An Ingestion and Analytics Architecture for IoT applied to Smart City Use Cases

Paula Ta-Shma, Adnan Akbar, Guy Gerson-Golan, Guy Hadash, Francois Carrez, and Klaus Moessner

***Abstract*—As sensors are adopted in almost all fields of life, the Internet of Things (IoT) is triggering a massive influx of data. We need efficient and scalable methods to process this data to gain valuable insight and take timely action. Existing approaches which support both batch processing (suitable for analysis of large historical data sets) and event processing (suitable for real-time analysis) are complex. We propose the *hut* architecture, a simple but scalable architecture for ingesting and analyzing IoT data, which uses historical data analysis to provide context for real-time analysis. We implement our architecture using open source components optimized for big data applications and extend them where needed. We demonstrate our solution on two real-world smart city use cases in transportation and energy management.**

***Index Terms*—big data, complex event processing, context-aware, energy management, ingestion, internet of things, machine learning, smart cities, spark, transportation**

I. Introduction

Sensors are by no means a new phenomenon: the first thermostat was invented in the 19th century and space travel would have been impossible without them. What is revolutionary today about the Internet of Things (IoT) lies in its recent adoption on an unprecedented scale, fueled by economic factors such as dramatic drops in costs of sensors, network bandwidth and processing. Moreover, unlike the Internet (of humans), the IoT allows data to be captured and ingested autonomously, avoiding the human data entry bottleneck. IoT data will arguably become the Biggest Big Data, possibly overtaking media and entertainment, social media and enterprise data. The question then becomes how to make effective use of this vast ocean of data?

The nature of IoT applications beckon real time responses. For example, in the transportation domain one might want to plan a travel route according to current road conditions, and in smart homes one might want to receive timely alerts about unusual patterns of electricity consumption. Some IoT sensors are capable of actuation, meaning that they can take some action, such as turning off the mains power supply in a smart home. Therefore real time insights can be translated into timely actions.

P. Ta-Shma, G. Gerson-Golan and G. Hadash are with the IBM Research, Haifa, Israel (email: paula@il.ibm.com; guyger@il.ibm.com; guyh@il.ibm.com)

A. Akbar, F. Carrez and K. Moessner are with the Institute for Communication Systems, University of Surrey, UK (email: adnan.akbar@surrey.ac.uk; f.carrez@surrey.ac.uk; k.moessner@surrey.ac.uk)

The importance of collecting and analyzing historical IoT data is less immediately apparent. Because of its sheer size, this is a costly endeavour, although the most relevant data for real time decisions would seem to be the most recent data. We argue that historical data analysis is essential in order to reach intelligent decisions, since without it one cannot understand the context of real time data. For example, does the current traffic (15 kph, 300 vehicles per hour) represent normal conditions for a city centre intersection in rush hour, or extreme congestion on a highway after a major accident? Does a sudden increase in home energy consumption result from heating in cold weather, or a faulty appliance? The answer is clear on analysis of the temporal patterns in historical sensor data.

We found that a large and important class of IoT applications has a focused set of requirements which can be handled using a highly streamlined and simplified architecture. We focus on applications which learn from IoT device history in order to intelligently process events in real time. Example applications include event classification (e.g. classifying a traffic event as 'good' or 'bad'), anomaly detection (e.g. alerting when unusual traffic conditions occur), and prediction (e.g. predicting future traffic conditions). We apply our work to smart city transportation and energy management, but it is generally applicable to almost all IoT domains. For example, anomaly detection can also be applied to car insurance (alterting on unusual driving patterns), utility management (alerting on water/oil/gas pipe leakage) and goods shipping (alerting on non compliant humidity and temperature). We present our simple streamlined architecture in this paper, and apply it to both event classification and anomaly detection in two IoT use cases.

To achieve high scalability and low deployment cost, we adopt a cloud based micro-services approach, where each capability (ingestion, storage, analytics etc.) is embodied in a separate scalable service. This approach is gaining widespread popularity for cloud platform-as-a-service (PaaS) [1], since each service specializes in what it does best, and can be managed and scaled independently of other services, avoiding monolithic software stacks. To achieve low development cost we adopt open source frameworks, and we also implemented our solution on the IBM Bluemix PaaS. We choose "best of breed" open source frameworks for each capability, and show how they can be assembled to form solutions for IoT applications.

The following contributions are made in this paper.

- We propose a streamlined and simplified architecture for a large and important class of IoT applications.

We name it the *hut* architecture because its flow diagram takes the form of a hut as shown in Figure 1. We use historical (batch) analytics to improve the quality of real-time analytics on IoT data.

- We implement our proposed architecture using a micro-services approach with best of breed open source frameworks while making extensions as needed. Our proposed solution is flexible with respect to the choice of specific analysis algorithms and suitable for a range of different machine learning and statistical algorithms.
- We demonstrate the feasibility of our proposed solution by implementing it for two real-world smart city use cases in transportation and energy management. We implement the transportation scenario on the IBM Bluemix PaaS and make the code available as open source.

The remainder of the paper is organized as follows.

Section II presents related work and explains how we extend prior research. Section III explains our proposed architecture along with descriptions of the various components involved in its implementation. Section IV-A describes the application of our proposed architecture to a smart transportation use case scenario. Section IV-B demonstrates the application of our solution to smart energy management. Finally we conclude the paper and highlight future work in section V.

II. Related Work

The massive proportions of historical IoT data highlight the necessity of scalable and low cost solutions. At first glance, IoT data is similar to Big Data from application domains such as clickstream and online advertising data, retail and e-commerce data, and CRM data. All these data sources have timestamps, are (semi) structured, and measure some metrics such as number of clicks or money spent. Similarly, the need to scalably ingest, store and analyze data from these domains is somewhat similar.

