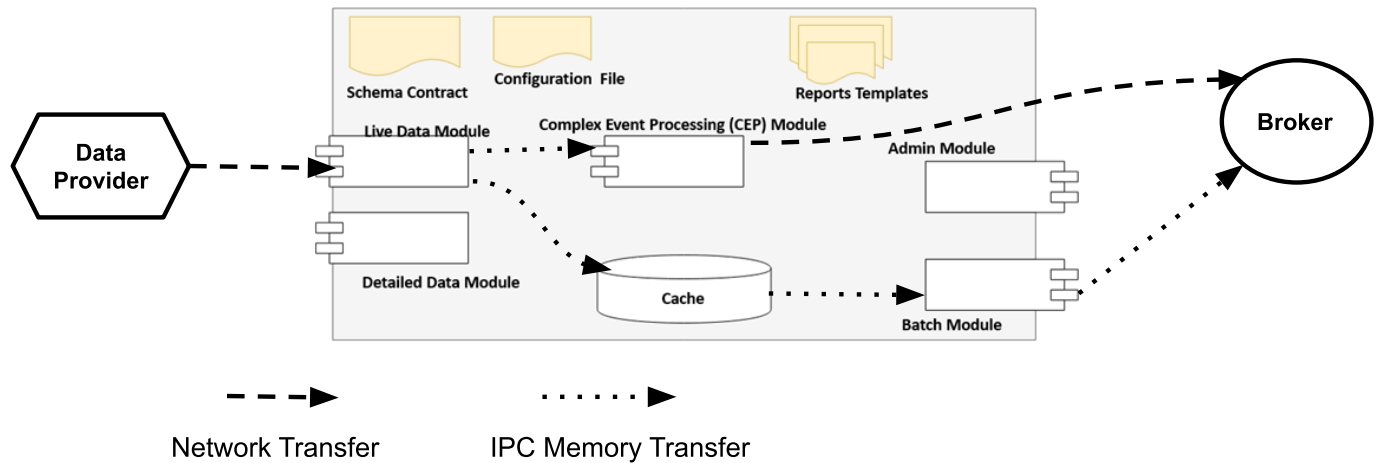


Figure 11 Live Data Flow

1. Data is streamed from the data provider live over a network connection to the live data module which accepts data at a specific open TCP port.
2. Live Data Module uses the schema contract to validate and apply light computation on the incoming data whether it's as simple as attribute selection or ETL-like operations.
3. Live Data Module forwards processed data to the Complex Event Processing (CEP) module and to the cache storage in parallel at the same time via internal memory.
4. In the CEP module the stream of data is executed on the event pattern query. Upon any match, the event is forwarded at real-time to the broker via network connection. CEP publishes to a specific broker topic to avoid real-time alerts delays.
5. The batch module runs at custom time intervals, collects cached data, and sends them in a batch to the broker via a network connection.

