Distributed Processing and Analytics of IoT data in Edge Cloud

Rola Alhalaseh

Helsinki December 12, 2018

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author
Rola Alhalaseh

Työn nimi — Arbetets titel — Title

Distributed Processing and Analytics of IoT data in Edge Cloud

Ohjaajat — Handledare — Supervisors
Kimmo Hätönen, Sasu Tarkoma, Hannu Toivonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	December 12, 2018	49 pages + 9 appendices

Tiivistelmä — Referat — Abstract

Sensors of different kinds connect to the IoT network and generate a large number of data streams. We explore the possibility of performing stream processing at the network edge and an architecture to do so. This thesis work is based on a prototype solution developed by Nokia. The system operates close to the data sources and retrieves the data based on requests made by applications through the system. Processing the data close to the place where it is generated can save bandwidth and assist in decision making. This work proposes a processing component operating at the far edge. The applicability of the prototype solution given the proposed processing component was illustrated in three use cases. Those use cases involve analysis performed on values of Key Performance Indicators, data streams generated by air quality sensors called Sensordrones, and recognizing car license plates by an application of deep learning.

ACM Computing Classification System (CCS):
C.2.1 → Network Architecture → Network Design Principles
C.2.1 → Software and its engineering → Distributed systems organizing principles → Cloud computing
C.2.3 → Network Services → Network Monitoring

Avainsanat — Nyckelord — Keywords
IoT, Storm, Publish/Subscribe, Edge

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

Appendices

1 Introduction

Telecommunication networks facilitate exchanging information between content providers and the network subscribers [34]. This information can be in the form of voice, video or data [76]. In recent decades, there has been an accelerating demand for services and an increase in traffic in these networks. This is due to the increased numbers of users of mobile devices and competitive businesses all over the world.

The network of IoT is predicted to have around 50 billion devices connected by the year 2020 [50]. This is double the number of connected devices in the year 2015. This growth means that more data is transferred through cellular networks. To meet this explosion in traffic, new generations of mobile networks are continuously being developed to achieve higher bandwidth and shorter delays.

In the current 4G networks, there has been an improvement in data rates reaching 50-100 Mbps. The future 5G networks extend from connecting users to also having IoT devices connected to the global IP network [72]. The 5G research aims to improve the scalability, connectivity and the speed of transferring the data over the network. 5G research further addresses challenges in providing massive coverage and ability to maintain the huge number of connected subscribers' devices [57]. The 5G research is also involved in the process of managing these networks and safely transmitting data and data streams over these networks. Mobile edge computing (MEC) is one of the main technologies where 5G plays a major role [14] as it provides faster connectivity rates and fewer delays, especially when transmitting data from IoT devices.

Cloud computing is a network and computing model where centralized storage, compute power and other resources are available on demand and accessed over a network in a self-service fashion [7]. By introducing the concepts of edge and fog computing the cloud moved towards a more distributed approach [55]. Edge cloudlets are distributed nodes in the network connected to a central cloud. IoT devices are data sources and they usually connect to the edge of the network [18].

In the edge computing model, the processing happens close to the data sources. This speeds up the computations since the data does not have to travel all the way to the central cloud before being processed. However, these edge cloudlets are usually limited in resources. For this reason, efficient ways to use the available resources are required.

Sensors in IoT devices create and send continuous flows of data or in other words, data streams [19]. Processing the data is an important step as it prepares the data for further analysis. Fast processing allows using the data for time-sensitive applications such as real-time alerts.

There are three main points to consider when collecting and processing data from sensors.

1. How to minimize the chances of failure in sensor readings and failure in maintaining the connection to the network since these failures can affect the quality of data collected from the sensors and how it further lead to inaccurate results and failures in data collection at the data sources.

2. How to process sensor data to make it ready for further analysis and reports. Analysis can be visualizations; summaries of measurements for a certain time window and recommended action to take (i.e. restart sensor, sensor low on battery, or some sensor is failing to produce readings).

3. How to control and synchronize processing of data and data streams in real time in a distributed environment. This distribution adds complexity and overhead but it allows for scaling.

Data streams are created when IoT devices at the edge of the network continuously produce and transmit data which arrives in the network at high rates [19]. The data in those streams can be of different types such as temporal data, videos from surveillance cameras, information on monitoring of a system, event logs and discrete signals.

Stream processing tasks are usually executed on the cloud. This is because the services running on the cloud have the ability to allocate more resources to meet the needs of the application workload. This thesis studies the efficient processing of data streams at the edge cloud given limited resources. The goal behind this is to be able to provide quick analysis and comparison results without the need to store the data.

This work will be enhancing a messaging prototype solution developed by Nokia. This system is a data streaming solution for network monitoring on edge cloud. The system operates close to the data sources and retrieves data based on requests made by applications through the system. In this work, these data streams provided by sensors in the IoT environment will be processed before they are sent to the

applications. I present three use cases two of them are empirical and the third is theoretical. The first use case involves the values from KPI calculation. The second use case involves live data generated by air-quality sensors where streams of measurements will be processed as soon as they are collected on the far edge; and the third one is a conceptual use case that involves an application of deep learning where the model will be trained at the central cloud and be used at the far-edge. In the three cases we will be processing the data given limited resources and without the need to store or archive the data.

The chapters of the thesis are structured as follows. Section 2 covers background concepts to help understand the system and the added component better. Section 3 illustrates the design and implementation of the solution. Section 4 illustrates the use cases. Section 5 presents an evaluation of the analysis. Section 6 discusses existing challenges and future work on the component as well as the conclusion of the findings.

The target of this work is to introduce a component design that helps in processing data streams, produced by the IoT devices, on the far edge. This work also observes the efficiency and feasibility of this processing given the limited storage and computation resources on the far-edge. Furthermore, the process happens without the need to store historical data.

By processing the data on the far edge, research questions involve empirically finding the best approach to reduce the amount of data transferred after processing as well as the number of times the data has to travel back and forth. Success in those will help in saving bandwidth, speeding up the processing and the production of faster results

2 State of the art

This section highlights the most important concepts to ease the understanding of the research problem. At first, there is a part on the advancement towards the 5G networks, their expectation, challenges and management. Afterwards, there is a presentation of the publish/subscribe architecture. This also includes Apache Kafka and the solution developed by Nokia. Then, we present the data streaming concept including Apache Storm architecture and different data stream processing solutions that implemented Storm as part of their models such as Lambda and Kappa.

2.1 Moving towards 5G Networks and the impact on network management

Telecommunication networks are a collection of connected nodes interacting with each other through links [1]. Cellular networks are one example of these networks where the last linking factor between the network and the subscriber is wireless. Cellular networks consist of several interconnected base transceiver stations (BTS) that cover large geographical areas. These geographical areas are divided further into sections called cells. Each BTS covers a few cells [47].

Generations of mobile networks have been evolving during the past decades to meet demands for higher bandwidth and shorter delay times when transmitting data over these networks [50]. In 4G and 4G LTE, the data rates improved significantly and the data rates are 50-100 Mbps, which enabled sharing larger amount of data through the network and fewer delays in data transmissions.

Base stations and sending packets through the network using an IP-based technology were also part of the 4G evolution [49]. Long-term evolution or LTE is a type of 4G that uses an air-interface called Orthogonal Frequency Division Multiplexing (OFDM) which is faster than normal 4G and can support a wider variety of users [65]. LTE focused on closing the gap between cellular networks with high mobility requirements and fixed wireless Local Area Networks with high bandwidth [65].

LTE advanced is a communication standard that enhanced LTE by providing higher bandwidth and data rates that reached up to 3 Gbps for downlink and 1.5 Gbps for uplink to accommodate more simultaneous active subscribers and aiming for cost efficiency of resources [4]. Furthermore, LTE advanced improved the performance at the edge of the cellular network.

Evolution from generation to generation included hardware improvements that supported each generation. Yet, the 5g evolution is more software oriented in managing the network [51] and it gathers many technologies, scenarios and use-cases [49].

The future 5G networks aim to extend the capacity of the network extending from connecting users to also having IoT devices connected to the cloud. The latter is referred to as massive machine-type communication.

5G research aims to improve the connectivity, security and communication [49] [59] [75] when handling data streams. It also aims to provide high scalability and data rates (up to 100mbps) [49]), reduce energy consumption [50] and delay times to support the network's services' integrity, availability and reliability due to the new emerging use cases [54]. For those reasons, 5G plays a major role in Mobile edge computing (MEC) [14].

5G is targeting a variety of use cases. These include supporting enormous density of mobile devices and ultra-low latency in user's communications over the network. Another use case is controlling and monitoring vehicles traffic. There is also a use case of monitoring an industrial process. All these use cases place high demands on the dependability of the network. Furthermore, human safety and even human lives depend on the availability and integrity of the network service.

2.2 Publish/Subscribe Model

Publish/subscribe is a messaging model where there are two main entities: the publishers and the subscribers [24]. The publishers are responsible for creating events. Those events are then consumed by the subscribers. The subscriptions are managed through a mediator that is responsible for waiting to receive updates from the publishers and sending those updates later to the list of subscribers. Two subscription models exist in defining the publish-subscribe scheme: the topic-based, and the content based.

In the topic-based subscription model, the relationship between publishers and subscribers is determined by the attributes set for the topic. The publishers generate data-streams that are organized into topics. The data fetching component of the system subscribes to one or more topics and will receive updates and responses based on which topics it subscribed to. The subscribers would then be consuming the whole topic to which they are subscribing and there are no restrictions on which subsets or parts of the topics updates they wish to receive.

The second model is the content based subscription, that is more specific than topic-based. The subscribers, in this case, are interested in certain subsets or parts of the events and for that, they would define filters for the part of the event's contents [13]. In this case, this shows more expressiveness but it is more difficult to implement. This is because the correlation between the publisher and the subscriber has to be computed before the event is created to define which content the subscriber will receive. A good example of a content-based publish-subscribe model is the one developed by IBM Gryphon [16, 67].

IBM Gryphon uses simple routing approach in which case routing tables are used and all the message brokers should be aware of all active subscription requests. Since all brokers need to know about all subscribers their routing tables can easily grow large. This limits the scalability and expressiveness of the IBM Gryphon model [52].

In a publish/subscribe architecture, both publishers and subscribers are fully decoupled in space, time and synchronization [22]. Space decoupling means that both publishers and subscribers are separated from each other and they do not have any knowledge about one another. In the prototype solution mentioned in the next section, e.g., both entities communicate by sending and receiving subscription requests through the prototype solution pipeline. Therefore there is no direct contact between them.

Time decoupling implies that publishers and subscribers can be active at different points in time. Events will be issued anytime by publishers and delivered to the mediators within the system. The mediators will hold these events until they expire or until the subscribers are available to consume them.

Synchronization decoupling means that publishers generate data regardless of whether there are subscribers or not. Furthermore, subscribers in synchronization decoupling can also subscribe to topics or issue subscription requests even if the topics are not there yet.

The prototype solution developed by Nokia, discussed in the next section, is based on the topic-based publish/subscribe model. The following is a more detailed illustration of the functionality and components of the prototype solution through which the subscription requests are made and answered.

2.2.1 Prototype solution

The proposed system is part of an ongoing research project conducted at Nokia Bell Labs, Espoo, in collaboration with the University of Helsinki's Department of Computer Science. The concept of the proposed system relies on the publish/subscribe architecture. The proposed system acts as the middleman in the network management plane of a cellular network.

The system operates at the edge cloud and handles transmitting events between publishers and subscribers. The publishers operate close to data sources and they are responsible for providing data for the subscribers issuing subscription requests. The subscribers are any kind of applications that need data in a certain form in order to be able to make decisions or show some analytics results. Those subscribers make a subscription request through the prototype solution.

Subscribers to the network issue subscription requests to events through the prototype solution. The prototype solution targets the publishers (i.e. the network elements) that are publishing those events to respond to the subscription requests. Data sources here are the publishers in this application of the publish/subscribe architecture. In response to subscription requests, the publishers issue a subscription response through the system. The process is illustrated in Figure 1.

The prototype aims to minimize the number of times the same data has to travel over the same link which is from the network elements (i.e. the publishers) to the subscribers. For instance when there are multiple subscribers requesting the same data then the solution prototype is able to route this data to all these subscribers by only requesting this data once and without having to generate a new request to fetch this data from the publishers each time. Furthermore, the prototype maximizes the amount of up to date information in the subscriptions [35] by feeding those updates to the subscription requests.

The aim is to process data streams coming from the data sources and that has been collected by the fetcher component. Application of this analysis will take into consideration the following three use cases: Calculations of Key performance indicators (KPIs) of network elements, processing of data streams generated by air quality sensors, and a demonstration of how to apply deep learning to the solution to recognize car license plates.

The architecture of the prototype solution consists of four main components. Those components are data hub, data switch, data fetcher and the global repositories [35].

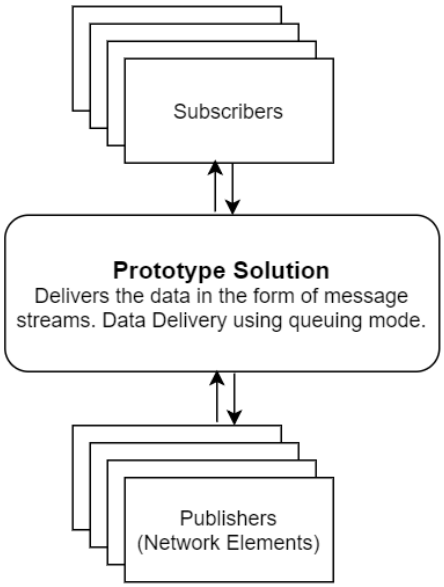

Figure 1: **Prototype Solution Architecture.** *Components of the management plane of the proposed system. An illustration of data transmission in publish/subscribe architecture.*

The global repositories help in controlling the system. During the operation of the system, an application makes a subscription request for data. This data is found at one of the network elements in the management plane. The data hub component in the system receives the subscription request. The data hub is the interface for serving applications and acts as the subscriber to the events published by the data fetchers.