Analytics frameworks for Big Data can often be categorized as either batch or real-time processing frameworks. Batch processing frameworks are suitable for efficiently processing large amounts of data with high throughput but also high latency - it can take hours or days to complete a batch job. Real-time processing typically involves time sensitive computations on a continuous stream of data.

One of the most common and widely used techniques for batch processing on Big Data is called MapReduce [2]. MapReduce is a programming model for carrying out computations on large amounts of data in an efficient and distributed manner. It is also an execution framework for processing data distributed among large numbers of machines. It was originally developed by Google as a generic but proprietary framework for analytics on Google's own Big Data, and later was widely adopted and embodied in open source tools. MapReduce was intended to provide a unified solution for large scale batch analytics and address challenges like parallel computation, distribution of data and handling of failures.

Hadoop [3], an open source embodiment of MapReduce, was first released in 2007, and later adopted by hundreds of companies for a variety of use cases. Notably, Amazon released Elastic Map Reduce (EMR) [4], a hosted version of MapReduce integrated into its own cloud infrastructure platform running Amazon Elastic Compute Cloud (EC2)[5] and Simple Storage Service (S3)[6]. OpenStack has a similar framework called Sahara which can be used to provision and deploy Hadoop clusters [7].

Hadoop provides generic and scalable solutions for big data, but was not designed for iterative algorithms like machine learning, which repeatedly run batch jobs and save intermediate results to disk. In such scenarios, disk access can become a major bottleneck hence degrading performance.

In order to overcome the limitations of Hadoop, a new cluster computing framework called Spark [8] was developed. Spark provides the ability to run computations in memory using Resilient Distributed Datasets (RDDs) [9] which enables it to provide faster computation times for iterative applications compared to Hadoop. Spark not only supports large-scale batch processing, it also offers a streaming module known as Spark streaming [10] for real-time analytics. Spark streaming processes data streams in micro-batches, where each batch contains a collection of events that arrived over the batch period (regardless of when the data was created). It works well for simple applications but the lack of true record-by-record processing makes time series and event processing difficult for complex IoT applications.

The need for real time processing of events in data streams on a record-by-record basis led to a research area known as complex event processing (CEP) [11]. CEP is specifically designed for latency sensitive applications which involve large volumes of streaming data with timestamps such as trading systems, fraud detection and monitoring applications. In contrast to batch processing techniques which store the data and later run queries on it, CEP instead stores queries and runs data through these queries. The inbuilt capability of CEP to handle multiple seemingly unrelated events and correlate them to infer complex events make it suitable for many IoT applications. The core of CEP is typically a rule-based engine which requires rules for extracting complex patterns. A drawback of CEP is that the authoring of these rules requires system administrators or application developers to have prior knowledge about the system which is not always available.

Big Data analytics systems have the challenge of processing massive amounts of historical data while at the same time ingesting and analyzing real-time data at a high rate. The dichotomy of event processing frameworks for real time data, and batch processing frameworks for historical data, led to the prevalence of multiple independent systems analyzing the same data. The Lambda architecture was proposed by Nathan Marz [12] to address this, and provides a scalable and fault tolerant architecture for processing both real-time and historical data in an integrated fashion. The purpose of this architecture was to analyze vast amounts of data as it arrives in an efficient, timely and fault tolerant fashion. Its focus was on speeding up Online Analytical Processing (OLAP) style computations, for example web page view and click stream analysis. It was not designed to make per-event decisions or respond to events as they arrive [13]. It comprises batch, speed

and serving layers, which must be coordinated to work closely together, and is complex and difficult to deploy and maintain [14].

In contrast to existing solutions, our architecture focuses on analyzing new events as they arrive with the benefit of wisdom gained from historical data. This encompasses a large class of algorithms including event classification, anomaly detection and event prediction. Our architecture is simpler and more focused than the lambda architecture, and it maps well to a microservices approach where minimal coordination is needed between the various services. Using our approach batch analytics is used independently on the historical data to learn the behaviour of IoT devices, while incoming events are processed on a record-by-record basis and compared to previous behaviour. Newly ingested data will eventually become part of the historical dataset, but unlike the lambda architecture, new events do not need to immediately be analyzed on a par with historical data. Our approach is practical, scalable and has low cost to develop, deploy and maintain.

III. A Simplified Architecture for IoT Applications

We propose the *Hut* Architecture, which meets the requirements of scalable historical data analytics as well as efficient real-time processing for IoT applications. IoT applications typically require responding to events in real time based on knowledge of past events. For example, using knowledge of past traffic behaviour for certain locations in certain times to trigger alerts on unexpected patterns such as congestion. Historical knowledge is essential in order to understand what behaviour is expected and what is an anomaly. Historical data must be analyzed ahead of time in order to allow real time responses to new situations. Despite its simplicity, our architecture can scale to deal with large amounts of historical data and can detect complex events in near real-time using automated methods.

A. The Hut Architecture

Our architecture gives a high level view of the various components in a solution and orchestrates how they fit together to solve a problem. Figure 1 presents its data flow diagram, which forms the shape of a *hut*. The purple arrows denote the batch data flows which form the base of the *hut*, while the green arrows denote the real time flows and form the roof of the *hut*. We first describe the real time flows.

Data acquisition denotes the process of collecting data from IoT devices and publishing it to a message broker. An event processing framework consumes events and possibly takes some action (actuation) affecting the same or other IoT devices or other entities such as a software application. Real time flows can be stand alone, in cases where real time data can be acted upon without benefitting from historical data, although usually historical data can provide further insight in order to make intelligent decisions on real-time data. For example, in order to recognize anomalies, a system first needs to learn normal behavior from historical data [15].