### 3.4.3. On-Demand Data Flow

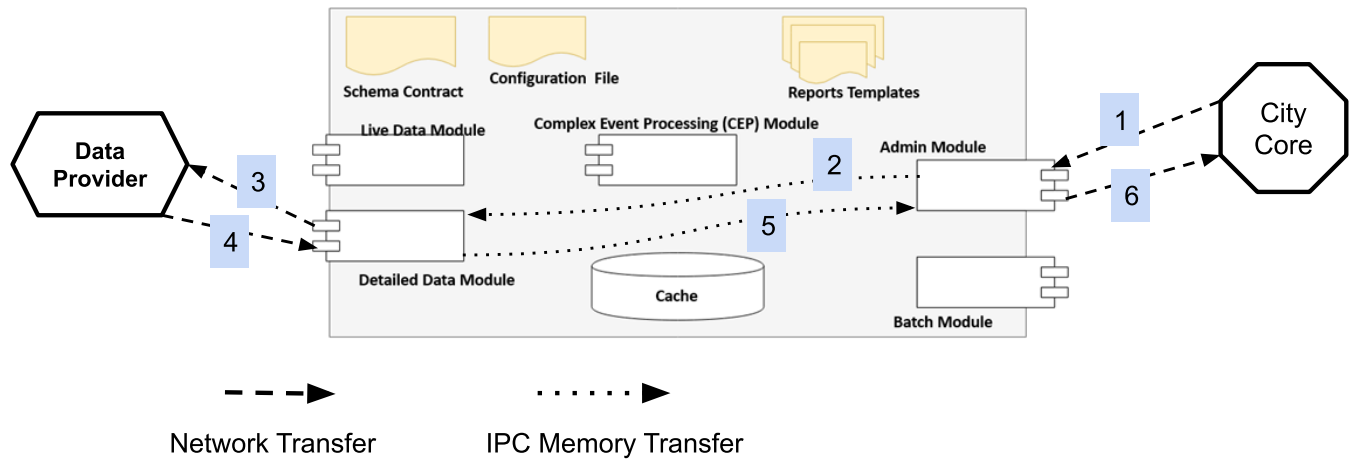


Figure 12 On-Demand Data Flow

1. CityPro Core initiates this process by sending a request, which is a detailed query about a specific subject, to the admin module which awaits connections and commands.
2. The admin module initiates a new instance of the detailed data module with the proper report parameters such as the exact datastore query and the report template.
3. The detailed data module has the capability of querying different types of databases whether SQL or NoSQL.
4. After querying the provider's database at rest and getting the result
5. The detailed data module will build the report and forward it to the admin module via inter-process communication.
6. The admin module will send back the report to the CityPro Core.

## 3.5. Key Design & Implementation Decisions and Techniques

After laying down the architecture, specifications, and functionalities of the edge part of the middleware, we implemented this framework. Next, we'll list technical key design and implementation points. We used modern, standard, and opensource technologies while implementing the software stack. Also, the performance was taken care-of along the process.

Providing a standard and unified interface to the data through the edge device is one of the main goals. For Example, we are using SQL to query any type of database and SQL is also used to write event patterns for the Complex Event Processing (CEP).

### 3.5.1. Kotlin Programming Language

We used Kotlin [22] as the main programming language for the edge framework. Kotlin was introduced in 2011 by JetBrains. The main reasons for choosing Kotlin are:

- It allows us to write functional-style code and run it with object-oriented on the same JVM.
- It's 100% compatible and inter-operable with the JVM, which allowed us to use existing Java libraries. Noting that most Apache and data-related libraries are distributed as Java Libraries.
- It's safe because it avoids NullPointerException and acting on Nullable types by detecting them on compilation.
- Multi-platform support: JVM, Android, Browsers, and Kotlin Native on embedded devices without a virtual machine.
- Modern exclusive features such as co-routines and extension functions
- Supported and adopted by big companies such as Google and Uber

### 3.5.2. Apache Avro for Data On-Wire Serialization and Schema

Apache Avro™ is a data serialization system [23]. It's part of the Apache Hadoop [24] ecosystem and designed to be compact, fast, and language independent. We selected Apache Avro as the serialization library for the data between the provider and the edge device. Three key points for selecting Apache Avro:

- It uses binary format which makes it more performant than XML, JSON, or any text-based serialization
- Although it's binary-based it uses a schema file (JSON) that describe the data to serialize and de-serialize
- It's widely adopted and supported

```
{
  "type": "record",
  "name": "CDR",
  "namespace": "citypro.edge.avro",
  "fields": [
    {
      "name": "ID",
      "type": "string"
    },
    {
      "name": "CALLING_NUM",
      "type": "string"
    },
    {
      "name": "CALLED_NUM",
      "type": "string"
    },
    {
      "name": "START_TIME_I",
      "type": "long"
    }
  ]
}
```

Figure 13 - Avro File - Schema Contract

We also used the Apache Avro schema file (.avsc) as the schema contract between the CityPro government agency and the data provider. Our edge device should validate incoming data with this schema. Figure 12 shows an example of an Avro schema file used as a schema contract in our project.

### 3.5.3. Embedded Apache Spark SQL Engine for Heterogeneous Databases

One of the main challenges in our research is the ability to query heterogeneous databases. We opted to use a proven to work, opensource and supported solution hence Apache Spark SQL [11]. Apache Spark is composed of different modules as shown in figure 12.

