UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Data Capsule
Generic and flexible representation of unstructured data in Cloud-Edge systems as support for autonomous robots

Radu-Dumitru Stochițoiu

Thesis advisor:

Prof. PhD. Eng. Florin Pop

BUCHAREST

2018

TABLE OF CONTENTS

ABSTRACT

In a world of fast development where data gathering and data mining have become a focal point in any domain, we propose a way to serialize and collect raw and heterogeneous data, to analyze, manipulate and store it in an efficient and scalable way and to make use of the advances in Cloud-Edge system architectures research. Machine Learning algorithms are used in order to make use of the data stored in Cloud, with real-time feedback, based on a flexible half-duplex communication pattern. Based on the experimental results, we bounded some of the parameters used in the face recognition algorithms, these results helping us to get a better grasp of the offline Cloud management.

1 INTRODUCTION

1.1 Context and motivation

As the number of devices drastically increased over the last decade, there are several predictions that in 2020 there will be more than 50 billion devices, which means more than 6 devices per person. This changes the way we currently handle data and communication. Cloud computing opened new ways of handling data as there is a steady increase in the number of companies which reorganized their communication, storage and processing architectures.

1.2 Thesis objectives

As there will be more devices, there will also be more communication and, at the same time, more raw data. The architecture that gained a lot of popularity lately, Cloud computing, will not be able to sustain such a high traffic in the IoT domain, where data is created and consumed in real-time, so we propose working with an architecture already considered in IoT, which has a new layer between Cloud and Edge, called Fog.

Also, considering that data coming from different locations, with different context and going to different sensors is heterogeneous, we define a data type called Data Capsule to serialize and simplify communication between Edge, Fog and Cloud.

1.3 Proposed solution

We propose a generic and flexible data type named Data Capsule for the purpose of serializing unstructured heterogeneous data. It should be used in Cloud-Edge systems working with autonomous robots which can periodically or randomly send data between themselves or in the Cloud storage.

1.4 Thesis structure

This thesis comprises the theoretical definitions for our proposed solution alongside the technologies used in order to achieve it and a practical proof of concept. Chapter number 2, Related work, presents how heterogeneous data is already handled in Cloud-Edge systems and what we can do to achieve the current performances offered by the state of the art. In chapter number 3, Data Capsule - a generic and flexible representation of unstructured data in Cloud-Edge, we define a new data type for Cloud-Edge systems, present the ways communication can be made in a transparent and efficient way and create a link between autonomous robots and Cloud storages. Chapter number 4, Implementation details, presents the technologies, implementation trade-offs and the workflows we used in order to achieve a valid proof of concept. In chapter number 5, Experimental results, we present some of the results of the machine learning part used for face recognition. What could be improved and other topics about things that can be done on top of our implementation, are presented in chapter 6, Conclusions and future work.

2 RELATED WORK

2.1 Data management in Cloud-Edge systems

The current concern in Cloud-Edge systems is data overflow as the number of IoT devices is growing exponentially and all of them acquire lots of raw data, summing to a huge number of unordered pieces of data. Most of this data has to be filtered, processed and analyzed properly. Nowadays, it is very important to think about all kinds of criteria and benchmarks of a system: latency, speed, privacy, bandwidth etc. This is where Cloud-Edge systems have an advantage.

As we can understand from [8], a good approach of handling the data streams would be separating each of them in a tuple, as we do with our Data Capsule. All of the streams are of unknown size, can carry any amount of data capsule, it only works when there is Cloud connectivity and the data comes in an unordered way.

We talk about an architecture that separates Edge from Fog and Cloud. The Edge layer consists of acquiring raw data from sensors or prelearned mechanism. Next, the Fog layer is actually the bridge between Edge and Cloud, so with a stream of data coming to it, it can preprocess, filter and modify data along the way. The Cloud layer is the backup and storage fixed point in the architecture.

We talk about an architecture that separates Edge from Fog and Cloud. The Edge layer consists of acquiring raw data from sensors or prelearned mechanism. Next, the Fog layer is actually the bridge between Edge and Cloud, so with a stream of data coming to it, it can preprocess, filter and modify data along the way. The Cloud layer is the backup and storage fixed point in the architecture.

These communication do not know how to communicate one to another so they need a Message Broker, who can handle messages coming from any stream going to any layer.