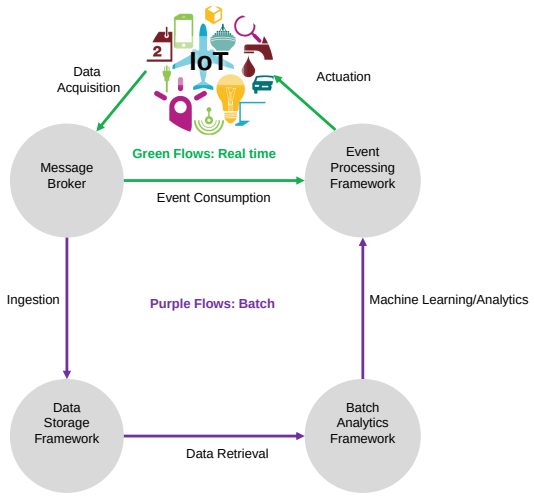


Fig. 1. The Hut Architecture

The batch flows fulfil this purpose. Data is ingested from the message broker into a data storage framework for persistent storage. Data can then be retrieved and analyzed using long running batch computations, for example, by applying machine learning algorithms. The result of such analysis can influence the behavior of the real time event processing framework. The batch flows can work independently of the real time flows to provide long term insight or to train predictive models using historical datasets [16].

B. A Hut Architecture Instance

For each node in Figure 1, one can choose among various alternatives for its concrete implementation. We refer to a certain choice of such components as a *hut architecture instance*. We now present a specific hut architecture instance, and later apply it to multiple real life use cases in following sections. We utlize existing proven open source components, as well as extending them where needed. A diagram of this instance is shown in Figure 2.

The role of each component and how it fits into overall architecture is described below. Where relevant we explain why we chose the relevant component.

1) Data Acquisition - Node Red: We use Node Red [17] to acquire data from heterogeneous devices or other information sources such as RESTful web services or MQTT data feeds. XML and JSON are two most commonly used formats which are used extensively for transmitting IoT data, although there is no limitation regarding the choice of format. Data feeds may contain redundant data which can be pre-processed or filtered. Node-Red provides these functionalities together with a fast prototyping capacity to develop wrappers for heterogeneous data sources. Node Red can then publish the data to the message broker. We chose this component because of its ease of use and flexibility.

2) Message Broker - Kafka: Message brokers typically provide a mechanism for publishing messages to certain topics and allowing subscription to those topics. In our context, the messages typically denote the state of an IoT device at a

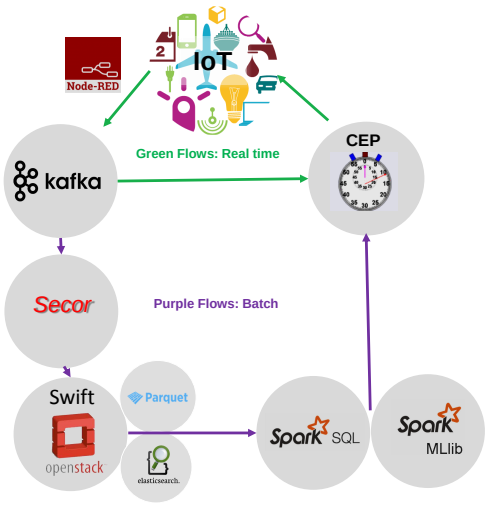


Fig. 2. Proposed solution Architecture

certain time. Apache Kafka [18] is an open source message broker originally developed by LinkedIn, designed to allow a single cluster to serve as the central messaging backbone for a large organization. Kafka emphasizes high throughput messaging, scalability, and durability. Although Kafka is less mature than other systems such as Rabbit MQ, it supports an order of magnitude higher throughput messaging [18]. Moreover, Kafka supports both batch consumers that may be offline, and online consumers that require low latency. Importantly Kafka can handle large backlogs of messages to handle periodic ingestion from systems such as Secor, and allows consumers to re-read messages if necessary. This scenario is important for our architecture. We chose Kafka for both these reasons.

3) Ingestion - Secor: Secor is an open source tool [19] developed by Pinterest which allows uploading Apache Kafka messages to Amazon S3. Multiple messages are stored in a single object according to a time or size based policy. We enhanced Secor by enabling OpenStack Swift targets, so that data can be uploaded by Secor to Swift, and contributed this to the Secor community. In addition we enhanced Secor by enabling data to be stored in the Apache Parquet format, which is supported by Spark SQL, thereby preparing the data for analytics. Moreover, we enhanced Secor to generate Swift objects with metadata. We chose Secor because it is an open source connector between Kafka and object storage (OpenStack Swift).

4) Data Storage Framework - OpenStack Swift: OpenStack [20] is an open source cloud computing software framework originally based on Rackspace Cloud Files [21]. OpenStack is comprised of several components, and its object storage component is called Swift [22]. OpenStack Swift supports CReate, Update and Delete (CRUD) operations on objects using a REST API, and supports scalable and low cost deployment using clusters of commodity machines. For this reason Swift is suitable for long term storage of massive amounts of IoT data. We chose OpenStack Swift because it is an open source object storage framework.

5) The Parquet Data Format: Apache Parquet [23] is an open source file format designed for the Hadoop ecosystem that provides columnar storage, a well known data organization technique which optimizes analytical workloads. Using this technique, data for each column of a table is physically stored together, instead of the classical technique where data is physically organized by rows. Columnar storage has two main advantages for IoT workloads. Firstly, organizing the data by column allows for better compression. For IoT workloads, many columns will typically contain IoT device readings which fluctuate slowly over time, for example temperature readings. For this kind of data some kind of delta encoding scheme could significantly save space. Note that each column can be compressed independently using a different encoding scheme tailored to that column type. Secondly, organizing the data according to columns means that if certain columns are not requested by a query then they do not need to be retrieved from storage or sent across the network. This is unlike the classical case where data is organized by rows and all columns are accessed together. This can significantly reduce the amount of I/O as well as the amount of network bandwidth required. We chose Parquet for these reasons - it is considered as one of the highest performing storage formats in the Hadoop ecosystem [24].