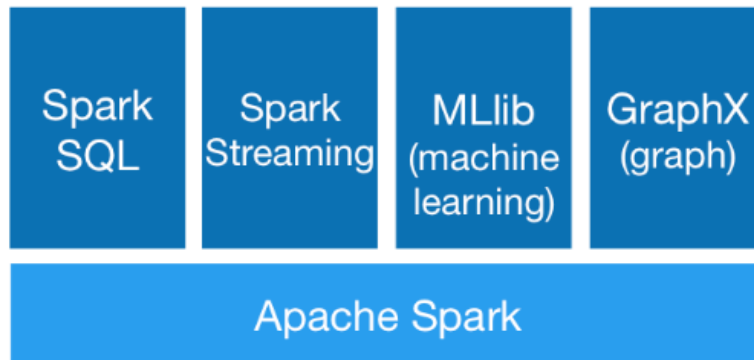


Figure 14 - Apache Spark Architecture

We embedded only the Apache Spark SQL module in our detailed-data component to access a variety of data sources by utilizing these two libraries 'org.apache.spark: spark-core\_2.11:2.4.3' and 'org.apache.spark: spark-sql\_2.11:2.4.3' from the maven repository [25]. This gives us the ability to use standard SQL to query different data sources. Furthermore, Apache Spark SQL provides two interfaces “DataFrame” and “DataSet” to add and extend data sources and data operations.

### 3.5.4. Embedded Siddhi.io for CEP

For the CEP module we embedded Siddhi.io [19] as a Java library. Their site introduces it as “Fully open-source, cloud-native, scalable, Streaming and Complex Event Processing System capable of building real-time analytics, data integration, notification and surveillance use cases”. The Input Stream in our case is the “live data module” stream while the output is a stream published to a broker endpoint.



Figure 15 – Siddhi [19]

```

21  var siddhiApp = "" +
22      "define stream EventStream (ID string, CALLING_NUM string, CALLED_NUM string, START_TIME_I long); "
23      "@sink(type='kafka',topic='${confObj["brokerTopicAlert"]}',partition.no='0',bootstrap.servers='${con
24      "define stream OutputStream (ID string, CALLING_NUM string, CALLED_NUM string, START_TIME_I long); "
25      "@info(name = 'query1') " +
26      "from EventStream[str:contains(CALLING_NUM, '07')] " +
27      "select ID, CALLING_NUM, CALLED_NUM, START_TIME_I " +
28      "insert into OutputStream;"
29
30  //Generate runtime
31  var siddhiAppRuntime = siddhiManager.createSiddhiAppRuntime(siddhiApp)
32
33  fun pushToCEPStream(event: Array<Any>) {
34      siddhiAppRuntime.getInputHandler("EventStream").send(event)
35  }
36
37

```

Figure 16 - Siddhi Code Snippet from CEP module

There are three main parts in a Siddhi based App as shown in our code snippet:

- Define the input stream
- Define the output stream (@sink), in our case push to a Kafka broker
- Define the pattern query that you would like to catch, in the above code we're looking for calling numbers containing '07'

### 3.5.5. Generic XML Templates and Java Reflection

We pushed our framework to be generic, dynamic, and provide standard interfaces. Part of this effort is on the detailed data report. To serve reports from a variety of data sources we use a configuration XML file and Java reflection techniques.

```

<?xml version='1.0' encoding='utf-8'?>
<report name="started_calls.xml">
  <query text="SELECT CALLED_NUM from FROM CDR WHERE CALLING_NUM = $0">
    <inputs>
      <input key="$0" />
    </inputs>
  </query>
  <result>
    <parser className="main.ST_PARSER" />
  </result>
</report>

```

Figure 17 - Edge Framework Report Template



As shown in the above figure, the XML file describes the input and the output. The input is an SQL query which is executed by the Apache Spark module as described in a previous section. The query also provides place holders for the query parameters. The output defines the java class that is responsible for parsing the result of the query. A parsing class should implement an interface with a single method (parse) to handle a “DataSet” returned by Apache Spark SQL. We consider the java class distributed within the custom edge package and we use Java reflection to load this class at runtime. This will make the data reporting module unaware of any report structure.