The first thing the Message Broker does is to separate the communication between the

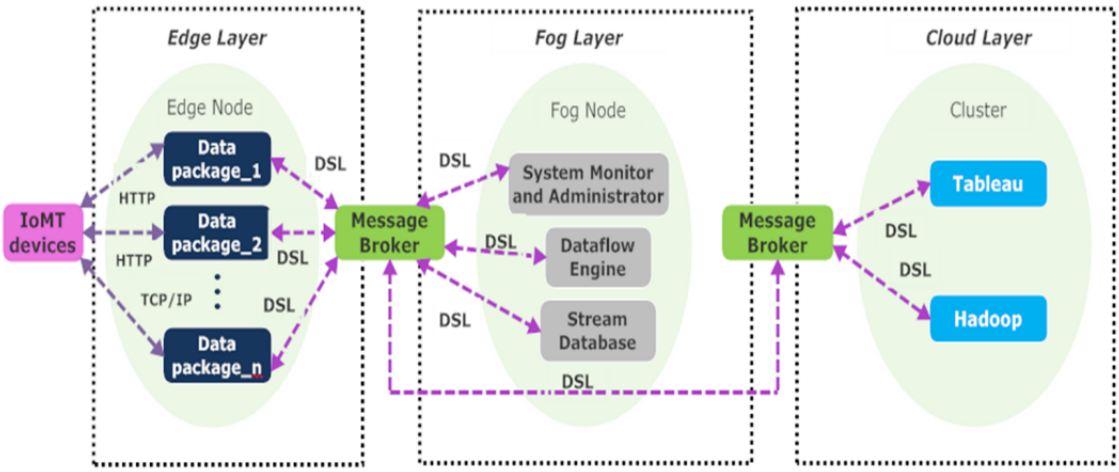

Figure 1: Cloud-Fog-Edge architecture [8].

Edge layer and the Fog layer, so the services that are called are perfectly separable. They receive and manage streams of messages at predefined periods of time. It can be ten seconds, three hours, two weeks etc. The second ability of the Message Broker is to break messages into significant data collections and route them from the Fog layer to Cloud. The third thing, that is also noticeable, is that the Message Broker allows messages to flow inside a specific layer, in this way enabling heterogeneous migration of data inside the Edge layer, and also significant data capsule migration inside the Fog layer.

2.2 Heterogeneous data management

When we talk about heterogeneous data we talk about an undefined number of sources, about data that is not well-structured, about data that is not structured at all, about data with multiple raw dimensions and, usually, about gigantic amounts of data. In contrast to the structured, homogeneous data, we need to take in consideration how to organize a big data storage, its management, the latency of data handling offline or in real time situations, and of course, the communcation. This means that we need to think about distributed data storages, a single unified interface for data accessing, serving, indexing and fast positioning for lookups, and other issues which we talk about more in the ROBIN-Cloud section, as process scheduling, distribution of tasks and their concurrent execution, security and the definition of a serialized, structured data type, called Data Capsule.

Because of the hardware and processing limitations of the robots, the online computation is much more demanding than the offline one. It is a problem that could be solved by reusing data, resulting in incremental computing, but being not always possible, we reach recalculation steps.

In this paper [10], they propose a classification of heterogeneous data in two main subsets:

- dynamic data;
- static data.

Static data represents the read-only part of the data and the dynamic data is every collection of samples that can be read or written, be it raw data or already processed data.

Regarding the static data, we need an architecture that handles offline computations in an efficient way, has enough storage and can be used in a series of specific microservices. For parallelizing and distributing the context, we can use Google's implementation of MapReduce[3], which is applied in Hadoop[4] and that is to divide data into shards[1] and assign tasks to multiple devices (computers) which should result in a boost of efficiency.