6) Metadata Indexing and Search using Elastic Search: OpenStack Swift allows annotating objects with metadata although there is no native mechanism to search for objects according to their metadata. This is essential in a scenario where we store massive amounts of IoT data and need to analyze specific cross sections of the data. We built a metadata search prototype similar to that of IBM SoftLayer [25] but extended with range searches and data type support to meet the needs of IoT use cases. Our prototype uses Elastic Search [26], based on Lucene[27]. We found it to be effective for our needs, although other Lucene based search engines, such as Solr [28], are available.

7) Batch Analytics Framework - Spark: Apache Spark is a general purpose analytics engine that can process large amounts of data from various data sources and has gained significant traction. It performs especially well for multi-pass applications which include many machine learning algorithms [9]. Spark maintains an abstraction called Resilient Distributed Datasets (RDDs) which can be stored in memory without requiring replication and are still fault tolerant. Spark can analyze data from any storage system implementing the Hadoop FileSystem API, such as HDFS, Amazon S3 and OpenStack Swift, which, together with performance benefits and SQL support (see next section), is the reason for our choice.

8) Data Retrieval - Spark SQL: RDDs which contain semi-structured data and have a schema are called DataFrames and can be queried according to an SQL interface. This applies to data in Hadoop compatible file systems as well as external data sources which implement a certain API, such as Cassandra and MongoDB. We implemented this API for OpenStack Swift with Parquet and Elastic Search, to allow taking advantage of metadata search for SQL queries.

9) Machine Learning - Spark ML: Spark MLlib [29] is Sparks library for machine learning. Its goal is to make

practical machine learning scalable and easy to use. Spark MLlib consists of common machine learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs.

10) Event Processing Framework - CEP: A Complex Event Processing (CEP) Engine is a software component capable of asynchronously detecting independent incoming events of different types and generating a Complex Event by correlating these events together. In this sense, Complex Events can be defined as the output generated after processing many small, independent incoming input data streams, which can be understood as a given collection of parameters at a certain temporal point. A CEP Engine is commonly provided with a series of plugins or additional sub-components in order to improve data acquisition from external sources, and also some kind of rule system to implement the business logic which creates the output of the system.

Our architecture is modular, so a particular component in this instance could be replaced by another. For example, Spark Streaming or Apache Storm could be used for the event processing framework instead of CEP software, and Hadoop map reduce could be used instead of Spark. Our focus here is on the architecture itself, and in order to demonstrate the architecture we made an intelligent choice of open source components as an architecture instance.

IV. USE CASES

The hut architecture, as well as our instance, is generic and can be applied to a range of IoT use cases. In this section we demonstrate its application to real-world problems and show how it can provide optimized, automated and context-aware solutions for large scale IoT applications. We demonstrate it in practice by applying it to the following two scenarios, Madrid Transportation and Taiwan Energy Management. We describe the first use case in detail and later describe how the same architecture and data flow can be applied to the second case. Despite the fact that these use cases are from different domains, they share the same architecture and data flow. Each use case has specific requirements which dictate different configurations and extensions which are also described in this section.

A. Use case 1: Madrid Transportation

Madrid Council has deployed roughly 3000 traffic sensors in fixed locations around the city of Madrid on the M30 ring road, as shown in Figure 3(a), measuring various traffic parameters such as traffic intensity and speed. Traffic intensity represents the average number of vehicles passing through a certain point per unit time whereas traffic speed represents the average speed of vehicles per unit time. Aggregated data is published as an IoT service using a RESTful API and data is refreshed every 5 minutes [1].

Madrid Council has control rooms where traffic administrators analyze sensor output and look for congestion or other traffic patterns requiring intervention as shown in Figure 3(b). Much of the work is manual and requires training and expertise regarding expected traffic behaviour in various parts of the city. Our objective is to automate this process and therefore provide a more responsive system at lower cost. Our approach is to collect traffic data for different locations and time periods and use this to model expected traffic behaviour using thresholds. We then monitor traffic in real time and assess the current behaviour compared to thresholds which capture what is expected for that location and time of day. Our system can alert traffic managers when an action may need to be taken, such as modifying traffic light behaviour, alerting drivers by displaying traffic information on highway panels, calling emergency vehicles and rerouting buses to avoid road blocks. In future our system could trigger these actions automatically.

We now describe our hut instance as applied to the Madrid Transportation use case.