### **3.5.6. WebSocket Communication for Admin Module**

The Admin module keeps a direct connection with the “City Core System” that doesn’t go through the broker. This connection is mainly for direct commands, remote configurations, and status reporting. For this kind of exchange, there is a need for a “keep-a-alive” connection. We implemented it using the WebSocket protocol [26]. WebSocket is based on TCP; a full-duplex and bi-directional connection. In addition, WebSockets don’t use pulling; the server can push data to the client at any time based on events which make it performant in real-time data scenarios. In our implementation we use the Javalin library [27] to host a WebSocket Server on the edge device, and the client is the City Core System which can connect to all the distributed edge devices at the same time using WebSocket client API. It’s worth to note that WebSockets can run over Secure Socket Layer (SSL) for encryption.

### **3.5.7. Inter-Process Communication over Memory-Mapped Files**

Inter-Process Communications (IPC) is a critical part in standardizing and optimizing data flow between the different components in our edge framework. For these reasons, we used IPC over memory-mapped files using Mappedbus library “a Java-based high throughput, low latency message bus” [28]. It’s clocked @14 million messages per second on Intel i7-4558U @ 2.8 GHz.

## **4. Proof of Concept (PoC) – Telecom Test Case**

To test the “Edge Centric Middleware” the inner workings and how the data flows we considered the case of Telecom Call Detail Records (CDR). We generated live phone calls and used Microsoft SQL Server as a database solution at the provider side. Our edge part of the middleware was deployed on a RaspberryPi [21] board and connected to the simulated database. The edge device detected alerting patterns and sent them to the broker, in addition to sending batches of data. We configured a Kafka broker with two topics and validated the data flow. Furthermore, we simulated an the city core system panel to test the admin channel. A complete video record of the experiment is available @<https://www.youtube.com/watch?v=frQ1TC4RaEs> *(hint use the HD view setting)*

### **4.1. Preparing a DataSet**

Due to privacy concerns, there are no real data sets for telecom CDR. So, we generated random CDR records. The CDR schema includes ID, CALLING\_NUM, CALLED\_NUM, START\_TIME, END\_TIME, CALL\_TYPE, CHARGE, and CALL\_RESULT

We wrote a NodeJS [29] script to randomize the values while keeping the numbers in Lebanese format. This script keeps running emulating current phone calls that are taking place right now.

### **4.2. Database Engine and Notification Service**

After generating call detail records (CDR) we consider Telecom as a data provider system. At the provider side we used Microsoft SQL Server [30] as the database solution. To establish live data from the telecom data provider system to the edge framework we implemented a .NET service that wraps an MS SQL Server feature called “query notifications”. “Query Notifications” are best defined and documented as “query notifications allow applications to be notified when data changes. This feature is particularly useful for applications that provide a cache of information from a database.” [31] Using query notifications our .NET service streams any new data inserted in SQL Server to our edge framework. As mention before this stream is serialized using Apache Avro [23].

### 4.3. Edge Framework on RaspberryPi

We highlighted in previous sections the key points and technologies in the edge framework software layer. The software stack for the edge is cross-platform so we don't have a problem in selecting the hosting operating system. For the hardware, the edge software can scale from a small computer board to a rack of enterprise servers according to the load at each data provider. This makes it more efficient in any budget planning. For our testing, we deployed it on a RaspberryPi [21] board with the following specifications:

- Model 3 B+
- 1 GB RAM
- 32GB Storage
- Quad-core 64-bit processor clocked at 1.4GHz.
- 300Mbps Ethernet



*Figure 18 RaspberryPi 3 B+*

There is a selection of operating systems for this board. We installed a light Linux based distribution without a graphical user interface to boost performance by being lighter on RAM and CPU. After the OS all we need is a JVM, we tested it on OpenJDK 8 JVM.

Deploying and running the edge software framework on limited resources such as the RaspberryPi board proved the performance and the work that has been done to optimize the computing footprint.