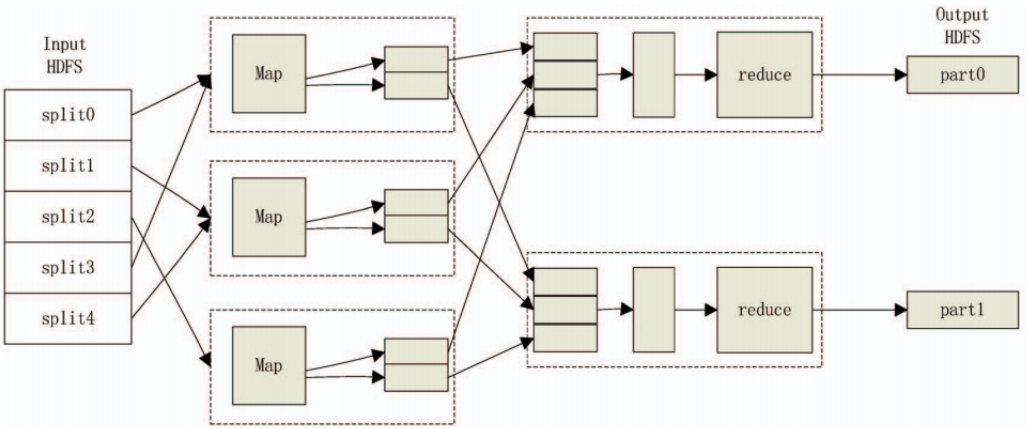

Figure 2: MapReduce architecture [10].

MapReduce is a programming model which allows in a simplified way to automatically distribute programs defined for one computer on any number of clusters that can be executed in parallel or concurrent. The idea of slicing the data into shards is what MapReduce can handle in a very efficient way using a function called Map. After Map

[1] A database shard is a horizontal partition of data in a database or search engine

functions produce outputs, they are sent and combined before the Reduce job. If there are any forcefully canceled or failed jobs, Hadoop MapReduce takes care of it by re-scheduling and running them. The entire process is shown in Figure 2.

As for the online computation, there will probably be huge amounts of data going to Cloud so we need caching systems.

Regarding the dynamic data, we need caching, indexing and scalable structuring that can be achieved by using Apache Storm, a framework made for stream computing which is used in the industry for analytics, statistics and management, all in real-time processing.

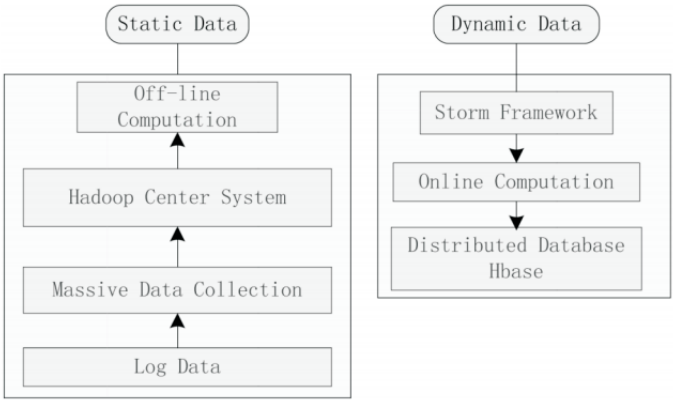

Figure 3: Offline and online computation [10].

3 DATA CAPSULE: A GENERIC AND FLEXIBLE REPRESENTATION OF UNSTRUCTURED DATA IN CLOUD-EDGE

3.1 Data modeling

This section formally describes the model used throughout the paper, containing the way in which two or more robots interact with each other, their connection with Cloud and the Data Capsule.

Advances in the IOT[1] domain transformed communication between robots and, essentially, hardware devices into a fast-paced exchange of data with footprints used for easier tracking. In this section we propose a new way of tracking, controlling and managing each of these devices.

3.1.1 Robots

We define $\mathcal{R}$ as the robot set modeling N heterogeneous agents with different characteristics such as sensors, processing power, peripheral entities which may acquire data, where $N = |\mathcal{R}|$. A robot $r \in \mathcal{R}$ has the following formal description:

$$i = (id, S_i, C_i) \tag{1}$$

where:

- id is a number that uniquely identifies a robot;
- S_i is the set of sensors that the robot $i \in \mathcal{R}$ is equipped with;
- $C_i = (C_{i1}, ..., C_{iN})$ is the *resource capacity vector* of the robot $i \in \mathcal{R}$, where C_{ir} is the amount of resource r available on the robot with $C_{ir} > 0$.

[1] Internet of Things

3.1.2 Data Capsule

Data flow management is one of the most important features of our proposed architecture. In this section we describe a structured format for data representation and storage, called Data Capsule. We define this format in a way to obtain a generic representational specification for a data unit in a time series database of unstructured data. We define a set of data capsules as following:

$$D = \{D_i | D_i = (t_i, c_i, m_i, d_i)\} \tag{2}$$

where:

- t_i is the timestamp for the current Data Capsule;
- c_i is an ordered tuple of strings (sub-contexts) and defines the context of the Data Capsule;
- m_i is a set of $(key, value)$ pairs of comparable keys and values that define the Data Capsule metadata;
- d_i is the data content that we want to store inside the Data Capsule.