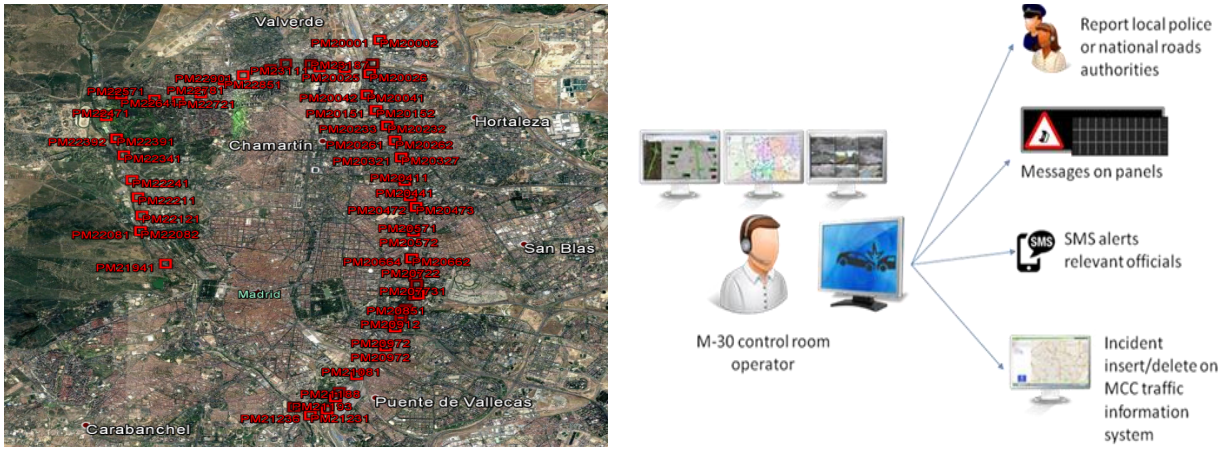
1) Data Acquisition: We used the Node-RED tool to periodically retrieve data from the Madrid Council web service and publish it to a dedicated Kafka topic, containing data from all of Madrid's traffic sensors. The published data has the following schema, where *intensity* denotes traffic intensity, *velocity* denotes traffic speed, *ts* denotes the timestamp in epoch format and *tf* denotes the time of day.

```
{"namespace": "cosmos",
  "type": "record",
  "name": "TrafficFlowMadridPM",
  "fields": [
    {"name": "code", "type":"string"},
    {"name": "occupation", "type":"int"},
    {"name": "load", "type":"int"},
    {"name": "service_level", "type":"int"},
    {"name": "velocity", "type":["null","int"]},
    {"name": "intensity", "type":["null","int"]},
    {"name": "error", "type":"string"},
    {"name": "subarea",  "type":["null","int"]},
    {"name": "ts", "type":"long"},
    {"name": "tf", "type":"string"}
  ]
}
```

2) Data Ingestion: We configured Secor to consume data from this Kafka topic and upload it as objects to a dedicated container in OpenStack Swift once every hour. We partitioned the data according to date which enables systems like Spark SQL to be queried using date as a column name. Using our enhancements to Secor we converted the data to Parquet format, and also generated metadata for each resulting object with minimum and maximum values for specified schema columns, as shown above. This metadata is stored in Swift as well as being sent to Elastic Search for indexing.

3) Data Retrieval: We defined our collection of data and metadata as a Spark SQL external data source, and implemented an associated driver. Given an SQL query over this dataset, our driver identifies selections on indexed columns, and searches Elastic Search for the names of Swift objects whose min/max values overlap the requested query ranges. For the Madrid Traffic use case, we needed to analyze traffic for different periods of the day separately, resulting in the

[1] http://informo.munimadrid.es/informo/tmadrid/pm.xml



(a) Sensors location on M30 ring road

(b) M30 Control Room

Fig. 3. Madrid Transportation Scenario

following query for morning traffic.

```
SELECT code, intensity, velocity
FROM madridtraffic
WHERE tf >= '08:00:00' AND tf <= '12:00:00'
```

To evaluate this query, our driver searches for objects whose min/max timestamps overlap this time period, and evaluates the query on these objects only. Objects which do not qualify do not need to be read from disk or sent across the network from Swift to Spark. For one example query we tested on the Madrid Traffic data we collected, we found our method to reduce the number of Swift requests by a factor of over 20.

4) Event Processing: We used CEP as an event processing component to consume events in real-time from the Message Broker and detect complex events like bad traffic. The core of CEP is a rule-based engine which requires rules for extracting complex patterns. These rules are typically based on various threshold values. An example rule analysing traffic speed and intensity to detect bad traffic events is shown in algorithm 1, which checks whether current speed and intensity cross thresholds for 3 consecutive time points. The manual calibration of threshold values in such rules require traffic administrators to have deep prior knowledge about the city traffic. In addition, rules set using a CEP system are typically static and there is no means to update them automatically.