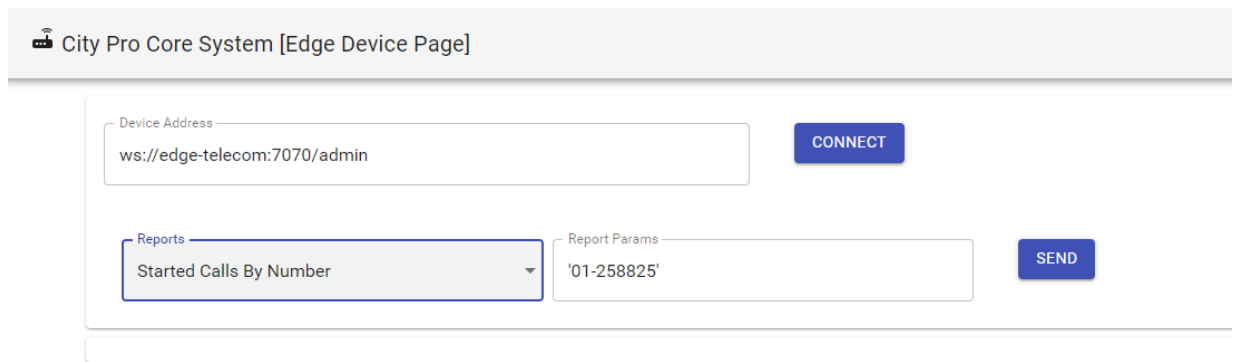
### 4.4. Kafka Broker

At the broker of the “Edge Centric Middleware,” we tested it with Apache Kafka [13] as the middle broker. For the telecom (CDR) data case we created two topics: one for normal data batches, and one for alerts. The batch component in the edge framework published messages to the normal topic while the CEP module published messages to the alert topic. This will guarantee fast delivery for alerts and then the infrastructure supporting each topic can be scaled and optimized accordingly.

## 4.5. Simple Web Interface to Connect to Admin Module

To test the admin and the detailed report components we build a simple web UI to connect, send commands, and receive reports from the edge device. This UI emulates a control interface in the “City Core System”. This simple web UI uses WebSocket [26] client APIs to connect the WebSocket server at the edge device.

After successful connection, you can pick the desired report and fill any report params and send the query. This will match the report template file distributed with the edge device that we described in section 3.5.5. The resulted report is sent back to this interface using the same WebSocket channel. Using and xml result helps to create dynamic visualizations.



City Pro Core System [Edge Device Page]

Device Address  
ws://edge-telecom:7070/admin

CONNECT

Reports  
Started Calls By Number

Report Params  
'01-258825'

SEND

Figure 19 City Core Simple UI

## 5. Conclusion & Future Work

---

Smart cities tackle development obstacles and improve the quality of life for citizens. CityPro provides a platform that focuses on city protection by supporting the collaboration of existing operational systems. While the need for a robust and standard middleware is critical for any smart city platform.

However, due to challenges such as continuous big data streams, heterogeneous systems, security, and privacy, in-addition to non-opensource software solutions there is a lack of such a middleware. This research proposes a new architecture for a smart city middleware using emerging edge computing trends and provides an open-source implementation for the proposed framework.

The "Edge Centric Middleware" consists of two parts: edge devices distributed along with the data providers' systems and a broker between edge devices and the "City Core System". We delegate some computation tasks to the distributed edge devices which provide a uniform and standard interface to the variety of existing systems. The edge devices follow the concept of black box to tackle privacy and security concerns. While the broker handles the scalability and availability of message queues from a distributed network of edge devices.

Although we didn't implement much security features due to time constraints, we consider our solution as a security and privacy enabler. For example, we deployed our framework on a separated and dedicated hardware where we can add physical tampering detection. In addition, only our certified edge devices can send messages to the broker. Furthermore, encryption on the network layer and on the device cache storage can be added.

More metrics and experimental validations are needed. We had very limited time to develop a PoC (Proof of Concept) for this research. More experimentation should stress test big data flows.

The software implementation is based on standard and opensource technologies. Developing within an opensource community helps in boosting the pace of solving issues and adding features. That's why we publish also the source code of our work.