The subscription requests include information on what events the application is subscribing for and how frequent this application wishes to receive subscription responses or updates on the subscription. The data switch component, implemented using Apache Kafka [35], acts as the broker that regulates the data flow in the system.

The data switch receives the subscription request and routes it to the proper data fetcher. If another application makes similar requests, the data switch will mirror that request and send it as a response to the requesting data hub. This hub then sends the response data back to the subscriber. This speeds up the process of data retrieval. When the data switch forwards the request to the data fetcher, the network elements will start to emit this data.

As we can notice in the structure of the prototype solution, the fetcher component

operates next to the data sources. The data fetcher then will perform computations or analysis on the response data and then send the response to the data switch. The data switch continues to route that published response to the data hub and then to the subscribers. There are two pipelines in Figures 2 and 3. The pipeline in Figure 2, illustrates the flow of the response to the subscriber request from the network elements. Network elements here represent the publishers of the events. Figure 3, illustrates the flow of the subscription request from the subscribers through the system.

The prototype solution has two different modes of communication: direct mode that relies on connecting the applications to the network elements by sharing their connection details. The other mode of communication is the publish/subscribe mode which is the focus of this work. For further details on the system architecture, the work in [35] describes the system and its components in more details.

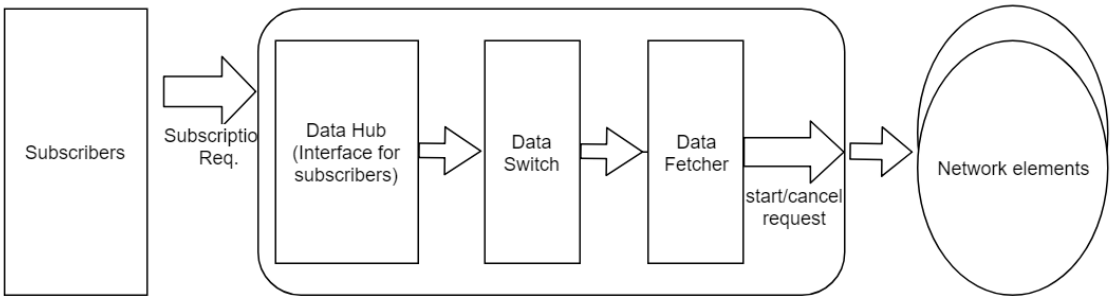

Figure 2: Subscription request flow pipeline through the proposed system

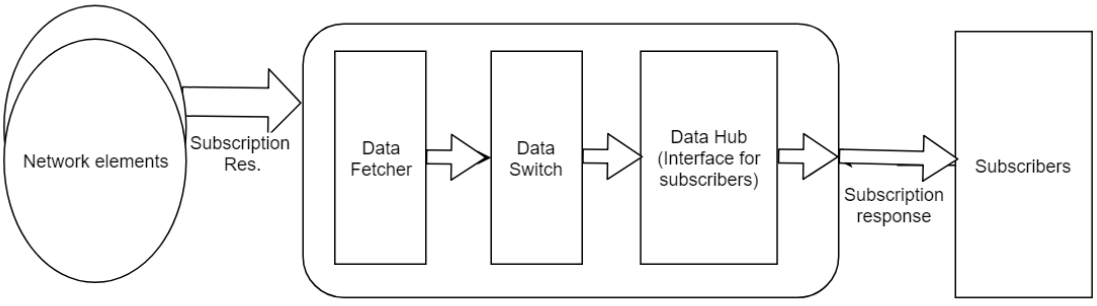

Figure 3: Subscription response flow pipeline through the proposed system

Data streams such as raw counter values and system logs are the kind of data fetched by the fetcher component of the prototype solution. Next part will be discussing the processing of data streams in more detail.

2.3 Processing of Data Streams

IoT devices and data sources generate continuous flows of data that arrive at high speed to the data collection points [19]. These data sources are connected to the edge of the network and are usually distributed at various geographical locations. The data flows generated by those systems arrive in sequential instances called data streams [26]. Data carried in those streams can, for example, be temporal data, videos from surveillance cameras, system monitoring information, event logs or discrete signals. In order to make sense of this data, several solutions have been developed for processing data streams.

Data stream processing, is the process of performing computations on data streams the moment they arrive at the data collection point of the system. The intention of this processing is to clean, transform and prepare the data to make it ready for further analysis and decision making, for example, anomaly detection, and it can also result in a new data stream. Examples of data stream processing (DSP) solutions are Spark streaming [10], World Wide Streams (WWS) [38], Apache Storm [11] and Apache Flume [8]. For this work, Apache Storm influenced the design of the proposed processing component architecture. Storm is explained in more detail in the next section.

Applications of IoT data streams include network monitoring, collecting data from sensors for environmental monitoring (such as detecting gas leaks, alerting of high CO2 levels and detecting anomalies in the readings produced by sensors) and many other applications. Incoming data streams can be classified into historical and real-time data. Both kinds have their own properties and use-cases.

Real-time refers to the recent data at the moment of arrival to the system. When processing the real time data there are limitations in storage and computational power. In real-time or near-real time processing, the result must be produced in a timely fashion to meet the time constraints within which an action should take place. Otherwise the result will no longer be of use. In cases where the incoming data streams are arriving at high rates, the computation results might be estimated values. However, in some systems, once the processing of the real time data is completed it can also be archived.

Historical data is the data that has been collected and stored over time. There can be the data store that is constantly being fed by the real-time data and grows in size over time. In the case of historical data there are no restrictions in terms of

storage and time to process the data. Retrieving query results from historical data is often slower because they are derived from a long span of actual data points. The results taken from this data can help in pattern detection, data mining and decision making.

Processing real-time data streams can be useful in several domains as they can point out patterns, detect failues in devices and gives insight on the overall state of the sensed environment [19].

Table 1: Comparison between real-time and archived data.

Feature	Archived	Real-time
Query	Many	One or few
Memory	No restrictions	Limited
Results	Actual	Estimated/Actual
Processing	Slow	Limited

Let's assume we have a stream of N bits and we are trying to count the number of 1's or 0's in that stream. In another scenario of a stream of N elements we might want to retain the sum of all the N elements in the stream and continue adding values as they arrive. Since we have a storage limitation, solving these issues present a challenge in processing data streams at the edge, unless we can do this without having any storage by maintaining a variable with an incremental value in memory and discarding the raw values after calculation. Having no storage might increase chances for error but this is a compromise since we want to do this processing to get results within a certain time constraint and the given storage and computational limitations.

Data streams are basically time series tuples. Each tuple arriving within the stream has a time-stamp. Incremental computation on mean, standard deviation, and correlation coefficient is possible with numerical values of data streams [26]. For the mean value, number of observations must be maintained in memory as well as the aggregate sum of the values in the stream. For standard deviation we also need to increment the number of observations as well as the aggregate sum of the observations as well as the squares sum of the observations. Correlation coefficient is useful when having two data streams and we need to find the linear interdependence between the two. For calculating the correlation coefficient we need again the incremental number of observations, sums, squared value sums as well as the sum of the cross product for both streams needs to be aggregated.

Certain kinds of statistics can be maintained for the above by constantly updating values for the calculations. Data can lose value over time. With this approach we cannot tell old data from new data This issue brings us to the concept of using time windows to process data streams.

Time windowing means that only those values, whose timestamp falls within the range of the time window, are included in the computation.A value may be computed many times in different time windows. A challenge in case of two streams is to define a time window that would fit both streams. Three approaches to time window were mentioned in [26] that are relevant when handling data streams: Landmark windows, sliding windows and tilted windows. Landmark windows will be used in this work and more explanation will be provided in the 'design and implementation' section.

Reduction techniques such as sampling and summarization techniques, such as histograms, data synopsis and wavelets, are applied on data streams to help compress those streams and reduce the size of the data that is then sent over the network. These techniques save bandwidth and increase the speed accordingly. However, the accuracy of the computation result or the summary provided are usually affected by the size of the time window from which the summary was obtained. Therefore, tuning the size of the time windows can help in achieving more accuracy. Increasing the size of time window can be at the cost of increasing the processing that needs to be done over that time window, more needed space and longer delay due to the increased processing.

2.3.1 Apache Kafka

Apache Kafka [9] is a real-time, fault-tolerant and durable messaging solution that follows the publish/subscribe architecture. Kafka open-source model is also used in distributed messaging. Kafka is able to handle large amounts of data and is used with data stream processing systems such as Apache Storm to help in distributing, delivering and routing the incoming data. As described in Figure 4, some of the components in the prototype system are built on Apache Kafka for delivering subscription messages between the publishers and the subscribers.

There are four main components in Kafka: the topics, the brokers, the publishers and the subscribers. Kafka pipeline is found in Figure 4. Publishers produce data streams that are received by Kafka which categorizes them to topics based on the category of this data stream. Those topics are further divided to partitions of

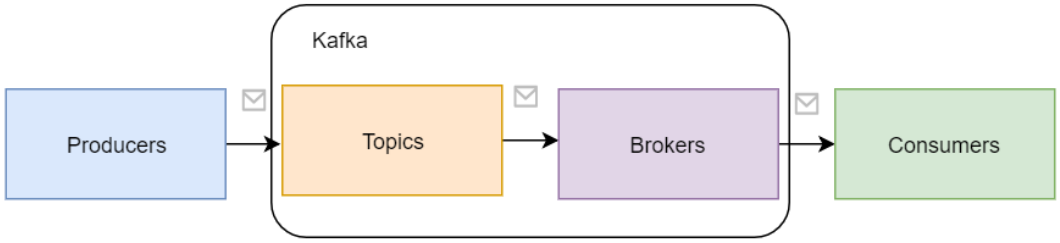

Figure 4: **Kafka components.**

equal sizes, each having a unique identifier. Each topic can have many partitions. Partitions are further replicated to avoid losing data. Brokers have the responsibility of maintaining published data and later sending it to the subscribers. Each broker can also have one or more partitions.

A subscriber wishing to receive contents of a topic initiates a subscription request through the system for that topic. Apache Spark or Apache Storm can then be used to process the data streams that are fetched from the publishers of the data streams, IoT sensors, for example. The term data stream will be used instead of topic to represent the events generated by the publishers and fed to the topic in the solution prototype.

2.4 Apache Storm

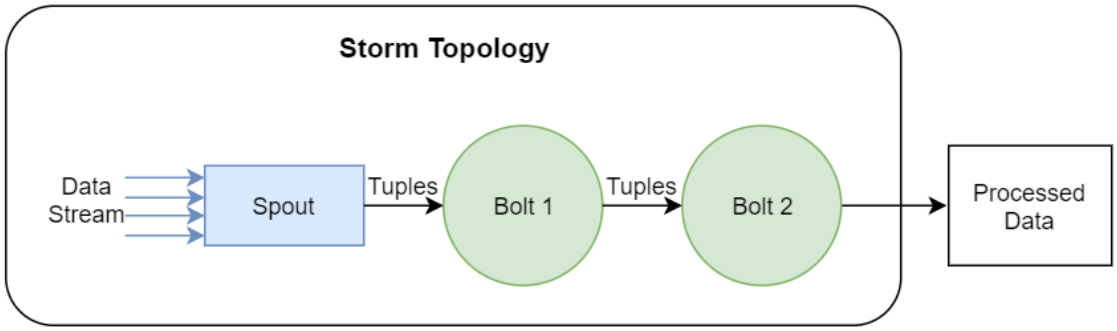

Figure 5: **Storm Topology.** *A simple Storm Topology consisting of one spout and two bolts.*

Apache Storm is a distributed computation system for processing unbounded streams of data [11, 23]. This processing happens at the moment of the data arrival. Data models used as input in Apache Storm are tuples and streams.

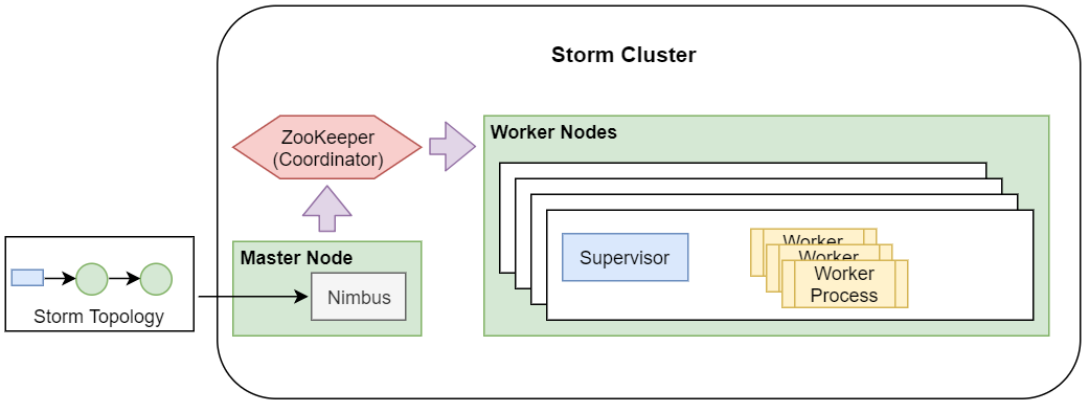

Figure 6: **Storm Cluster.**

The processing model followed in Apache Storm is Directed Acyclic Graph (DAG) [56]. The DAG model means that there is a path between any two vertices without any cycles. Storm consists of topologies (See Figure 5). Each topology consists of two main building blocks that are connected, i.e., a spout and bolts [21]. The topology specifies the connections between spouts and bolts.

The spout is responsible for receiving data streams from input sources or from message brokers such as Kafka, which helps the spout consume the incoming stream. The spout then transforms the stream of data to stream of tuples. These tuples are then forwarded to the bolts. Each bolt that processes the incoming tuple and sends it forward to another bolt. The last bolt outputs the processed stream to another layer in the processing pipeline.

Bolts are the processing units in the topology and each bolt is responsible for one processing task. The processing logic can include joins, filtering or aggregation operations on the incoming tuples. While processing the tuples, no information is stored about their status. Therefore, Storm is a stateless processing system.