Algorithm 1 Example Rule for CEP

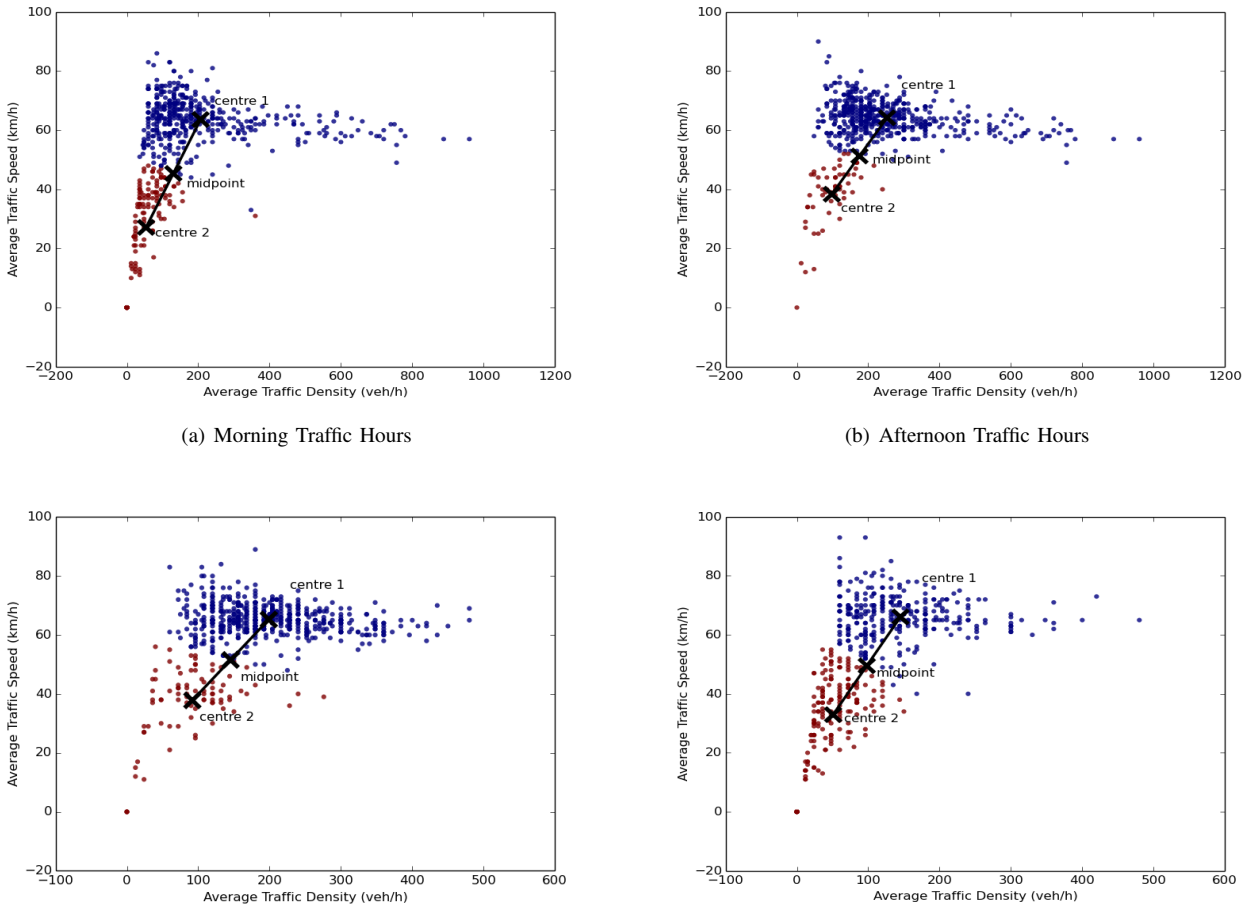
1: **for** $(speed, intensity) \in TupleWindow(3)$ **do**
2: **if** $(speed(t) < speed_{thr}$ and $intensity(t) < intensity_{thr}$ AND
3: $speed(t+1) < speed_{thr}$ and $intensity(t+1) < intensity_{thr}$ AND
4: $speed(t+2) < speed_{thr}$ and $intensity(t+2) < intensity_{thr})$ **then**
5: Generate complex event Bad Traffic
6: **end if**
7: **end for**

In contrast, we adopted a context-aware approach using machine learning to generate optimized thresholds automatically based on historical sensor data and taking different contexts including time-of-day and day-of-week into account. New rules are generated dynamically whenever our algorithm detects a change in the context. The idea of using machine learning to generate optimized thresholds for CEP rules was proposed in our initial work [30] where we demonstrated a context-aware solution for monitoring traffic automatically. In this paper, we improve our initial approach, extend our experimental evaluation and integrate it with the more general *hut* architecture and *hut* instance, optimized for large scale IoT applications.

5) Machine Learning: In order to classify traffic events as 'good' or 'bad' we built a model for each sensor location and time period (morning, afternoon, evening and night) using k-means clustering, an unsupervised algorithm (not requiring labeled training data) implemented in Spark MLlib and optimized for large data sets. The data points are separated into k different groups, in our case $k = 2$ and the groups represent good versus bad traffic. The resulting cluster boundary generates thresholds for real time event processing, since crossing these thresholds signifies moving from good to bad traffic (or vice versa).

Experimentation results of our approach on Madrid traffic data are shown in Figure 4 for a particular location on a weekday. Different sub-figures indicate different time contexts (morning, afternoon, evening and night). Blue clusters represent high average speed and intensity indicating good traffic state, whereas red clusters represent low average speed and intensity indicating bad traffic state (note the varying scales of the X-axes in the various graphs). Midpoints between cluster centers represents the boundary separating both states and we use this boundary to define threshold values for detecting complex events.

6) When to Recompute the Thresholds?: Statistical properties of the underlying data may change over time resulting in inaccurate threshold values. Therefore, we assess the cluster quality for different contexts as new data arrives, and once it significantly deteriorates, we retrain the k-means models and



(a) Morning Traffic Hours

(b) Afternoon Traffic Hours

(c) Evening Traffic Hours

(d) Night Traffic Hours

Fig. 4. Clustering results on Madrid traffic data for location 1

generate new threshold values. The Silhouette index $s(i)$ [31] is used to assess cluster quality by quantitatively measuring the data fitness on existing clusters and is defined as

$$s(i) = \frac{b(i) - a(i)}{max(a(i), b(i))} \tag{1}$$

where $a(i)$ is the mean intra cluster distance, and $b(i)$ is the mean nearest-cluster distance i.e. distance with the nearest cluster center which the data is not part of. $s(i)$ ranges from -1 to 1 where 1 indicates the highest score and -1 the lowest for cluster quality. Table I shows the instance of threshold values for both speed and intensity for location 1 (Figure 4) where values of silhouette index ($s(i) >= 0.5$) indicate good cluster quality. As new data arrives, the silhouette index is calculated incrementally using the cluster centroids and if $s(i) < 0.5$, it acquires the latest data and repeats all steps.

7) Evaluation: In order to evaluate our proposed solution, we followed the approach outlined in [32]. We defined an ideal set of threshold values for the rule mentioned in algorithm 1 for four different locations with the help of traffic administrators from Madrid city council, and refer to this as Rule R. As we have different threshold values for different contexts, we need ideal threshold values for each context to provide fair analysis of results. Rules learned by the automatic generation of threshold values using our proposed clustering algorithm are represented as R^*.