We defined this structure such that it allows retrieving Data Capsules based on context match, timestamp intervals and meta-data based selections. Let us define an operator that selects the Data Capsules in a context. We consider that operator to be defined as it could take account of a special sub-context $(*)$ that matches any existing sub-context on a specific hierarchy level. Let that operator be:

$$SearchContext(D_{set}, c_{match}) = \{D_i \in D_{set} | \forall s_j \in c_i \} \\ \wedge \forall s_k \in c_{match} s.t. j = k \wedge (s_j = s_k \vee s_k = *)\} \tag{3}$$

where:

- D_{set} is the set of Data Capsules that we are searching on;

- c_{match} is the context that should be matched and may contain some wild-card (*) sub-contexts;
- c_i is the context of D_i;
- $*$ is a special sub-context that matches any sub-context.

Let us define an operator that extracts certain Data Capsules by meta-data:

$$SearchMeta(D_{set}, m_{match}) = \{D_i \in D_{set} | \forall (key_j, value_j) \in m_i, \\ \exists (key_k, value_k) \in m_{match} s.t. key_j = key_k \wedge value_j = value_k\} \quad (4)$$

where:

- D_{set} is the set of Data Capsules that we are searching on;
- m_{match} is the set of meta values that should be matched;
- m_i is the meta-data of D_i.

Since we aim to describe some time series of data, an operator that selects all the Data Capsules for a context in a specified time interval is needed. Let that operator be:

$$SearchInterval(D_{set}, c_{match}, t_{min}, t_{max}) = \\ \{D_i \in SearchContext(D_{set}, c_{match}) | t_i \geq t_{min} \wedge t_i < t_{max}\} \quad (5)$$

where:

- D_{set} is the set of Data Capsules that we are searching on;
- c_{match} is the context that should be matched and may contain some wild-card (*) sub-contexts;
- t_{min} and t_{max} are the bounds of the time interval;
- t_i is the timestamp for the Data Capsule D_i;
- $SearchContext$ is the operator that we described at (3).

3.2 Data collection and data accuracy

As today's raw data can be consisting of noise, machine-generated data, GIS[2] data (latitude, longitude), multimedia data (relevant sounds, images), sensor data (pollution degree, values given by an accelerometer) etc. we define three kinds of data collection purposes:

- **local** data;
- **Edge** data;
- **Cloud** data.

Sometimes the last two, or in rare cases, even all of them, may coincide, but we talk about this in the next section.

Based on the robot's processing power and its available amounts of resources, defined at (2), as well as the usefulness of the data to read, we keep the data used for local processes (for example, data used for a robot's moving decisions) on the device and send everything else, be it useless or not, to a gateway which would distribute the data to the Edge nodes. But for us to decide whether the data is relevant or not, implies the use of a DPS[3] with the predominant function of preprocessing the data gathered in real-time-based systems.

Data preprocessing is divided into multiple steps, as it follows:

- Data cleaning;
- Data integration;
- Data transformation;
- Data reduction.

Data cleaning represents the process of removing noise and misread or data that misfits from our input. Data that is collected by a robot may be noisy, inconsistent or incomplete. Noisy data is any collection which contains similar values to the targeted

[2]Geographic information system

[3]Data Processing System

or expected ones, or erroneous data. Inconsistent data is any collection which does not follow a specific set of rules, established beforehand, or data that contains differences in the way it is labeled. Incomplete data is any collection which lacks keys, values, attributes that are of interest.