Topology is run by the Storm cluster (see Figure 6). This cluster follows a master-slave architectural pattern and has the topology as the input to it. A daemon process called Nimbus runs on the master node and it is responsible for the assignment, distribution, monitoring and reassignment of tasks (i.e. responsibility for different elements defined in the topology) to worker nodes. Those tasks are the spouts and bolts within the topology. Figure 5 illustrates a simple Storm topology consisting of one spout and two bolts. The spout is the entry point of the data stream. As the data stream is transformed to tuples those tuples are sent to Bolt 1 which performs

some processing task and then sends the tuples to Bolt 2 which performs another processing task and then sends the completed data on its way through the rest of the pipeline.

Storm cluster in Figure 6 contains one or more worker nodes. Each worker node consists of one supervisor daemon and multiple worker processes. The supervisor communicates with the master node through ZooKeeper which facilitates the communication between the master and slave nodes. After the topology is ready it is sent to Nimbus daemon. Nimbus downloads the code of the topology locally and communicates with a supervisor to assign tasks to the worker processes. Nimbus is responsible for distributing the tasks to the supervisor of a worker node through zookeeper. The number of the worker processes is determined by the number of tasks (i.e. the spouts and bolts).

The supervisor gets information about the topology from zookeeper and copies them to a local file system. The supervisor assigns each bolt and spout in the topology to a worker process and then starts those processes where each process is responsible for a single bolt or spout task. Each worker node can have multiple worker processes. Tasks will continue running until a topology is killed and the Nimbus does the monitoring of the tasks through zookeeper that acts as the coordinator.

Storm has been one of the popular data stream processing system. Storm has been implemented in architectures such as Lambda and Kappa that are solutions for processing data in real time. The following two sections illustrate both Lambda and Kappa architectures in details. Those architectures has influenced the proposed solution component described in the third section of this thesis.

2.4.1 Lambda architecture

Lambda architecture is a cloud-based solution for building stream processing applications on top of stream processing systems such as Storm [71]. Lambda operates in a scalable and distributed fashion to process and store data streams [73].

Analyzing data streams coming from sensors and other IoT devices is a challenge when operating at the edge cloud due to limitations in resources, the need to get results fast, and the challenge to discover the resources at the edge [53]. For that, it is crucial to find the proper environment. Regarding the discoverability of resources, the prototype solution in this thesis is already connected to the data sources and this work will study different issues related to the connectivity of sensor devices and the

Table 2: Comparison between Apache Spark and Apache Storm

Feature	Storm	Spark
Latency	Lower latency	Higher latency, up to few seconds
Throughput	Higher throughput, achieved with some limitations imposed	Capable of handling higher throughput
Languages	Available in Python, Java, R, JavaScript, Ruby and Scala	Available in Python, Java, R and Scala
Stream primitives	Incoming stream is transformed to tuples	DStream is the input stream primitive
Incoming stream source	Spout	The network or HDFS
Model reliability	Supports different modes of reliability including at least once	Spark supports exactly once
Computation units	Bolts are the units responsible for the computations	Relies on windowing

prototype solution. The data fetching component already operates at the network edge close to the data sources.

Data streams are classified into two categories: 1) real-time data flowing into the system and 2) the historical data that has been collected and stored [27]. This thesis deals with the processing of real-time data only.

A serverless computing model is a cloud-based approach that enables the definition of functions to a shared pool of cloud-servers. This makes the task of managing the server no longer necessary [28].

The lambda architecture consists of Three main layers that are the batch, the serving and the speed layers [27]. The batch component uses Hadoop which is usually used for processing batches of stored data and aggregates new data once it arrives. This gives more recent insights on the data. The speed and the serving layers both use Storm. The serving layer performs indexing operations on the data in the batch layer to speed up the querying operations, and this is not covered in the scope of this work. Querying operations happen on both the batch and speed layers. Furthermore, the

query result is merged to get a view on the complete data that is near real-time.

Apache Hadoop is an open source framework for distributed processing of historical data. It uses Hadoop file system HDFS which is a set of shared libraries and utilities. It is based on a programming model called MapReduce. It has widely been used in processing archived data. In the Lambda architecture, it is used in the batch layer to process both real-time and batch data. MapReduce splits the input data set into independent chunks that are then mapped to worker nodes and processed in parallel. Once the worker nodes complete the processing with a collection of sub results they are combined and reduced to a final result.

There are two challenges when using the lambda architecture, the first is completing the tasks in real-time, i.e., within a time constraint. The other is the repetition of business logic for both the batch and real-time data stream processing and that was approached in [36] where they suggested a language abstraction that works over both kinds of data. The proposed component in this thesis is influenced by the Lambda architecture and will avoid the need to use batch layer in the architecture and rely only on processing real-time data streams.

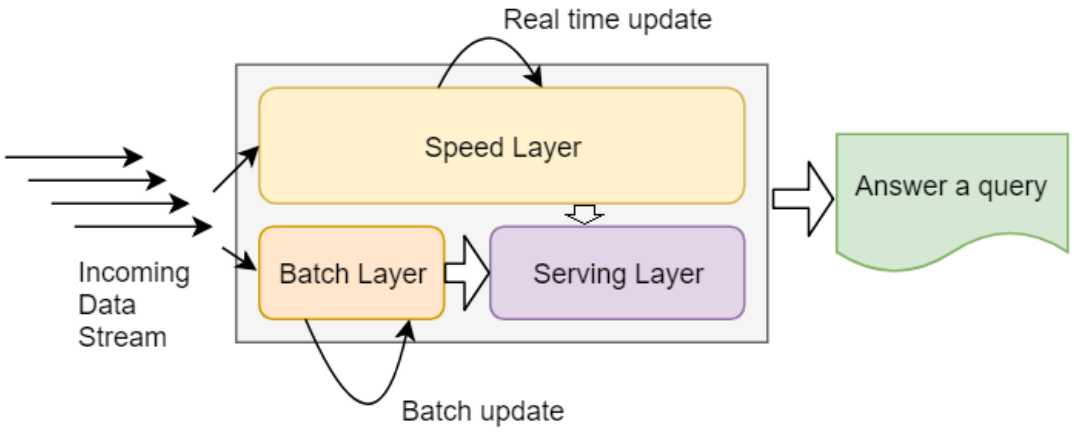

Figure 7: **Lambda Architecture.** *The figure shows the interactions between the three layers in the architecture. The serving layer is responsible from getting the real time and batch updates to compute both the real time and the batch views. The downside with Lambda is that the incoming data is copied twice.*

2.4.2 Kappa architecture

Kappa is a serverless software architecture that came after Lambda. This it is similar to the Lambda architecture but aims to reduce the amount of processing. The aim

was to overcome the shortcoming of having to copy the incoming stream to both the batch and the speed layer as in the Lambda architecture. Kappa architecture does not include the batch layer as part of the architecture and this removes the need to duplicate the data and to do batch processing.

The main goal behind Kappa architecture is to have both the real time processing of the data and the reprocessing provided by one layer rather than two. However, while the data is going through the speed layer it is also stored and preserved using some of the available data storage solutions.

Processing of the data flows (i.e, data stream) happens at the speed layer and the results are forwarded to the serving layer to provide a real-time view or to be stored. This architecture provides re-computations for the data when there is a need such as in machine learning applications.

Kappa can be used in cases where a historical view does not needed to be maintained. An example of such a use use case would be triggering an alarm when a sensor value is out of range or if some sensor is malfunctioning [58].

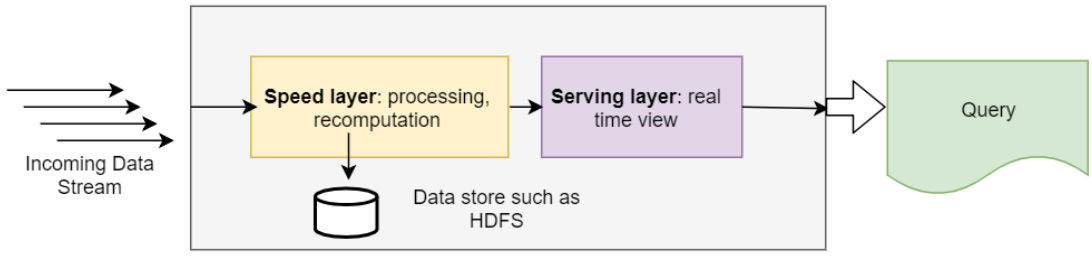

Figure 8: **Kappa Architecture.** *In the Kappa architecture there is no batch layer and the incoming stream of data is not duplicated.*

3 Solution design and implementation

This section describes the architecture design for the proposed processing component developed to enhance the Nokia prototype system. Section 3.1 describes the rationale behind the designed processing component. Section 3.2 describes the prototype solution pipeline in more detail. Section 3.3 describes the design and the implementation of the proposed processing component.

3.1 Rationale

By combining 5G technologies and Mobile edge computing (MEC) the aim is to reduce latency and increase the connection speed. Extending the network to the edge and far edge, which is an extension to the network edge, brings the processing systems closer to the data sources. 5G brings the ability to connect more devices to the network. This can further minimize the number of data transmissions for processing, save bandwidth and increase the speed of processing. The network edge spreads over various geographical locations, which disseminates the demand on the network and computation power.

Yet, the edge of the network has limitations in storage and computing resources. *Thus, the proposed processing component aims to perform the processing on the incoming streams of data without the need to store the data or send it to the central server to be processed.* The prototype solution is already implemented with a hard-wired processing capability. Therefore, this work introduces a processing component that would enhance the processing capability at the edge/far edge by enabling more distribution to the computations. A Storm like solution is proposed for this purpose. This thesis further studies the applicability of the proposed architecture in three use cases involving data streams from different data sources, i.e., counter values, sensor reading and a video/image stream. The following subsections introduce the concepts of far edge and fog computing.

3.1.1 Far edge

Far edge is a name given to portable devices or systems connected to the edge of the network. These systems are able to generate data or process it. Devices at the far edge provide a wireless interface for communicating [70]. The network edge can contain a variety of connected components [5]. Some of these components can

include micro data centers, cloudlets and smart routers [62]. Far edge, a recent emerging concept, refers to a variety of devices such as smart mobile devices, IoT wearables and sensors. Far edge connects to the edge-cloud through some form of wireless or radio connection. The name far edge is due to being further from the central cloud than what is usually considered the network edge.

By connecting to the edge the aim is to reduce latency and save battery power in the connected far edge devices. Yet, the constant data transmission between the edge and the far edge devices is inefficient. Each data transfer operation consumes power and bandwidth.

Having the computational power operate on the edge/far edge to process the incoming data streams can be more efficient in two main ways. We save resources, and we operate faster as it will take fewer trips for the data to travel to have it processed [6].

The results are then sent to the edge after processing. Same benefits apply for having computational power at the edge instead of the central cloud which also minimizes the number of trips that the data has to make and, speeds up the processing given the limited resources available at the far edge.

We used Sensordrones as data sources to generate continuous flows of data and a laptop as the far edge device. The communication with the laptop is direct and this creates a far-edge-to-far-edge (F2F) communication model. Rehman et. al discussed F2F in more detail [70].

3.1.2 Fog computing

Fog computing concept, created by Cisco, is part of the cloud computing model [44]. The focus of fog computing is to bring the computing power to the edge of the network, so as to minimize delays and bandwidth usage [15].

Fog computing is similar to edge computing in terms of bringing the computation power close to the data sources. The location of the computations is the main difference between the edge and fog computing models. Edge computations happen on devices or gateway devices that are directly connected to the sensors while Fog computations take place on the processors of the Local Area Network (LAN) or the hardware of the LAN and therefore it may be more physically distant when compared with Edge computations [29].

The obvious presence of data streaming at the edge makes it a convenient platform

for IoT devices and real-time applications since there is a possibility to distribute the computation tasks and make them execute in parallel which makes them complete in shorter time. Analyzing data streams at the edge saves bandwidth as mentioned earlier. This is because there is no need to transfer the collected data to the central network for that purpose. This also enables the calculations on the data to happen in real-time [61] and retrieve the results in time. There is a possibility to deploy the network edge computing resources either on base stations or within Radio Access Networks (RAN) aggregation points.

3.2 Existing prototype solution pipeline

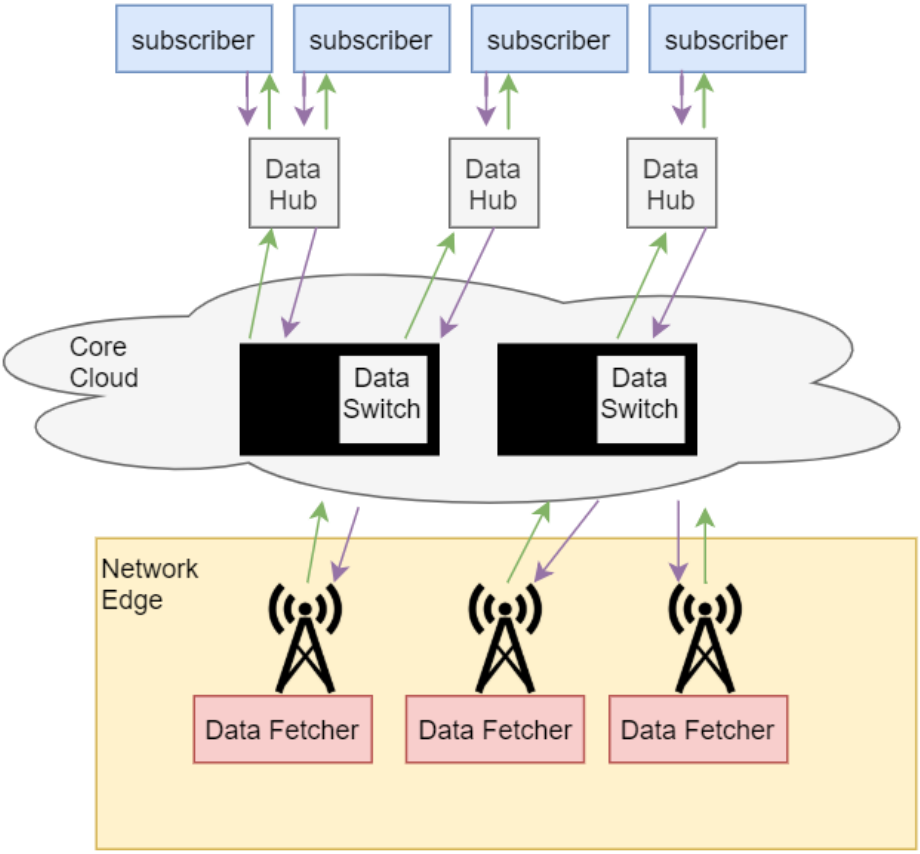

Figure 9: **Prototype solution Components.** *The arrows show the request/response between publishers and subscribers through the prototype [35].*

The prototype solution is an architecture for coordinating message delivery and message updates between publishers and subscribers in the network. The current model of the prototype provides a processing for large data streams in near real-time.