TABLE I
THRESHOLD VALUES UPDATE (WEEKDAYS)

Traffic Period	Time Range	Threshold Values	Silhouette index
Morning	8 am to 12 pm	130 veh/h, 43 km/h	0.51
Afternoon	12 pm to 4 pm	175 veh/h, 51km/h	0.57
Evening	4 pm to 8 pm	145 veh/h, 49km/h	0.55
Night	8pm to 12 am	96veh/h, 48 km/h	0.50

We measure the performance of our system quantitatively by generating an evaluation history of traffic events and use both R and R^* to detect bad traffic events. This enables us to measure the precision of our algorithm which is the ratio of the number of correct events to the total number of events detected; and the recall, which is the ratio of the number of events detected by R^* to the total number of events that should have been detected based on ideal Rule R. Mathematically, they are represented as:

$$Precision = \frac{TP}{TP+FP}, \quad Recall = \frac{TP}{TP+FN} \tag{2}$$

where TP is true positive, FP is false positive and FN is

false negative. Results are shown below in table II. In general, we got high values of recall for all four locations which indicates high rule sensitivity (detecting 90% of events from the traffic data stream). The average value of precision lies at around 80% indicating a small proportion of false alarms.

TABLE II
CEP RULES EVALUATION FOR MADRID SCENARIO

Locations	TP	FP	FN	Precision	Recall
1	97	17	8	0.85	0.92
2	68	14	7	0.82	0.91
3	71	12	11	0.85	0.86
4	112	29	9	0.79	0.93

8) Discussion: The main focus of our work is on a generic architecture for IoT data analytics which allows plugging in various algorithms. We presented one particular choice where unsupervised machine learning (k-means clustering) was used for event classification. Our modular approach enables exploration of other unsupervised or supervised methods for the same problem. In addition, our architecture can be used for additional applications; for example, one can train regression models with Spark MLlib using Madrid Council's historical dataset and provide traffic predictions [33].

B. Use case 2: Taiwan Electricity Metering

Smart energy kits are gaining popularity for monitoring real time energy usage to raise awareness about users' energy consumption [34]. The Institute for Information Industry (III) Taiwan have deployed smart energy kits consisting of smart plugs and management gateways in over 200 residences. These smart plugs have built-in energy meters which keep track of real-time energy usage of connected appliances by logging electrical data measurements. They are connected to a management gateway via the ZigBee protocol, which is connected to the internet via WiFi.

Our aim is to monitor energy consumption data in real time and automatically detect anomalies which are then communicated to the respective users. An anomaly can be defined as unusual or abnormal behaviour. For example, a malfunctioning electronic device or a fridge with its door left open can result in excessive power dissipation which should be detected and reported as soon as possible. Another type of anomaly is appliance usage at unusual times such as a radiator during the summer or an oven operated at 3am. Automatic monitoring of devices to detect anomalies can contribute to energy savings as well as enhanced safety.

III requests users to provide information on devices connected to smart plugs such as appliance type as well as expected behaviour such as expected wattage and current ranges. However expected behaviour is not usually known by users and is difficult for them to determine. Our approach of collecting historical appliance data for various time periods (summer versus winter, day versus night, weekday versus weekend) provides a way to automatically generate reliable information about expected behaviour. For each device and time context (such as weekday mornings during summer), we calculate the normal working range for current and power for an appliance using statistical methods. A CEP rule is defined based on this working range, and as soon as the readings are outside this range a CEP rule will be triggered generating a complex event representing an anomaly which can then be used to notify the user as well.

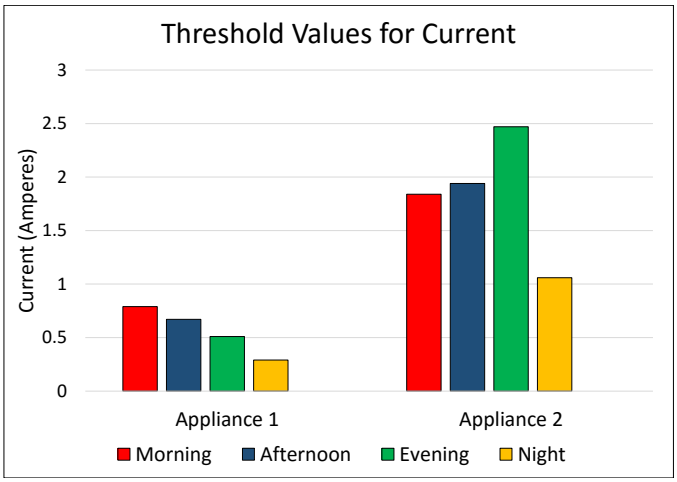


Fig. 5. Appliances threshold values for monitoring current

An example of threshold values for two appliances during summer weekdays is shown in the Figure 5, calculated using the historical data of the specific device. As can be seen, both appliances have lower usage at night indicating smaller threshold values for current whereas appliance 1 has higher usage during mornings compared to appliance 2, which has a peak during evening time. A rule can be defined which compares the average current taken by an appliance over the specific time period to compare it with the expected readings for that time context.

In summary, the same data flow applies to this use case as for the Madrid Transportation use case described earlier. The main difference lies in how the historical data is analyzed (event classification versus anomaly detection).

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed and implemented an architecture for extracting valuable historical insights and actionable knowledge from IoT data streams. Our proposed architecture supports both real-time and historical data analytics using its hybrid data processing model. We implemented our proposed architecture using open source components optimized for large scale applications. The feasibility of the proposed architecture was demonstrated with the help of real-world smart city use cases for transportation and energy management, where our proposed solution enables efficient analysis of streaming data and provides intelligent and automatic responses by exploiting large historical data.