## References

- [1] M. Dbouk, M. Mcheick and I. Sbeity, "CityPro: city-surveillance collaborative platform," *Int. J. Big Data Intelligence*, vol. 4, no. 3, 2017.
- [2] W. Shi, J. Cao and Q. Zhang, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, 2016.
- [3] Cisco, "Edge computing vs. fog computing: Definitions and enterprise uses," [Online]. Available: <https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html>.
- [4] Cisco, "Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper," 19 November 2018. [Online]. Available: <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html>.
- [5] B. Rinner, M. Jovanovic and M. Quaritsch, "Embedded Middleware on Distributed Smart Cameras," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, 2007.
- [6] M. Fazio, M. Paone, A. Puliafito and M. Villari, "Heterogeneous Sensors Become Homogeneous Things in Smart Cities," in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2012.
- [7] F. J. Villanueva, M. J. Santofimia, D. Villa, J. Barba and J. C. López, "Civitas: The Smart City Middleware, from Sensors to Big Data," in *2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2013.
- [8] A. d. M. D. Esposte, F. Kon, F. M. Costa and N. Lago, "InterSCity: A Scalable Microservice-based Open Source Platform for," in *6th International Conference on Smart Cities and Green ICT*, 2017.
- [9] L. Wang and R. Ranjan, "Processing Distributed Internet of Things Data in Clouds," *IEEE Cloud Computing*, vol. 2, no. 1, 2015.
- [10] Z. Liu, F. Cretton, A. L. Calvé, N. Glassey, A. Cotting and F. Chapuis, "MUSYOP: Towards a Query Optimization for Heterogeneous Distributed Database," in *International conference*

*on Computing Technology and Information Management*, Dubai, 2014.

- [11] Apache Spark, [Online]. Available: <https://spark.apache.org/>.
- [12] TIBCO Inc., "What is a Message Broker?," [Online]. Available: <https://www.tibco.com/reference-center/what-is-a-message-broker>.
- [13] "Apache Kafka," [Online]. Available: <https://kafka.apache.org/>.
- [14] "Apache ActiveMQ Artemis," [Online]. Available: <https://activemq.apache.org/components/artemis/>.
- [15] A. Zookeeper. [Online]. Available: <https://zookeeper.apache.org/>.
- [16] N.-L. Tran, S. Skhiri and E. Zim'nyi, "EQS: An Elastic and Scalable Message Queue for the Cloud," in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, 2011.
- [17] D. Luckham and B. Frasca, "Complex Event Processing in Distributed Systems," Stanford University, 1998.
- [18] "Apache Flink," [Online]. Available: <https://flink.apache.org/news/2016/04/06/cep-monitoring.html>.
- [19] "Siddhi," [Online]. Available: <https://siddhi.io/>.
- [20] "Esper," [Online]. Available: <http://www.espertech.com/esper/>.
- [21] "RaspberryPi," [Online]. Available: <https://www.raspberrypi.org>.
- [22] "Kotlin," [Online]. Available: <https://kotlinlang.org/>.
- [23] "Apache Avro," [Online]. Available: <https://avro.apache.org/>.
- [24] "Apache Hadoop," [Online]. Available: <https://hadoop.apache.org/>.
- [25] "Maven Central Repository," [Online]. Available: <https://mvnrepository.com/repos/central>.
- [26] "WebSocket Protocol RFC," [Online]. Available: <https://tools.ietf.org/html/rfc6455>.
- [27] "Javalin," [Online]. Available: <https://javalin.io/>.

- [28] caplogic, "Mappedbus," [Online]. Available: <http://mappedbus.io/>.
- [29] "NodeJS," [Online]. Available: <https://nodejs.org/en/>.
- [30] "Microsoft SQL Server," [Online]. Available: <https://www.microsoft.com/en-us/sql-server/>.
- [31] Microsoft, "Query Notifications in SQL Server," [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/query-notifications-in-sql-server>.
- [32] N. Kyung-Won and J.-S. Park, *Software Platform Architecture for Ubiquitous City Management*, 2011.