One of the main goals of the prototype solution is to minimize the data transfers over the network and increase the up to date information in the subscriptions. To achieve those goals the computations are done close to the data generation points and the analysis is performed on the data in real-time [35].

The prototype solution in Figure 9 consists of data hubs, data switches, global repositories and data fetchers. Data hubs operate close to the subscribers and are responsible for serving those subscribers. The subscribers issue subscription requests through the data hub which are delivered to the data fetchers through the pipeline. Data fetchers operate close to the publishers and send responses for the subscription requests. The processing of the data happens at the data fetcher.

Data switches receive computation outcomes from the data fetchers. Data switches act as the router and the coordinator of messages between the data fetchers and the data hubs. The prototype follows the publish/subscribe model (i.e. the data hubs act as the subscribers within the model and the data fetchers act as the publishers).

Data fetchers send the published events only once to the data switches. The data switches then route the published events to the data hubs. The data hubs will then serve the subscribers with the published responses.

Global repositories were not shown in Figure 9 and they are not in the scope of this thesis. However, the purpose behind global repositories is to assist in coordinating the subscriptions and relationships between components within the prototype solution system.

This work applies the prototype solution's fetcher component at the far edge network device. This far edge device acting as the gateway between the prototype solution and the publishers of the data, i.e., environmental sensing devices. The fetcher is assumed to be directly connected to the far edge IoT devices and the aspect of device discoverability is not covered in the scope of this work.

3.3 The proposed processing component architecture

The proposed processing component design aims to enhance the processing of data streams that are fetched from the publishers, by the data fetcher, to the prototype solution. The design of the component was influenced by the Kappa architecture for processing data streams in near real-time.

As illustrated in Figure 10, the layers of the proposed processing component are

highlighted in red. The layers of this component are the speed and the serving layers. The speed layer is implemented at the data fetcher component of the prototype solution while the serving layer is implemented at the data switch component of the prototype.

The serving layer acts as the broker to deliver the processing results sent from the fetcher. This part was implemented in [20] and will not be in the scope of this work. The main focus here is on the speed layer where the processing of the data takes place at the data fetcher side.

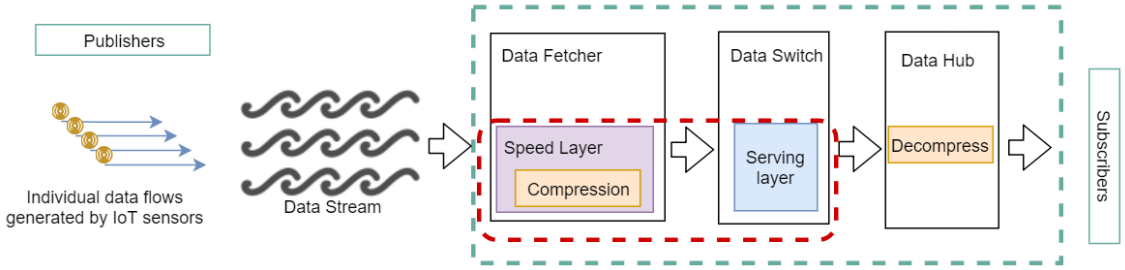

Figure 10: **Proposed component architecture.** *The processing component consists of two layers. The speed and the serving layers. The speed layer is implemented at the data fetcher, while the serving layer is implemented at the data switch.*

The speed layer detailed in Figure 11, shows a storm like model of a topology. Data sources in the network, i.e., each LTE counter or sensor individually generate a flow of data. Those continuous flows together feed the data stream and that is also illustrated in Figure 10 above.

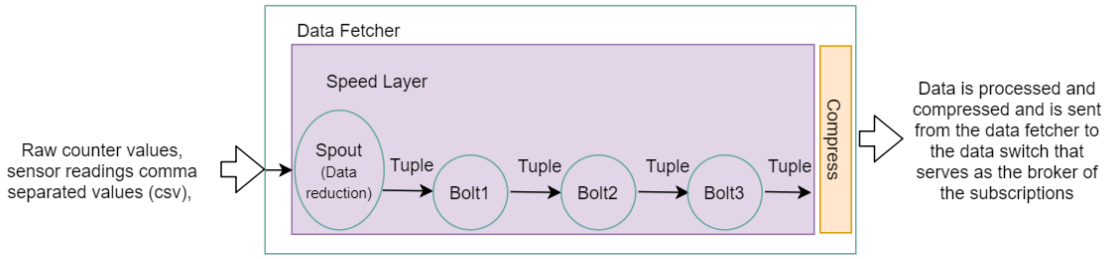

Figure 11: **Components of the computation model.** *This is a generalized model with one spout and 3 bolts.*

After the data fetcher component fetches those streams into the prototype solution, they are received by the spout component of the topology. The spout is the stream's entry point to the rest of the topology. If needed, data reduction can take place at the spout.

Data reduction and the reasoning behind it is detailed for each of the use cases. The spout processes the stream and transforms it into a set of tuples that are then sent to bolt1 for the first processing task. For demonstration purposes Figure 11 has 3 bolts and each of them is responsible for a single task.

After the topology is finished processing, the processed data is compressed for each of the subscription topics and sent to the data switch. The data switch will route it to the data hub where it will be decompressed before the subscription response is sent to the subscribers [32]. The following gives more insight on data compression and which form is suitable when dealing with different kinds of data.

Data compression

There are two kinds of compression techniques available. The lossless and the lossy compression. Table 3 illustrates the different usage scenario for each kind. Lossless compression is reversible, which means that it is possible to perfectly reconstruct the original data without losing any parts of it.

Table 3: Comparison between lossy and lossless compression techniques

Compression Type	Lossy	Lossless
Reversibility	Irreversible	Reversible
Data types	JPEG, video, sound	text, spreadsheets, GIF
Data Retrieval	Data is partially discarded during compression.	Original data can be recovered when the file is uncompressed

GZIP is an example of lossless compression algorithms. lossless compression is preferable when dealing with critical data readings such as financial data. lossless compression works better for the KPI calculations and the environmental sensing use cases and this compression happens before sending the processed data from data fetcher to the data hub where it will be decompressed before it is sent to the subscribers. What makes lossless of compression desirable is that it is reversible, which means that we can retrieve the original data content when decompressing.

As for lossy compression, it is an irreversible compression technique where parts of the data are discarded during compression and cannot be perfectly restored after decompressing the data. The main reason behind using lossy compression is that it

provides a very high compression ratio. Discarding nonessential parts of the data still often allows for a sufficiently accurate reconstruction of the original data.

Lossy compression has been used in earlier work [32] as a second compression layer for the aggregated LTE KPI values. However, the quality of compression can be assessed by using the euclidean distance. Eucleadian distance is a measure to find the similarity between the time series before the compression and after decompressing the data [33]. The lossy compression technique is a good choice for compressing images such as JPEG or video and sound and is a suitable choice for the third use case in this thesis.

Compression happens at two stages. In the case of KPI and sensor data, semantic compression is applied at a lower level in data fetcher. lossless compression algorithm GZIP is used before sending the processed data away from the data fetcher. Since the purpose of lossless is to minimize the amount of data to be transmitted and since the processing happens at the far edge close to the data collection point, it makes more sense to compress the data before transmitting it.

Once the processed data is decompressed, if it is a lossless compression, then we are sure that no parts of the processed data were discarded during the compression. However, with this approach we make sure to slightly reduce the data before processing, minimize the size of the transmitted data and retain the original data after decompression. There is an Java implementation of the GZIP algorithm for compression and decompression of data.

Figure 12 shows an hour glass representation of the prototype solution highlighting the benefits of compressing the data before sending it through the network, which is saving bandwidth resources.

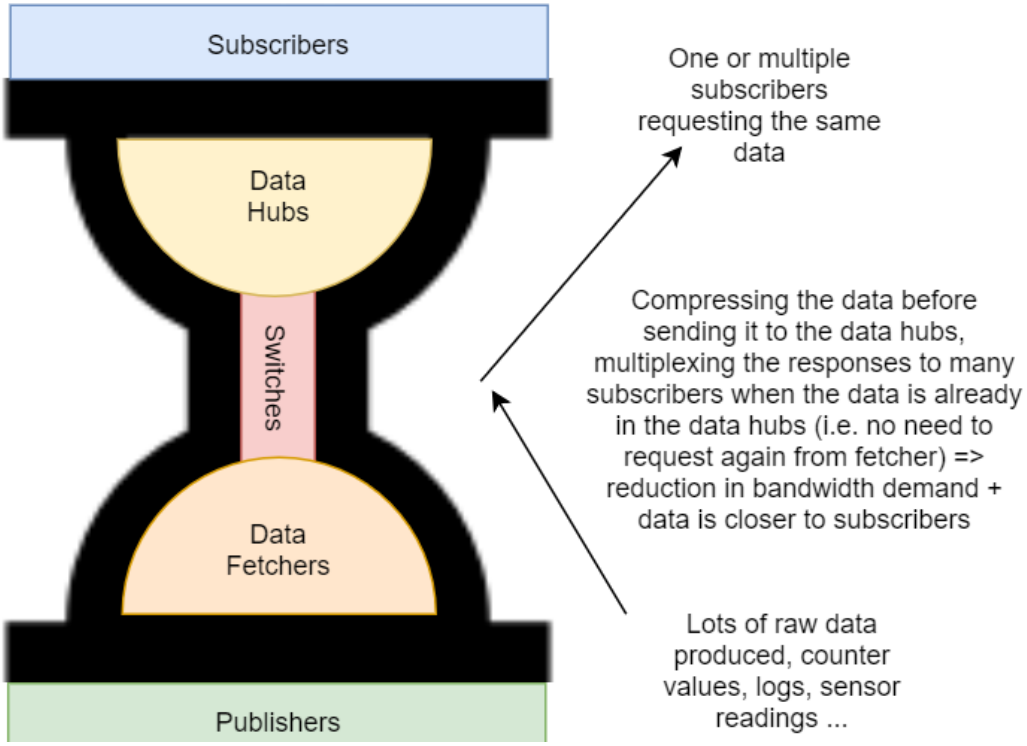

Figure 12: **Compression and multiplexing through the prototype solution.** *Reduced bandwidth demand and reduced requests to fetch the data.*

Data processing

Data reduction, i.e., semantic compression on the incoming data such as calculating KPIs from LTE counter values helped in giving a straightforward indicator on how well this base station cell is performing within the network. This also reduced the amount of data that needed to undergo further processing before sending to subscription requests.

Bolt1 receives the reduced data and splits it into non-overlapping time windows, i.e., landmark windows. In landmark windows, a defined landmark marks the end of a time window and the start of a new time window. This restarts the processing computations to include the values of the newer time window.

The choice of window size depends on the number of data samples we need to have in each window portion. For example, if we are receiving the data at the speed of one sample per 10 seconds and we need to know the overall performance every five minutes. Then, each window will perform the calculations on samples from within those five minutes.

Bolt2 receives the time windows and calculates the aggregated sum and values count for each time window and send those new tuples to Bolt3. In Bolt3, the mean is calculated by dividing the aggregated sum by the values count which gives us the average performance in that time window. After all the processing is complete, the processed data is then compressed and sent through the prototype solution to reach the subscribers.

In case of having too little data produced from the processing at the speed layer, compressing the data might result in larger data size. This might be a result of added header information by the compression algorithm. In such a case compression would be counterproductive and would not serve the goal of reducing the transmitted data over the network to the other components of the prototype solution.

The following two Algorithms represent defining a new topology and the pipeline for processing the defined topology. Algorithm 1, presents a topology consisting with one spout and three bolts. Each of these tasks takes in data, does the necessary processing and outputs tuples to the next task. The next task is specified for each of the tasks.

Algorithm 1 Topology for processing data stream

1: **procedure** TOPOLOGY
2: *Spout(data,NextTask)* ← data stream entry point
3: *Bolt1(data,NextTask)* ← Splitting streams to time windows
4: *Bolt2(data,NextTask)* ← Aggregate values + increment counters per data point
5: *Bolt3(data,NextTask)* ← Average data value for each data point

Algorithm 2, illustrates the pipeline of the processing at the data fetcher, where a data stream enters the spout and is transformed into a tuple and sent to Bolt1 and so on. Compression is set to happen after all the processing completes and each processed portion of data is assigned to the specified subscription topic.

Algorithm 2 Pipeline to processing data stream

```
1: procedure ProcessData
2:     NewTopology ← Define a new Topology
3:     Tuples ← NewTopology.Spout(DataStream,Bolt1)
4:     Tuples2 ← NewTopology.Bolt1(Tuples,Bolt2)
5:     Tuples3 ← NewTopology.Bolt2(Tuples2,Bolt3)
6:     Tuples4 ← NewTopology.Bolt3(Tuples3,None)
7:     CompressedData ← CompressData(Tuples4) - compress the processed data
8:     DistributeToSubscriptionTopics(CompressedData)
```

4 Use cases

This section demonstrates the applicability of the prototype solution to three use cases. A use case about Key performance indicators calculations for LTE data counters and this has been covered in previous work. Environmental monitoring use case using mobile air quality sensors called Sensordrones. The third use case discusses the applicability of the prototype solution and the Storm like architecture in the proposed solution on a deep learning example of detecting car license plates and processing this at the far edge network.