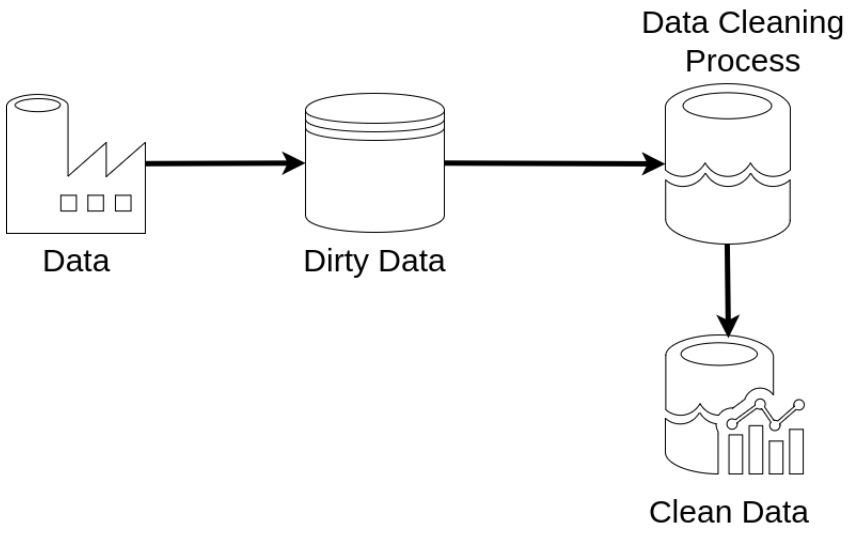

Figure 4: Data cleaning process [12].

Among the usual sources of data **incompleteness** we can enumerate:

- inconsistent data that went through a previous layer which cleaned it;
- data not relevant for the current purpose;
- data not transmitted by a faulty machine;
- lack of data availability;
- censored data.

If there is some missing data, like a label, or something decisive for a collection of information, we can either:

- use a default value;
- use a Bayesian network to find the most probable missing data;
- use linear regression to find the most probable missing data;
- drop the whole collection of data.

As for the data **inconsistency**, there are a number of typical reasons:

- human or machine error;
- encoding and naming conventions;
- different storing formats.

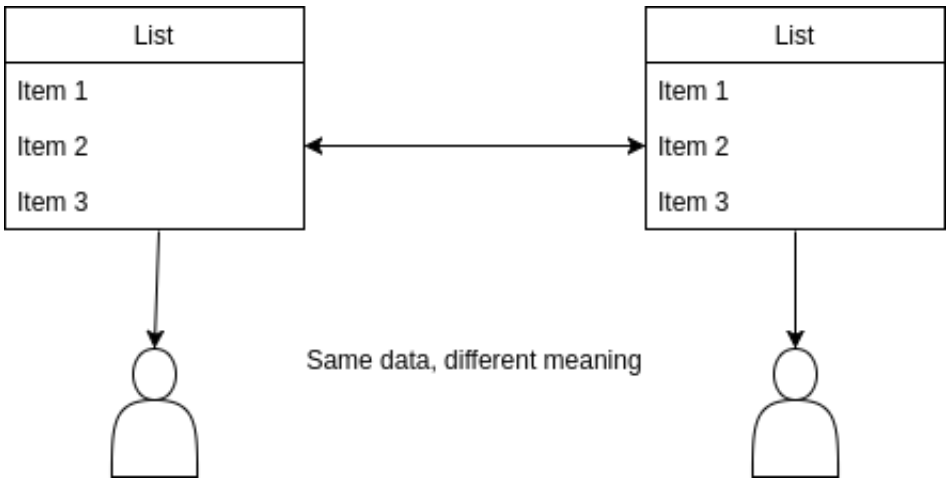

Figure 5: Data inconsistency.

Noise is usually understood as the instability of data. Some of its causes may be:

- random noise, which is unavoidable in a non-perfect system;
- values that do not belong to the data set;
- mathematical, human or machine errors.

Some of the used ways of removing noise are:

- machine learning: "As the name implies, linear regression solves a regression problem.In other words, the goal is to build a system that can take a vector $x \in \mathcal{R}^n$ as input and predict the value of a scalar $y \in \mathcal{R}$ as its output[6]."; for our case, we have lots of examples with noisy and, mostly not noisy, data and we want to fit a line between all of these points and consider that line the targeted value;
- "a spectral clustering technique for noisy data. Our core idea was to decompose the similarity graph into two latent factors: sparse corruptions and clean data. We

jointly learned the spectral embedding as well as the corrupted data. We proposed three different algorithmic solutions using different Laplacians. Our experiments have shown that the learned embeddings clearly emphasize the clustering structure and that our method outperforms spectral clustering and state-of-the-art competitors[3].";

- binning: considering the vicinity of each noise example, we just divide the values into buckets (bins); for example, if $data = 3.4 \wedge (3 \leq x < 4 \rightarrow x \in buckets[3]) \rightarrow data \in buckets[3]$.