We implemented a version of the Madrid Traffic use case on the IBM Bluemix platform, together with collaborators from the IBM Bluemix Architecture Center. Bluemix is IBM's PaaS offering, providing microservices for the main components used in our *hut* architecture instance (Node-red, Apache Kafka, Apache Spark and OpenStack Swift). Source code for this

implementation is available for experimentation and adaptation to other IoT use cases [35]. This demonstrates the amenability of our architecture to the microservices model, and provides tools to the community for further research. In addition, our work led to the development of a bridge connecting Message Hub (the Bluemix Kafka service) with the Bluemix Object Storage service [36].

Our experiments using the hut architecture extend existing solutions by providing simple but integrated batch and event processing capabilities. Our implementation applies to both transportation and energy management scenarios with only minor changes. Given the generality of the proposed architecture, it can also be applied to many other IoT scenarios such as monitoring goods in a supply chain or smart health care. A major benefit of adopting such an architecture is the potential cost reduction at both development and deployment time by using a common framework for multiple IoT applications and plugging in various alternative components to generate variations as needed.

In future, we aim to evaluate our architecture on additional IoT applications where knowledge about complex events can contribute to more innovative and automated solutions. Furthermore, we intend to improve the process of automatic generation of threshold values by considering other machine learning algorithms. We also plan to make our system more context-aware by ingesting and analyzing social media data.

ACKNOWLEDGEMENTS

The research leading to these results was supported by the European Union's FP7 project COSMOS under grant No 609043 and European Union's Horizon 2020 project CPaaS.io under grant No 723076.

REFERENCES

[1] R. Sakhuja. (2016) 5 Reasons why Microservices have become so popular in the last 2 years. [Online]. Available: https://www.linkedin.com/pulse/5-reasons-why-microservices-have-become-so-popular-last-sakhuja

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[3] A. Hadoop, "Hadoop," 2009.

[4] D. Guide, "Amazon elastic mapreduce," 2010.

[5] Amazon EC2 - Virtual Server Hosting. [Online]. Available: https://aws.amazon.com/ec2/

[6] Amazon S3 AWS Cloud - Simple, durable, massively scalable object storage. [Online]. Available: https://aws.amazon.com/s3/

[7] OpenStack Sahara. [Online]. Available: https://wiki.openstack.org/wiki/Sahara

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[9] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[11] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2187671.2187677

[12] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2015.

[13] Simplifying the (complex) Lambda architecture. [Online]. Available: https://voltdb.com/blog/simplifying-complex-lambda-architecture

[14] Questioning the Lambda Architecture. [Online]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[15] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 15:1–15:58, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1541880.1541882

[16] A. Akbar, F. Carrez, K. Moessner, and A. Zoha, "Predicting complex events for pro-active iot applications," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec 2015, pp. 327–332.

[17] Node-RED, "Node-RED: A visual tool for wiring the Internet of Things ," http://nodered.org//, 2016, [Online; accessed 6-May-2016].

[18] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[19] Pinterest Secor. [Online]. Available: https://github.com/pinterest/secor

[20] OpenStack: Open source software for creating private and public clouds. [Online]. Available: https://www.openstack.org/

[21] CloudFiles: Scalable cloud object storage. [Online]. Available: https://www.rackspace.com/cloud/files

[22] OpenStack Swift Documentation. [Online]. Available: http://docs.openstack.org/developer/swift/

[23] Apache Parquet Documentation. [Online]. Available: https://parquet.apache.org/documentation/latest/

[24] J. Kestelyn. (2016) Benchmarking Apache Parquet: The Allstate Experience. [Online]. Available: https://blog.cloudera.com/blog/2016/04/benchmarking-apache-parquet-the-allstate-experience/

[25] Softlayer - API Operations for Search Services. [Online]. Available: http://sldn.softlayer.com/article/API-Operations-Search-Services

[26] Elastic Search github repository. [Online]. Available: https://github.com/elastic/elasticsearch

[27] Welcome to Apache Lucene. [Online]. Available: http://lucene.apache.org/

[28] Learn More About Solr. [Online]. Available: http://lucene.apache.org/solr/

[29] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *JMLR*, vol. 17, no. 34, pp. 1–7, 2016.

[30] A. Akbar, F. Carrez, K. Moessner, J. Sancho, and J. Rico, "Context-aware stream processing for distributed iot applications," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec 2015, pp. 663–668.

[31] S. Petrovic, "A comparison between the silhouette index and the davies-bouldin index in labelling ids clusters," in *Proceedings of the 11th Nordic Workshop of Secure IT Systems*, 2006, pp. 53–64.

[32] A. Margara, G. Cugola, and G. Tamburrelli, "Learning from the past: Automated rule generation for complex event processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 47–58. [Online]. Available: http://doi.acm.org/10.1145/2611286.2611289

[33] A. Akbar, A. khan, F. Carrez, and K. Moessner, "Predictive analytics for complex iot data streams," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2017.

[34] T. Hargreaves, M. Nye, and J. Burgess, "Making energy visible: A qualitative field study of how householders interact with feedback from smart energy monitors," *Energy policy*, vol. 38, no. 10, pp. 6111–6119, 2010.

[35] Real Time Traffic Analysis Sample Application. [Online]. Available: https://github.com/cfsworkload/data-analytics-transportation

[36] P. Ta-Shma. (2017) End-to-End IoT Data Pipelines in IBM Bluemix: Introducing the Message Hub Object Storage Bridge. [Online]. Available: https://www.ibm.com/blogs/bluemix/2017/03/end-to-end-iot-data-pipelines-introducing-the-message-hub-object-storage/