4.1 Key Performance Indicator calculations

Key performance indicators (KPIs), are calculations derived from simpler performance counter values that bear some significance regarding the performance of the system. The results of these calculations give an indication of the network performance. KPIs are a form of network metrics. Network metrics are defined to be any form of calculations that represent a certain aspect of the network status and those metrics are known to have a scalar value [12].

Cullen and Frey method, which is a method for finding the type of statistical distribution followed by the data, found that KPI calculations followed a form of continuous probability distribution called Beta distribution [32]. The beta distribution data values are found within the interval [0,1].

KPI calculations for this use case were used in an earlier work [12]. The goal was to study the performance management of the LTE network. In [12] KPI calculations are computed from LTE counters published per cell in a Base Station. Base stations are

distributed across the test network. The test network is an LTE network operating in Espoo and Helsinki and managed by Nokia. The test network consists of 20 LTE Base Stations and 36 LTE cells.

The data fetcher is assumed to have a direct connection to receive the raw LTE counter values of cells in Base Stations. Each cell LTE data counter is an individual data flow, and a collection of flows fetched into the system make up the data stream. Once these counter values reach the spout component of the topology in the speed layer, semantic compression is done to produce the KPI calculations of each counter [35]. Semantic compression is a technique used to convert a fragment of text using terms that are less detailed while preserving the meaning at the same time [31].

KPI calculations of LTE counter values are an aggregation of the values of one or more raw counter data values. These KPI calculations give an indication of how this base station cell is performing and when sending only the KPI calculation instead of all the LTE counter values we are sending less data [35]. A script is defined for calculating each KPI and consumes specified counter values [35]. The more counter values we can discard due to a KPI calculation the more compression is achieved and the more reduction there is in the processed and transmitted data.

After the reduction, the spout generates tuples from the data stream and then moves them to Bolt1 to be split to time windows. Bolt2 counts the KPI calculations for each time window and aggregates the values. The mean for each time window is calculated in Bolt3. The processed data for each subscription topic is then compressed using GZIP algorithm and sent by the fetcher to the data switch where it is routed to the proper subscription request through defined topics. The data hub decompresses the processed data before sending it to the subscribers. The idea behind keeping each processing task in a separate topology component is to help in distributing those tasks and to make the design more scalable we can add more tasks in between and connect them to the designed processing pipeline.

4.2 Environmental Sensing

Applications are increasingly relying on data provided by sensors. These sensors can be already available in mobile devices or IoT wearable devices and they generate continuous data over time. Traditionally this data needs to be collected and then transferred to the central cloud for processing and further analytic operations. These operations can be general computations, monitoring or generating data logs [2].

Environmental awareness is very important. Pollution levels are rising and it is affecting communities and people's health and their health on many levels [25]. The quality of the air we breathe inside the workplace, houses and outdoors can have an impact on how energetic and productive we are and we are witnessing an increase in health problems due to polluted air.

Air quality sensors have features to detect if there is a gas leak or if the levels of oxygen are dropping in a way that would put people's lives in danger or alarm if there are indicators of a fire starting. Processing the data streams coming from these sensor devices as soon as it arrives can help in firing an alarm in the right moment. Therefore, reliability and survivability are two important qualities in such sensors. The Megasense project studies air quality monitoring [46].

Evolution in IoT, sensors and smartphone technology resulted in the emergence of low-cost sensors. Those sensors are cheap, light-weight and easy to carry around. This enables many new applications including such that enabled people to be more aware of the air they breathe and help them monitor their bodies and health.

For this thesis, we chose to use an environmental sensing product called Sensordrone that was manufactured by SensorCon [63]. The Sensordrone is able to connect through Bluetooth LE to the far edge or a gateway device. An open source Java library is used to handle the connection and data collection from the Sensordrone devices.

Sensordrone provides three modes of operation: sending most recent data, data streaming (sending data in real-time) and data logging where it stores the readings in memory until they are needed and downloaded as csv files.

Figure 13 illustrates the main components of the Sensordrone. Main Sensordrone unit contains integrated sensors for gas (precision electrochemical, oxidizing gases, reducing gases), temperature, humidity, pressure, non-contact thermometer, proximity capacitance, color intensity (Red, Blue, Green, illumination) and expansion connector that enables adding more sensors to the device [64]. Several factors affecting measurements will also be considered since the device tends to heat up while it is charging which can affect the temperature measurements. Furthermore, the sensors have the tendency to drift sometimes. For that, it will also be good to detect such anomalies in the measurements and give an indicator when the sensor is drifting in order to check the reason behind the drifts in measurements.

A detailed table of the sensors and their IDs and main functions can be found in

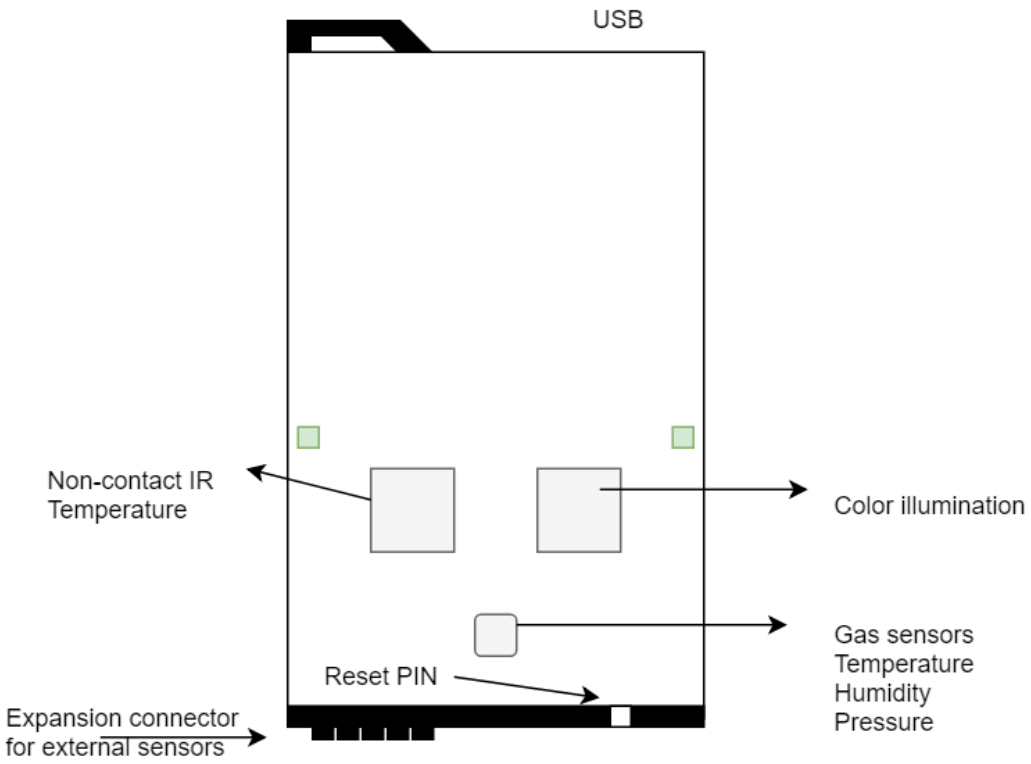

Figure 13: **Sensordrone is a portable sensing device** *An illustration of the main components of Sensordrone.*

Appendix 1, Table 6. The code for data collection is provided in Appendix 2. The data collection code is written in Java and it is based on an open source project [1]. Some modifications were done to the code. A sample of collected data is shown in Appendix 3 with more details.

Earlier work with Sensordrone

Traditional stationary air quality and pollution monitoring sensors come with high accuracy. However, they tend to be bulky and expensive and it is difficult to move them to different locations of interest as they have to be installed in each of these locations.

Portable sensors are cheaper and remove this inconvenience of having the sensor in a fixed place. Portable sensors enabled the crowd to further assist in collecting data from different locations in a city or to monitor indoor and outdoor air quality. This means that multiple sensors can be used in many locations to perform air quality

[1] https://github.com/jmineraud/sensordrone-air-quality-logger

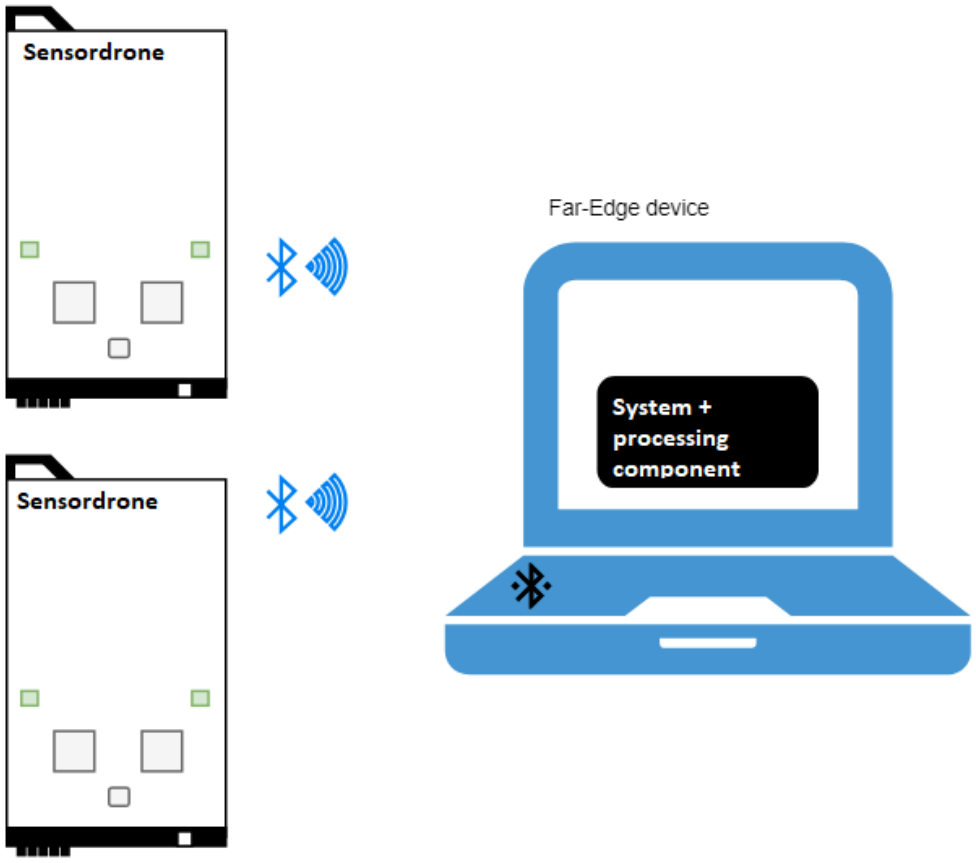

Figure 14: **Sensordrones sending streams and the solution prototype connecting to the data sources.** *The far edge device is the gateway between the prototype solution and the IoT devices, Sensordrones in this case. The connection between the solution prototype's fetcher component and the Sensordrones is a Bluetooth LE connection.*

monitoring. Yet, those portable sensors can be less accurate and tend to experience drifting in readings quite often.

Participatory sensing [66] or community-based sensing [30] is an environmental monitoring method that uses smartphones or other kinds of portable sensing devices. This method depends on the crowd actively participating in the data collection process.

The studied previous work with Sensordrones involved participatory sensing in the process of collecting the data. The following work has been done with Sensordrones and they also used Android-based smart devices and the process involved the storage and transmitting the data to some central server for further processing.

Work in DroneSense [66] involved two kinds of sensing devices, Sensordrone and Waspmote [41]. Sensordrone sends data via Bluetooth LE and Waspmote sends the data via Wifi. The smart device collects the data from both devices and then sends the data to cluster heads and a central server for further processing. Cluster heads attempt to reduce the computation load on the central server. Kalman Filter was applied to the collected data to reduce noise in the measurements. Furthermore, the central server gave the users some ranking that infers which users are producing more reliable measurements than others. Aggregation was used to exploit redundant data. The work also handled a failure scenario, where the data is redirected to another cluster if one of the cluster heads fail.

SecondNose [39] system had a crowd of 80 citizens collecting environmental data. The work involved storage of historical data and a back-end system for collecting and analyzing the results. Furthermore, they further developed a web application to visualize the aggregated measurements to give a weekly personalized view for each authorized user in the crowd participating. Connection failure in this work was handled by re-attempts to connect until there is a successful connection.

Jafari et. al. [30] developed an air quality monitor. The data was stored both on the smart device and another copy was sent to the central server for processing via TCP sockets. Collection and aggregation of measurements happened at the server that then visualized the results on google maps in both real-time and historical views.

SmartVent [43] system had both static and portable sensors for collecting measurements. Static sensing device was an Arduino UNO R3 that had temperature and humidity sensors and it was deployed in rooms that were highly occupied. The smart devices collected data from Sensordrones, while the Arduino sent the data directly to the central server. The collection and storage happened at the central server where the data was analyzed and then visualized.

Applying environmental sensing to the prototype solution

The work in this thesis differs from the above in that the collection will happen directly on the laptop, i.e., the far edge device that connects to the Sensordrones. The processing of the data will happen at the far edge device and there will be no transmission of the data to the central server to do the processing.

The data collection script in Appendix 2 was modified to collect measurements from multiple Sensordrone devices every ten seconds. The data was stored as comma

separated values (csv). A sample of the data is presented in Appendix 3.

The data from the sensors was collected in a room over the course of three days to measure the quality of the air in that room. During the process, the system occasionally experienced some failures to connect to some of the sensors. After resetting the connection the data collection proceeded normally.

We can control the speed at which the data is collected. In Appendix 2, we configured the data collection script to collect the data every ten seconds.

In processing, the spout transforms each line of sensor readings into a tuple. Bolt1 then groups the tuples into non-overlapping time windows. Bolt2 aggregates the readings of each sensor and it also counts the number of readings. The average reading for each sensor in a time window is calculated in Bolt3 and the data is then compressed, to reduce the size of the data, and sent to the data switch.