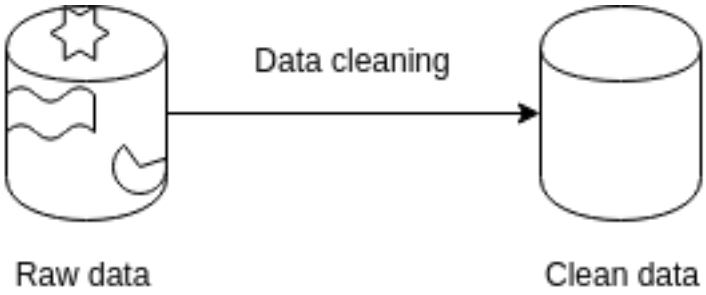

Figure 6: Data cleaning.

Data integration represents the process of concatenating samples that fit together, in this way reducing the number of samples used and also using more specific data.

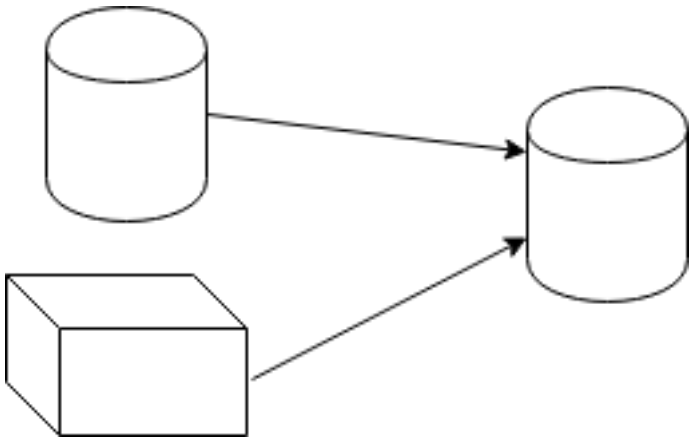

Figure 7: Data integration.

Data transformation is the way we keep our data in a required threshold, making it scalable.

Data reduction means compressing the data in a readable, simplified way.

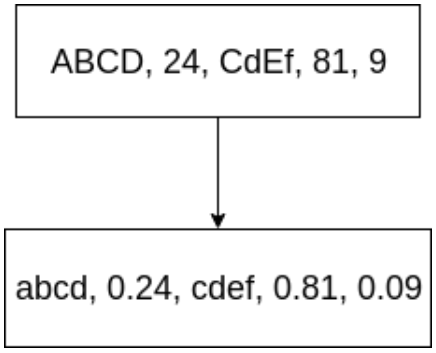

Figure 8: Data transformation.

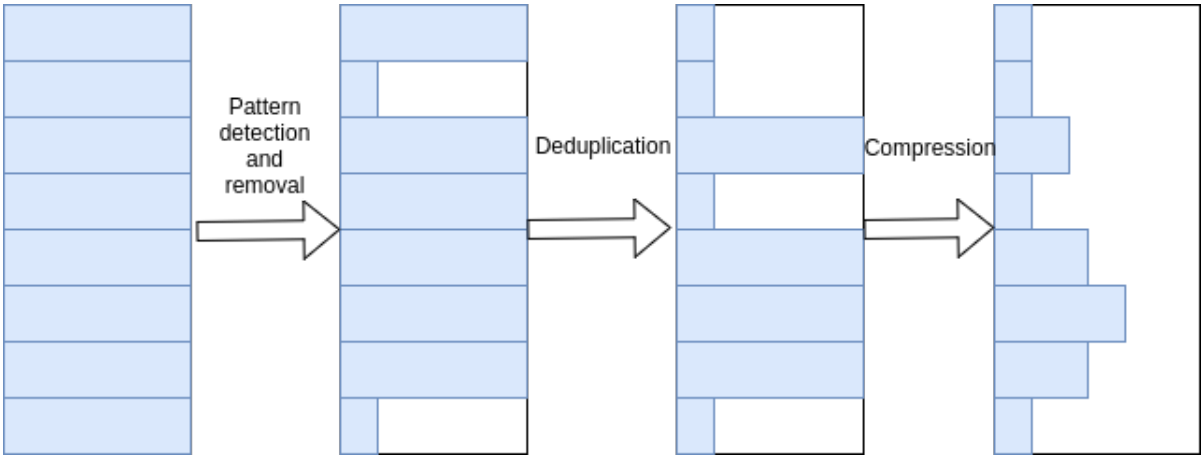

Figure 9: Data reduction.