4.3 Deep Learning Application

Deep learning is a branch of machine learning [74] where the word deep indicates the complexity of the model of computation. This complexity involves several layers each having a simple task. Each layer in the deep learning model is linked to other layers in the model in a logical manner. The input to each layer is processed and passed to the next processing unit after the processing is completed in the previous layer.

Modern surveillance and traffic control management are increasingly using techniques, such as deep learning, for detecting car license plates. Detecting license plates involves recognizing the plate itself the digits and characters present on a plate given an image, sequence of images or a video. This can help in detecting stolen cars, managing parking spaces, and allowing authorized vehicles to enter in restricted parking spaces. In such cases, there is a high demand for real-time processing and quick responses.

Deep Convolutional Neural Networks are one application of deep learning and they consist of feed-forward Neural Networks and back-propagation Neural Networks [37]. In back-propagation Neural Networks the weights are constantly learnt for these networks. Convolutional Neural Networks are a form of back-propagation Neural Networks and they are applied in tasks such as digit recognition and image classification.

Figure 15: **Finnish car license plates.**
The top part shows different license plates for vehicles. The middle part shows different plates for veteran vehicles which are vehicles that are older than 30 years and have a black background. The last part shows motor bike license plates. Images collected from Trafi 'Finnish Transport Safety Agency' website[68].

This use case is a study of the applicability of deep learning to process an image or video stream using the proposed processing model and provide results in real-time. For the model in this use case, we chose three categories of license plates issued in Finland [69]. Those categories are for vehicle license plates, veteran vehicles and motorbikes. Figure 15 shows some categories of vehicle license plates. Each license plate type in each category vary in dimensions, has two to three Latin characters and two to three numbers separated by a dash in some plates. All the plates use the same font and all letters are in upper case.

In newer plates, there is a blue field on the top left with the European flag twelve stars and the nationality ID FIN below the stars. Furthermore, license plates can be placed in different locations depending on the vehicle size and type. The plates themselves have different dimensions as in Figure 15 and those dimensions depend on the rules for issuing them.

However, a traffic monitoring system can make mistakes when issuing traffic speeding tickets, for example, a ticket issued for a vehicle in Sweden or Estonia can be registered for a vehicle in Finland or vice versa. This is due to similarities between license plates and the recognition algorithm's failure to detect which country the plate in question belongs to. Therefore it is also important to recognize which country a vehicle plate belongs to by understanding the differences in the features of the plates between countries.

To ensure the accuracy of the deep learning model it is important to provide clean data with proper annotations [45]. This clean data consists of vehicle images with the license plate images in clear locations and with minimal noise introduced to them and this is for the task of detecting the plates in an image. Another set of clean data needed is for the task of recognizing the digits in the plates with minimal noise introduced to those images. Using minimal or no noise in the training data helps the model achieve higher accuracy when it is applied later on.

Due to the limitation in storage and computation resources at the edge, the training of the model will happen on the central server [45]. Due to the need of heavy computations and the resources for these computations are available at the central server. After the training part, the model is applied at the edge of the network. When running the model it will start by detecting the plate in an image and then it will move to detecting and recognizing the digits in the plate. There are different methods for detecting the vehicle license plate location in an image such as edge features, where the detection algorithms look for rectangular shapes in the image, image features, i.e., having characters in the license plate helps the algorithm detecting it [42]. Usually using a combination of location detection techniques help in getting better results [42].

Factors that are important when detecting the license plate from a given image include the location of the plate as different vehicle models have different plate sizes, plates can further have different shapes depending on the date of issuing them or again the type of the vehicle. Colors and fonts of the license plate affect the classification task and when all the vehicles use the same font the task is made easier for the classifier. A plate can be missing, for example, if the car is also missing and in this case, the system will not proceed with the pipeline. Tilted plates or plates full of dirt makes the task harder. Changes in the environment are also important factors that affect the accuracy of the detection algorithm as well as the degree of illumination.

The surveillance camera provides an input image or video to the system once a car appears in front of the gate or when malfunctions are detected. Those are the input stream in this case. Lossy compression or compressive sensing techniques are to be applied to help reduce the data load but still preserve the quality. Then, the image is converted to grey-scale which reduces it from 24-bit RGB value to 8-bit grey-scale value [60]. This input image then goes through the processing component to detect the plate and then the characters on that plate. Location algorithm using edge-based features are used to detect the plate on the vehicle.

For detecting the characters on the plate, the image is passed to the segmentation deep learning algorithm. This divides the characters on the plate and sends them to the recognition deep learning task. In the recognition task, each of the characters is classified to the character category they belong to. However, some errors in classifying characters can still happen in cases where characters might look similar such as B and 8, letter O and number zero, I and 1, A and 4, C and G, D and O, K and X [40].

The pipeline in Figure 16 shows the transition from one stage to another through the processing layer of the system to detect the characters on the license plate. After the last classification task has completed, the final real-time aggregated result of the characters on the plate is then sent by the data fetcher to the serving layer in the data switch component of the prototype solution in order to provide the near real-time view. The data switch then maps the result through the rest of the solution prototype and the apps acting as subscribers check whether the car is authorized to park. Applying the model close to the data sources brings the data closer to the processing pipeline and in such cases, this can help in reducing the amount of data transfers.

In case of having new types of plates issued then the model will have to be trained again and then applied at the edge and also in the case where there is a need to include a wider range of plate types.

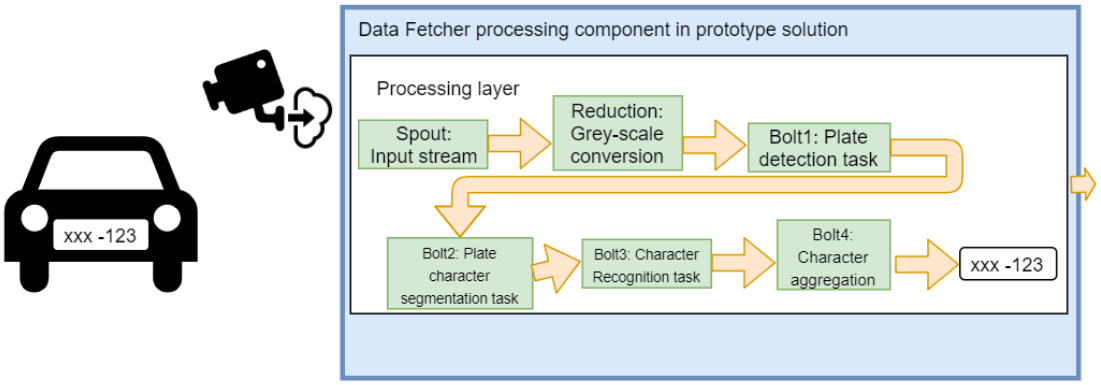

Figure 16: **Applying deep learning to detect car license plates numbers.** *The camera streams video or images of cars once they reach the parking space. Deep learning models involve plate location detection, plate character detection and character recognition.*

5 Discussion

We are heading towards a more dynamic lightweight solution. This thesis studied data streams and data stream processing solutions including Lambda and Kappa. The proposed solution illustrated applying Kappa model within the prototype solution developed by Nokia.

We studies differences between the Lambda and the Kappa data stream processing models. Where the Lambda model has a layer for processing historical data and continuously update this data with the new incoming data. While the Kappa model is doing the processing and the reprocessing only on the new incoming data. The reasoning behind the Kappa model is reducing the need for batch processing and limit the task to processing the data in near real-time only with limiting the need for storage.

Apache Storm influenced the design of the proposed processing component and the Kappa processing solution was applied to the prototype solution in two stages, the data fetcher which contained the speed layer and the data switch that contained the linking layer.

The prototype system is working toward minimizing the number of data transfers through the network. By operating at the edge or at the far-edge, we brought the processing capability closer to the data sources. In this work, the issue of device discoverability was not studied but it will be in the scope of future work as this

problem has different aspects to it.

However, while collecting data from the Sensordrone devices we experienced some connection problems with the devices and that took several attempts to be able to connect and proceed with the data collection again. Future work will also include applying the prototype solution to connect to stationary sensing devices like the ones used in the Megasense project.

Through this work, we studied the possibility to distribute the processing computations that are being performed at the far-edge device. Due to limited computation and storage capabilities, distributing the computations using a Storm like architecture and executing processing tasks in parallel and having them distributed on different devices can help distribute the computation load and would allow the processing to complete at a faster speed. The proposed Storm like architecture will be developed further in the continuation of this work.

Data reduction and compression techniques were discussed in this work. Data reduction techniques, i.e., semantic compression can be useful when meaningful summaries can be generated from the collected data as in the case of generating KPI calculations from base station cells' LTE data counters. While the data reduction is also applied when setting how often to collect data as in the case of environmental sensing. In the deep learning application, the reduction is made to the quality of the image.

Data compression was set to happen once the processing of the data is completed at the data fetcher component of the prototype solution. Two compression techniques were compared that are the Lossy and the Lossless compression techniques [3]. Lossy compression is applied to the data stream causing some data is discarded in order to reduce the actual size of the data which reduces the amount of data that needs to be processed [17].

In Kapoor's work [32], he implemented the compression in the data fetcher component. Kapoor defined the accuracy limits for KPI data compression using quality of monitoring concept for each KPI calculation. Quality of monitoring is done to maintain accuracy while compressing the data. Kapoor proposed a modified version of piece-wise constant approximation algorithm when compressing the KPI data. However, using the quality of monitoring concept introduced compression gain. To calculate this gain he assumed i to be the number of KPIs calculated from the counter values. LTE counter data needed to calculate KPI, we denote it here by c. The number of data values that needed to be transferred if the calculation of KPI

did not happen at the edge of the network would be i*c. In Kapoor's solution, i values needed to be transferred which lead to a direct compression gain.

Another technique that is applicable to reduce the size of the incoming data, video or image stream is compressed sensing where the number of measured signals and images are reconstructed from reduced number of measurements while preserving the structure. In cases of having critical data, lossless is more advisable while when discarding parts of the data and still being able to construct the signal then Lossy will not cause any critical losses in the data. However, when having very few data points, compression can lead to larger compressed file size due to the header added by the compression algorithm.

In this thesis, the proposed processing component aimed to enhance the existing prototype solution by moving the computations to the far-edge device that acted as the gateway for the prototype solution and by that the computation power was moved to the far-edge network, reducing the load on the edge and the central server.

Table 4 compares the different requirements needed for applying the processing component to each of the use cases. We can see the different data reduction methods that are applicable in each case as illustrated. The resulting data after the processing is also different in each use case and the component should be able to tolerate varying data types. As for the compression techniques, the recommended technique is also highlighted in the table.

Table 4: Use case requirements comparison.

Use case	LTE Data counter values	Environmental sensing	License plate detection
Data reduction before processing	Semantic compression (calculating KPI from raw counter values).	Control how often to collect the data.	Reduce image quality from 24-bit RGB to 8-bit grey-scale.
Processing outcome	Indication of network performance.	Average reading for each of the sensors for every time window.	Letters and digits on a license plate.
Compression Technique	Lossless compression	Lossless compression	Lossy compression

6 Conclusion

In this thesis, we studied the applicability of processing data streams at the far edge. We designed a processing component to enhance the processing done in the Nokia prototype solution. The prototype solution followed a topic-based publish/subscribe architecture. The solution prototype consists of data hubs, data switches and data fetchers. The data fetchers in our case are connected to the data sources. The processing component developed in this thesis is implemented within the data fetcher component.

A Storm-like architecture was designed as part of the processing component. Two data stream processing solutions, Lambda and Kappa, were studied. Kappa showed to be an improvement over Lambda as it removed the need to duplicate the incoming stream for both the batch and the real-time processing. However, each stream processing solution can fit a different use case scenario and they also influenced the designed processing solution at the data fetcher.

Moving the processing away from the central server to the far-edge reduces the number of data transfers and this saves bandwidth. Combining this with compression when needed helps in saving more bandwidth.

Data compression was proposed in earlier work done on the prototype solution. Lossless techniques such as gzip algorithms were more suitable for data such as KPI calculations and sensor readings. In the pipeline, the example application of KPI calculations applied semantic compression on LTE counter values to compute KPIs which were an indicator of those counter values. Before sending the data back to the data hubs gzip compression is used to reduce the size of the transmitted data.

The data hub then decompresses the data and sends it to the subscribing applications. By using two compression techniques, we reduced the size of the data that we needed to process and in the second one we reduced the data that we needed to send to the applications which also saves bandwidth.

Environmental sensing is another use case where we studied the applicability of the proposed processing component. The input stream came from Sensordrone air quality sensing devices. Future work will include better quality and more accurate sensors. Sensordrones were used for the demonstration of the use case in this thesis and study the applicability in the domain of environmental sensing.

Deep learning can also be applicable on the solution prototype. The training of the model can take place on a central server and applying the model can be done at the

Table 5: Requirement analysis for the proposed solution.

Framework	Proposed processing solution
Need to duplicate incoming data	No need since there is only one processing layer and no storage is needed for the data.
Batch/real time processing	Near real-time processing.
Model design	Speed layer (i.e. the processing layer) implemented at data fetcher. Serving layer (routing the subscriptions) at the data switch.
Resource usage	Data reduction aimed to reduce the amount of data that needed processing and the data compression aimed to reduce the size of the data transmitted over the network.

edge or the far-edge. Future work might include applying the proposed pipeline to apply deep learning computations at the edge of the network.

References

1 Design elements - telecommunication networks. Accessed on 10.12.2018. URL: https://conceptdraw.com/a915c3/preview--DesignElements.

2 Edge computing vs. cloud computing: What's the difference? URL: https://www.datamation.com/cloud-computing/edge-computing-vs.-cloud-computing-whats-the-difference.html.

3 What is lossless and lossy compression? Accessed on 10.12.2018. URL: https://whatis.techtarget.com/definition/lossless-and-lossy-compression.

4 3GPP. The mobile broadband standard. Accessed on 10.12.2018. URL: http://www.3gpp.org/technologies/keywords-acronyms/97-lte-advanced.

5 K. A. Alam, R. Ahmad, and K. Ko. Enabling far-edge analytics: performance profiling of frequent pattern mining algorithms. *IEEE Access*, 5:8236–8249, 2017.

6 M. Amadeo, C. Campolo, A. Molinaro, and G. Ruggeri. IoT data processing at the edge with named data networking. In *European Wireless 2018; 24th European Wireless Conference*, pages 1–6. VDE, 2018.

7 Amazon. What is cloud computing? - Amazon Web Services. Accessed on 10.12.2018. URL: https://aws.amazon.com/what-is-cloud-computing/.

8 Apache. Apache Flume. Accessed on 10.12.2018. URL: https://flume.apache.org/.

9 Apache. Apache Kafka. Accessed on 10.12.2018. URL: http://kafka.apache.org/.

10 Apache. Apache Spark - Unified Analytics Engine for Big Data. Accessed on 10.12.2018. URL: http://spark.apache.org/.

11 Apache. Apache Storm. Accessed on 10.12.2018. URL: http://storm.apache.org/.

12 K. Apajalahti, J. Niiranen, S. Kapoor, and V. Räisänen. Sharing performance measurement events across domains. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 463–469. IEEE, 2017.

13 R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *25th IEEE International Conference on Distributed Computing Systems, 2005.*, pages 437–446. IEEE, 2005.

14 B. Blanco, J. O. Fajardo, I. Giannoulakis, E. Kafetzakis, S. Peng, J. Pérez-Romero, I. Trajkovska, P. S. Khodashenas, L. Goratti, M. Paolino, et al. Technology pillars in the architecture of future 5G mobile networks: NFV, MEC and SDN. *Computer Standards & Interfaces*, 54:216–228, 2017.

15 F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

16 I. T. J. W. R. Center. IBM. Gryphon: Publish/subscribe over public networks. Technical report. Technical report, IBM, 2001.

17 H. Chen, J. Li, and P. Mohapatra. RACE: Time series compression with rate adaptivity and error bound for sensor networks. In *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 124–133. IEEE, 2004.

18 Cisco. Edge computing vs. fog computing: Definitions and enterprise uses, Jan 2018. Accessed on 10.12.2018. URL: `https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html`.

19 M. D. de Assuncao, A. da Silva Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

20 V. Dobrodeev. Efficient network management plane data dissemination with publish/subscribe paradigm [unpublished]. Master's thesis, Helsingin yliopisto, 2018.

21 B. Erb and F. Kargl. A conceptual model for event-sourced graph computing. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 352–355. ACM, 2015.

22 P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

23 R. Evans. Apache Storm, a hands on tutorial. In *2015 IEEE International Conference on Cloud Engineering (IC2E)*, pages 2–2. IEEE, 2015.

24 F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM Sigmod Record*, volume 30, pages 115–126. ACM, 2001.

25 D. L. Gadenne, J. Kennedy, and C. McKeiver. An empirical study of environmental awareness and practices in smes. *Journal of Business Ethics*, 84(1):45–63, 2009.

26 J. Gama and P. P. Rodrigues. Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer, 2007.

27 Z. Hasani, M. Kon-Popovska, and G. Velinov. Lambda architecture for real time big data analytic. *ICT Innovations*, pages 133–143, 2014.

28 S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

29 K. Ismail. Edge computing vs. fog computing: What's the difference?, Aug 2018. URL: `https://www.cmswire.com/information-management/edge-computing-vs-fog-computing-whats-the-difference/`.

30 H. Jafari, X. Li, L. Qian, and Y. Chen. Community based sensing: A test bed for environment air quality monitoring using smartphone paired sensors. In *Sarnoff Symposium, 2015 36th IEEE*, pages 12–17. IEEE, 2015.

31 P. Jedrzejowicz, N. Nguyen, and K. Hoang. Computational collective intelligence. technologies and applications. In *4th International Conference, ICCCI*, page 163, 2012.

32 S. Kapoor. Mobile edge compression of management data in cellular networks using quality of monitoring classes. Master's thesis, Helsingin yliopisto, 2017.

33 E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Sigmod Record*, 30(2):151–162, 2001.

34 M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data*, pages 2785–2792. IEEE, 2015.

35 V. Kojola, S. Kapoor, and K. Hätönen. Distributed computing of management data in a telecommunications network. In *International Conference on Mobile Networks and Management*, pages 146–159. Springer, 2016.

36 J. Kreps. Questioning the lambda architecture.(2014). *https://www. oreilly. com/ideas/questioning-the-lambda-architecture>. Citado*, 2:25, 2014. Accessed on 10.12.2018.

37 A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

38 T. L. World Wide Streams - a new World Wide Web for in-the-moment information. Accessed on 10.12.2018. URL: `https://www.bell-labs.com/our-research/publications/298833/`.

39 C. Leonardi, A. Cappellotto, M. Caraviello, B. Lepri, and F. Antonelli. Secondnose: an air quality mobile crowdsensing system. In *Proceedings of the 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*, pages 1051–1054. ACM, 2014.

40 H. Li and C. Shen. Reading car license plates using deep convolutional neural networks and LSTMs. *arXiv preprint arXiv:1601.05610*, 2016.

41 Libelium. Waspmote. Accessed on 10.12.2018. URL: `http://www.libelium.com/products/waspmote/`.

42 Y. Liu, H. Huang, J. Cao, and T. Huang. Convolutional neural networks-based intelligent recognition of chinese license plates. *Soft Computing*, 22(7):2403–2419, 2018.

43 D. Lohani and D. Acharya. Smartvent: A context aware IoT system to measure indoor air quality and ventilation rate. In *2016 17th IEEE International Conference on Mobile Data Management*, volume 2, pages 64–69. IEEE, 2016.

44 T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015.

45 S. Z. Masood, G. Shu, A. Dehghan, and E. G. Ortiz. License plate detection and recognition using deeply learned convolutional neural networks. *arXiv preprint arXiv:1703.07330*, 2017.

46 Megasense. Sensing and analytics of air quality, Oct 2018. Accessed on 10.12.2018. URL: `https://www.helsinki.fi/en/researchgroups/sensing-and-analytics-of-air-quality`.

47 G. Miao, J. Zander, K. W. Sung, and S. B. Slimane. *Fundamentals of Mobile Data Networks*. Cambridge University Press, 2016.

48 J. Mineraud. jmineraud/sensordrone-air-quality-logger, Nov 2018. URL: `https://github.com/jmineraud/sensordrone-air-quality-logger`.

49 A. R. Mishra. *Fundamentals of Network Planning and Optimisation 2G/3G/4G: Evolution to 5G*. John Wiley & Sons, 2018.

50 R. N. Mitra and D. P. Agrawal. 5G mobile technology: A survey. *ICT Express*, 1(3):132–137, 2015.

51 J. Moysen and L. Giupponi. From 4G to 5G: Self-organized network management meets machine learning. *Computer Communications*, 2018.

52 G. Mühl. *Large-scale content-based publish-subscribe systems*. PhD thesis, Technische Universität, 2002.

53 S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

54 Nokia. Security challenges and opportunities for 5g mobile networks, 2017. Accessed on 10.12.2018. URL: `https://onestore.nokia.com/asset/201049`.

55 C. Pahl and B. Lee. Containers and clusters for edge cloud architectures–a technology review. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 379–386. IEEE, 2015.

56 M. Pal Singh, M. A. Hoque, and S. Tarkoma. A survey of systems for massive stream analytics. *arXiv preprint arXiv:1605.09021*, 2016.

57 S. Parkvall, E. Dahlman, A. Furuskar, and M. Frenne. Nr: The new 5G radio access technology. *IEEE Communications Standards Magazine*, 1(4):24–30, 2017.

58 P. Persson and O. Angelsmark. Kappa: serverless IoT deployment. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 16–21. ACM, 2017.

59 P. Pirinen. A brief overview of 5G research activities. In *2014 1st International Conference on 5G for Ubiquitous Connectivity (5GU)*, pages 17–22. IEEE, 2014.

60 C. Saravanan. Color image to grayscale image conversion. In *2010 Second International Conference on Computer Engineering and Applications*, pages 196–199. IEEE, 2010.

61 D. Satria, D. Park, and M. Jo. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems*, 70:138–147, 2017.

62 M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 2009.

63 SensorCon. SensorCon. Accessed on 10.12.2018. URL: `https://sensorcon.com/`.

64 SensorCon. Sensordrone: The 6th Sense of Your Smartphone. Accessed on 10.12.2018. URL: `https://www.kickstarter.com/projects/453951341/sensordrone-the-6th-sense-of-your-smartphoneand-be`.

65 G. Sites. Lte encyclopedia. Accessed on 10.12.2018. URL: `https://sites.google.com/site/lteencyclopedia/home`.

66 W. Sun, Q. Li, and C.-K. Tham. Wireless deployed and participatory sensing system for environmental monitoring. In *2014 Eleventh Annual IEEE International Conference on Sensing, Communication, and Networking*, pages 158–160. IEEE, 2014.

67 S. Tarkoma. *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

68 TraFi. Finnish transport safety agency. Accessed on 10.12.2018. URL: `https://www.trafi.fi/`.

69 TraFi. Registration plates. Accessed on 10.12.2018. URL: `https://www.trafi.fi/en/road/registration_plates`.

70 M. H. ur Rehman, C. S. Liew, T. Y. Wah, and M. K. Khan. Towards next-generation heterogeneous mobile data stream mining applications: Opportunities, challenges, and future research directions. *Journal of Network and Computer Applications*, 79:1–24, 2017.

71 J. S. van der Veen, B. van der Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer. Dynamically scaling Apache Storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 154–161. IEEE, 2015.

72 M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182. IEEE, 2016.

73 M. Villari, A. Celesti, M. Fazio, and A. Puliafito. Alljoyn lambda: An architecture for the management of smart environments in IoT. In *2014 International Conference on Smart Computing Workshops (SMARTCOMP Workshops)*, pages 9–14. IEEE, 2014.

74 J. Wan, D. Wang, S. C. H. Hoi, P. Wu, J. Zhu, Y. Zhang, and J. Li. Deep learning for content-based image retrieval: A comprehensive study. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 157–166. ACM, 2014.

75 Q. Wu, G. Y. Li, W. Chen, D. W. K. Ng, and R. Schober. An overview of sustainable green 5G networks. *IEEE Wireless Communications*, 24(4):72–80, 2017.

76 Y. Wu, P. A. Chou, S.-Y. Kung, et al. Information exchange in wireless networks with network coding and physical-layer broadcast. Technical report, MSR-TR-2004, 2005.

Appendix 1. Sensordrone sensors and their usage

Here is a table that details the sensors that exist in the SensorDrone device and the usage of each of them.

Table 6: Sensordrone sensors and their reference IDs

Sensor type	Sensor ID	Details
Temperature	1	Temperature measured in Celsius
Color	2	Red, Blue, Green Illumination
Reducing Gas	3	Hydrocarbons such as Methane, Propane and alcohols
Pressure	4	Barometer, Blood Pressure
Precision Gas	5	Calibrated for Carbon Monoxide
Oxidizing Gas	6	Chlorine, Ozone, Nitrogen Dioxide
IR temperature	7	Simple resistance temperature
Humidity	8	Humidity percentage
Capacitance	9	Fluid level, intrusion detection, stud finder
Altitude	10	Altimeter
Battery Voltage	11	Measures battery voltage

Appendix 2. Code used for collecting data streams from sensors

Code for collecting the data from the sensordrones is taken from [48]. The following code was modified to generate the streams of measurements without the longitude and the latitude.

The modified task code can be called as in Figure 17.

```
$ ./gradlew run -q -PappArgs="['-d', '10000', '-m', '00:17:e9:50:e1:75']" 2>> 22
noverr175 1>> 22novoutput175
```

Figure 17: **Gradlew command to run the data collection task.** *The measurements are set to be collected every 10 seconds.*

```
//sensordrone-air-quality-logger/src/main/java/fi/helsinki/cs/sensordrone/air/quality/logger/SensorDroneDataCollectionTask.java
package fi.helsinki.cs.sensordrone.air.quality.logger;
import com.sensorcon.sensordrone.DroneEventHandler;
import com.sensorcon.sensordrone.DroneEventObject;
import com.sensorcon.sensordrone.java.Drone;
import java.util.Locale;
public class SensorDroneDataCollectionTask implements Runnable {
    private static Drone drone;
    private static String macAddress;
    private final long timeout;
    private boolean batteryMeasured = false;
    private boolean altitudeMeasured = false;
    private boolean capacitanceMeasured = false;
    private boolean humidityMeasured = false;
    private boolean irTemperatureMeasured = false;
    private boolean oxidizingGasMeasured = false;
    private boolean precisionGasMeasured = false;
    private boolean pressureMeasured = false;
    private boolean reducingGasMeasured = false;
    private boolean rgbMeasured = false;
    private boolean temperatureMeasured = false;
    private boolean debug = false;
    private boolean[] measuring = new boolean[12];
    private static final int TEMPERATURE_SENSOR_ID = 1;
    private static final int COLOR_SENSOR_ID = 2;
    private static final int REDUCING_GAS_SENSOR_ID = 3;
    private static final int PRESSURE_SENSOR_ID = 4;
    private static final int PRECISION_GAS_SENSOR_ID = 5;
    private static final int OXIDIZING_GAS_SENSOR_ID = 6;
    private static final int IR_TEMPERATURE_SENSOR_ID = 7;
    private static final int HUMIDITY_SENSOR_ID = 8;
    private static final int CAPACITANCE_SENSOR_ID = 9;
    private static final int ALTITUDE_SENSOR_ID = 10;
    private static final int BATTERY_VOLTAGE_SENSOR_ID = 11;
    private static final int[] SENSOR_MASK = new int[] {
     COLOR_SENSOR_ID, PRESSURE_SENSOR_ID, OXIDIZING_GAS_SENSOR_ID,
     TEMPERATURE_SENSOR_ID, HUMIDITY_SENSOR_ID
    };
    private static boolean validSensor(int sensorId) {
        if (SENSOR_MASK == null) return true;
        for (int id : SENSOR_MASK) {
            if (id == sensorId) return true;
        }
        return false;
    }
    SensorDroneDataCollectionTask(String macAddress, long timeout, boolean debug) {
```

```
if (SensorDroneDataCollectionTask.macAddress == null) {
    SensorDroneDataCollectionTask.drone = new Drone();
    SensorDroneDataCollectionTask.macAddress = macAddress;

    for (int i = 0; i < measuring.length; i++) {
        measuring[i] = validSensor(i + 1);
    }
    DroneEventHandler mDroneEventHandler = new DroneEventHandler() {
        @Override
        public void parseEvent(DroneEventObject event) {
            if (event.matches(DroneEventObject.droneEventType.CONNECTED)) {
                drone.setLEDs(126, 0, 0); // Set LED red when connected

                Enable the desired sensors defined in SENSOR_MASK
                if (validSensor(ALTITUDE_SENSOR_ID)) drone.enableAltitude();
                if (validSensor(CAPACITANCE_SENSOR_ID)) drone.enableCapacitance();
                if (validSensor(HUMIDITY_SENSOR_ID)) drone.enableHumidity();
                if (validSensor(IR_TEMPERATURE_SENSOR_ID)) drone.enableIRTemperature();
                if (validSensor(OXIDIZING_GAS_SENSOR_ID)) drone.enableOxidizingGas();
                if (validSensor(PRECISION_GAS_SENSOR_ID)) drone.enablePrecisionGas();
                if (validSensor(PRESSURE_SENSOR_ID)) drone.enablePressure();
                if (validSensor(REDUCING_GAS_SENSOR_ID)) drone.enableReducingGas();
                if (validSensor(COLOR_SENSOR_ID)) drone.enableRGBC();
                if (validSensor(TEMPERATURE_SENSOR_ID)) drone.enableTemperature();
                if (validSensor(BATTERY_VOLTAGE_SENSOR_ID)) drone.measureBatteryVoltage();
                // Enabled
            } else if (event.matches(DroneEventObject.droneEventType.ALTITUDE_ENABLED)) {
                drone.measureAltitude();
            } else if (event.matches(DroneEventObject.droneEventType.CAPACITANCE_ENABLED)) {
                drone.measureCapacitance();
            } else if (event.matches(DroneEventObject.droneEventType.HUMIDITY_ENABLED)) {
                drone.measureHumidity();
            } else if (event.matches(DroneEventObject.droneEventType.IR_TEMPERATURE_ENABLED)) {
                drone.measureIRTemperature();
            } else if (event.matches(DroneEventObject.droneEventType.OXIDIZING_GAS_ENABLED)) {
                drone.measureOxidizingGas();
            } else if (event.matches(DroneEventObject.droneEventType.PRECISION_GAS_ENABLED)) {
                drone.measurePrecisionGas();
            } else if (event.matches(DroneEventObject.droneEventType.PRESSURE_ENABLED)) {
                drone.measurePressure();
            } else if (event.matches(DroneEventObject.droneEventType.REDUCING_GAS_ENABLED)) {
                drone.measureReducingGas();
            } else if (event.matches(DroneEventObject.droneEventType.RGBC_ENABLED)) {
                drone.measureRGBC();
            } else if (event.matches(DroneEventObject.droneEventType.TEMPERATURE_ENABLED)) {
                drone.measureTemperature();
                // Measured
            } else if (event.matches(DroneEventObject.droneEventType.BATTERY_VOLTAGE_MEASURED)) {
                logSample(BATTERY_VOLTAGE_SENSOR_ID, drone.batteryVoltage_Volts);
                batteryMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.ALTITUDE_MEASURED)) {
                // drone.altitude_Meters, drone.altitude_Feet
                logSample(ALTITUDE_SENSOR_ID, drone.altitude_Meters);
                altitudeMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.CAPACITANCE_MEASURED)) {
                logSample(CAPACITANCE_SENSOR_ID, drone.capacitance_femtoFarad);
                capacitanceMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.HUMIDITY_MEASURED)) {
                logSample(HUMIDITY_SENSOR_ID, drone.humidity_Percent);
                humidityMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.IR_TEMPERATURE_MEASURED)) {
                // drone.irTemperature_Celsius, drone.irTemperature_Fahrenheit, drone.irTemperature_Kelvin
                logSample(IR_TEMPERATURE_SENSOR_ID, drone.irTemperature_Celsius);
                irTemperatureMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.OXIDIZING_GAS_MEASURED)) {
                logSample(OXIDIZING_GAS_SENSOR_ID, drone.oxidizingGas_Ohm);
                oxidizingGasMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.PRECISION_GAS_MEASURED)) {
                logSample(PRECISION_GAS_SENSOR_ID, drone.precisionGas_ppmCarbonMonoxide);
                precisionGasMeasured = true;
            } else if (event.matches(DroneEventObject.droneEventType.PRESSURE_MEASURED)) {
                // drone.pressure_Pascals, drone.pressure_Atmospheres, drone.pressure_Torr
                logSample(PRESSURE_SENSOR_ID, drone.pressure_Pascals);
                pressureMeasured = true;
```

```
                } else if (event.matches(DroneEventObject.droneEventType.REDUCING_GAS_MEASURED)) {
                    logSample(REDUCING_GAS_SENSOR_ID, drone.reducingGas_Ohm);
                    reducingGasMeasured = true;
                } else if (event.matches(DroneEventObject.droneEventType.RGBC_MEASURED)) {
                    // drone.rgbcLux, drone.rgbcColorTemperature, drone.rgbcClearChannel, rgbcBlueChannel, rgbcGreenChannel,
                          rgbcRedChannel
                    logSample(COLOR_SENSOR_ID, drone.rgbcLux, drone.rgbcColorTemperature,
                    drone.rgbcClearChannel, drone.rgbcBlueChannel, drone.rgbcGreenChannel, drone.rgbcRedChannel);
                    rgbMeasured = true;
                } else if (event.matches(DroneEventObject.droneEventType.TEMPERATURE_MEASURED)) {
                    // drone.temperature_Celsius, drone.temperature_Fahrenheit, drone.temperature_Kelvin
                    logSample(TEMPERATURE_SENSOR_ID, drone.temperature_Celsius);
                    temperatureMeasured = true;
                }
            }
        };
        SensorDroneDataCollectionTask.drone.registerDroneListener(mDroneEventHandler);
    }
    this.timeout = timeout;
    this.debug = debug;
}
private boolean allDataCollected() {
    return (temperatureMeasured || !measuring[0]) &&
            (rgbMeasured || !measuring[1]) &&
            (reducingGasMeasured || !measuring[2]) &&
            (pressureMeasured || !measuring[3]) &&
            (precisionGasMeasured || !measuring[4]) &&
            (oxidizingGasMeasured || !measuring[5]) &&
            (irTemperatureMeasured || !measuring[6]) &&
            (humidityMeasured || !measuring[7]) &&
            (capacitanceMeasured || !measuring[8]) &&
            (altitudeMeasured || !measuring[9]) &&
            (batteryMeasured || !measuring[10]);
}
@SuppressWarnings("ResultOfMethodCallIgnored")
private void logSample(int sensorId, double... sensorValues) {
    if (sensorValues.length == 0 ||
            (sensorValues.length == 1 && (Double.isNaN(sensorValues[0]) || !Double.isFinite(sensorValues[0])))) {
        // We dont care about these if the data is not defined
        return;
    }
    StringBuilder sampleSb = new StringBuilder(String.format(Locale.ENGLISH, "%d;%s;%d",
            System.currentTimeMillis(), macAddress, sensorId));
    for (double v : sensorValues) {
        // Handle the case when data is not finite
        if (Double.isNaN(v) || !Double.isFinite(v)) {
            sampleSb.append(";");
        }
        else {
            sampleSb.append(String.format(Locale.ENGLISH, ";%f", v));
        }
    }
    System.out.println(sampleSb.toString());
}
private void reset() {
    batteryMeasured = false;
    altitudeMeasured = false;
    capacitanceMeasured = false;
    humidityMeasured = false;
    irTemperatureMeasured = false;
    oxidizingGasMeasured = false;
    precisionGasMeasured = false;
    pressureMeasured = false;
    reducingGasMeasured = false;
    rgbMeasured = false;
    temperatureMeasured = false;
    if (validSensor(ALTITUDE_SENSOR_ID)) drone.measureAltitude();
    if (validSensor(CAPACITANCE_SENSOR_ID)) drone.measureCapacitance();
    if (validSensor(HUMIDITY_SENSOR_ID))drone.measureHumidity();
    if (validSensor(IR_TEMPERATURE_SENSOR_ID)) drone.measureIRTemperature();
    if (validSensor(OXIDIZING_GAS_SENSOR_ID)) drone.measureOxidizingGas();
    if (validSensor(PRECISION_GAS_SENSOR_ID)) drone.measurePrecisionGas();
    if (validSensor(PRESSURE_SENSOR_ID)) drone.measurePressure();
    if (validSensor(REDUCING_GAS_SENSOR_ID)) drone.measureReducingGas();
```

```
        if (validSensor(COLOR_SENSOR_ID)) drone.measureRGBC();
        if (validSensor(TEMPERATURE_SENSOR_ID)) drone.measureTemperature();
        if (validSensor(BATTERY_VOLTAGE_SENSOR_ID)) drone.measureBatteryVoltage();
    }
    @Override
    public void run() {
        if (!drone.isConnected) {
            drone.btConnect(macAddress, debug);
        }
        if (!drone.isConnected) {
            System.err.println("Connection Failed to drone " + macAddress + " !");
            return;
        }
        long startTime = System.currentTimeMillis();
        reset();
        while (drone.isConnected && !allDataCollected() && (System.currentTimeMillis() - startTime < timeout)) {
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    void disconnect() {
        drone.disconnect(debug);
    }
}
```

```
//sensordrone-air-quality-logger/src/main/java/fi/helsinki/cs/sensordrone/air/quality/logger/SensorDroneAirQualityLogger.java
package fi.helsinki.cs.sensordrone.air.quality.logger;

import org.apache.commons.cli.*;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class SensorDroneAirQualityLogger {
    public static void main(String[] args) {
        // create Options object
        Options options = new Options();
        // add t option
        options.addRequiredOption("d","delay", true, "The delay for collecting the data from the sensordrone");
        options.addRequiredOption("m","mac", true, "The mac address of the sensordrone");
        options.addOption("timeout", true, "The timeout set to collect the data");
        options.addOption("debug", "Put the log to debug");
        // create the parser
        CommandLineParser parser = new DefaultParser();
        try {
            // parse the command line arguments
            CommandLine cmd = parser.parse(options, args);
            long delay = Long.parseLong(cmd.getOptionValue("d"));
            String macAddress = cmd.getOptionValue("m");
            long timeout = Long.parseLong(cmd.getOptionValue("timeout", "10000")); // default of 10 seconds
            boolean debug = cmd.hasOption("debug");

            final ScheduledExecutorService scheduler =
                    Executors.newScheduledThreadPool(1);
            final SensorDroneDataCollectionTask task = new SensorDroneDataCollectionTask(macAddress, timeout, debug);
            final ScheduledFuture<?> sensorDroneCollectionHandler =
                        scheduler.scheduleAtFixedRate(task,0, delay, TimeUnit.MILLISECONDS);
            Runtime.getRuntime().addShutdownHook(new Thread(() -> {
                sensorDroneCollectionHandler.cancel(true);
                if (!debug) {
                    // Create a stream to hold the output
                    ByteArrayOutputStream baos = new ByteArrayOutputStream();
                    PrintStream ps = new PrintStream(baos);
                    // Tell Java to use your special stream
                    System.setOut(ps);
                }
```

```
                task.disconnect();
            }));
        }
        catch (ParseException exp) {
            // oops, something went wrong
            System.err.println(exp.getMessage());
            HelpFormatter formatter = new HelpFormatter();
            formatter.printHelp( "gradle run", options);
        }
    }
}
```

Appendix 3. Sensordrone: Sample data stream

```
1542912225889;00:17:e9:50:e1:75;8;26.669067
1542912226004;00:17:e9:50:e1:75;8;26.676697
1542912226107;00:17:e9:50:e1:75;4;102004.000000
1542912226158;00:17:e9:50:e1:75;2;359.388367;2444.372559;993.168152;162.802582;411.344727;525.808472
1542912226578;00:17:e9:50:e1:75;1;24.664951
1542912226653;00:17:e9:50:e1:75;8;26.630920
1542912226699;00:17:e9:50:e1:75;4;102004.000000
1542912226749;00:17:e9:50:e1:75;2;357.757080;2444.406982;989.552246;162.802582;410.051208;524.544495
1542912226824;00:17:e9:50:e1:75;1;24.664951
1542912233815;00:17:e9:50:e1:75;8;26.051086
1542912233916;00:17:e9:50:e1:75;4;102008.000000
1542912233929;00:17:e9:50:e1:75;2;141.924881;2580.425293;397.749390;68.984146;162.985641;199.706085
1542912233983;00:17:e9:50:e1:75;1;24.825829
1542912244174;00:17:e9:50:e1:75;8;25.326294
1542912244188;00:17:e9:50:e1:75;6;36837000.000000
1542912244199;00:17:e9:50:e1:75;4;102004.000000
1542912244250;00:17:e9:50:e1:75;2;88.932762;2377.018799;260.345062;42.770168;104.776489;139.035889
1542912244325;00:17:e9:50:e1:75;1;25.083229
1542912253803;00:17:e9:50:e1:75;8;24.601501
1542912253886;00:17:e9:50:e1:75;4;102004.000000
1542912253938;00:17:e9:50:e1:75;2;138.040741;2571.751221;386.901672;67.604462;159.105042;195.914200
1542912253987;00:17:e9:50:e1:75;1;25.276281
```

Figure 18: **Collected measurements sample from SensorDrone.**

Data is collected in a text file. Each line represents a reading of a sensor at a given timestamp. The values are separated by semicolons. The first value is the timestamp of the reading, the second value shows the MAC address of the Sensordrone, the third value is the ID of the sensor, the last value is the reading(s) value that the sensor generated.