All of the steps presented above have a main goal of analyzing and ordering the raw data towards a higher quality so it can be used in training phases of Machine learning or KDD[4]. Noisy, inconsistent or wrongfully labeled input (for example: {human: yes, hasTail:yes}) can be eliminated in the preprocessing step, therefore making the data collection and mining more efficient.

3.3 Content delivery: communication between Edge and Cloud

3.3.1 Edge

The *Edge* layer consists of robots (devices) that collect, analyze and preprocess data from the surrounding environment and format it to *Data Capsules* which are sent to

[4]Knowledge Discovery in Databases

APs[5] based on their context. "The term *Edge* refers to pushing the data processing tasks to the Edge of the network rather than using a centralized Cloud-based approach[11]."

Data collected by the robots can be one of the following types:

- GIS data;
- sensor data;
- multimedia (sounds, images);
- machine-generated data;
- location (latitude, longitude);
- documents;
- noise;
- online social interactions.

Robot cooperation is considered possible so they can talk to each other with the purpose of solving more demanding tasks and also to offload specific jobs to other entities, devices or robots. This is the model described in this subsection.

In this way, some of the devices located at the Edge of the network can communicate with each other, but this is not always true. Some Access Points may be so far away one from another or the task may be so hard to process that the only available solution is to send the data upwards in *Fog* towards *Cloud*. Being a distributed system, this allows data to be divided into chunks and rebuilt on their way back to the robots.

3.3.2 Fog

The *Fog* layer consists of agents which are Access Points for the robots. The Access Points are responsible to receive data from the robots and to forward it to *Cloud*. After the data or task has been handled in Cloud, the Access Points must forward the response to the robot that sent the original request. If needed Access Points can process locally as they are configured with a container (e.g., Kubernetes) deployment and can run simple microservices. Alternatively, the Access Points can forward the data to other Access Points, as they are connected in a peer-to-peer network.

[5]Access Points

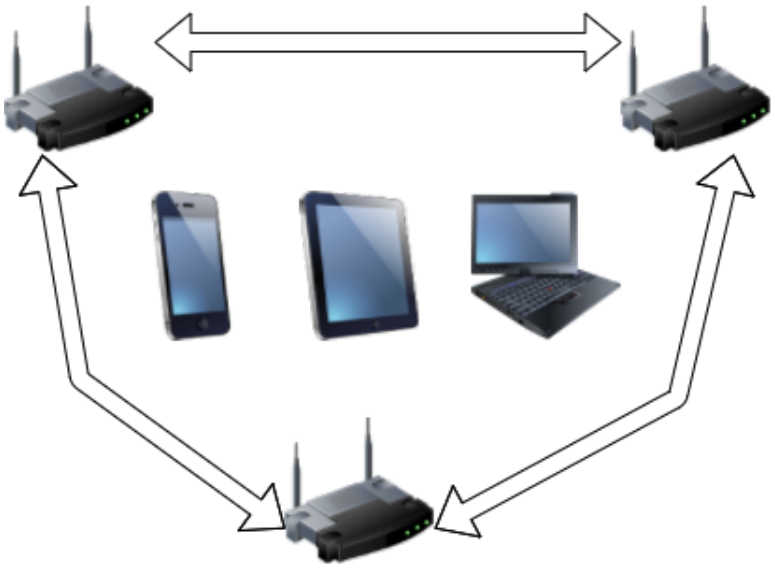

Figure 10: Devices surrounded by Edge access points that can communicate.

"Fog computing, which extends Cloud computing to the Edge of the network can effectively solve the bottleneck problems of data transmission and data storage. [...] Fog computing devices are located between endpoints and the traditional Cloud, thus resources and services are available and are closer to the end-users, and the delays induced by service deployments can be reduced. Compared with the Cloud computing concept, which is more centralized, Fog computing provides resources and services in a distributed way" [5]

3.3.3 Cloud

The *Cloud* layer consists of multiple components: a message processing queue component which stores data and tasks received from Access Points, an auto-scalable microservices component which manages interaction between the message processing queue component and the upper components of storage and processing. The data received from the robots will be saved in the storage component which is a distributed database and will be handled by the processing platform which will analyze the data using different algorithms.

Cloud computing can be divided into the following [13] services:

- Infrastructure as a Service (IaaS);

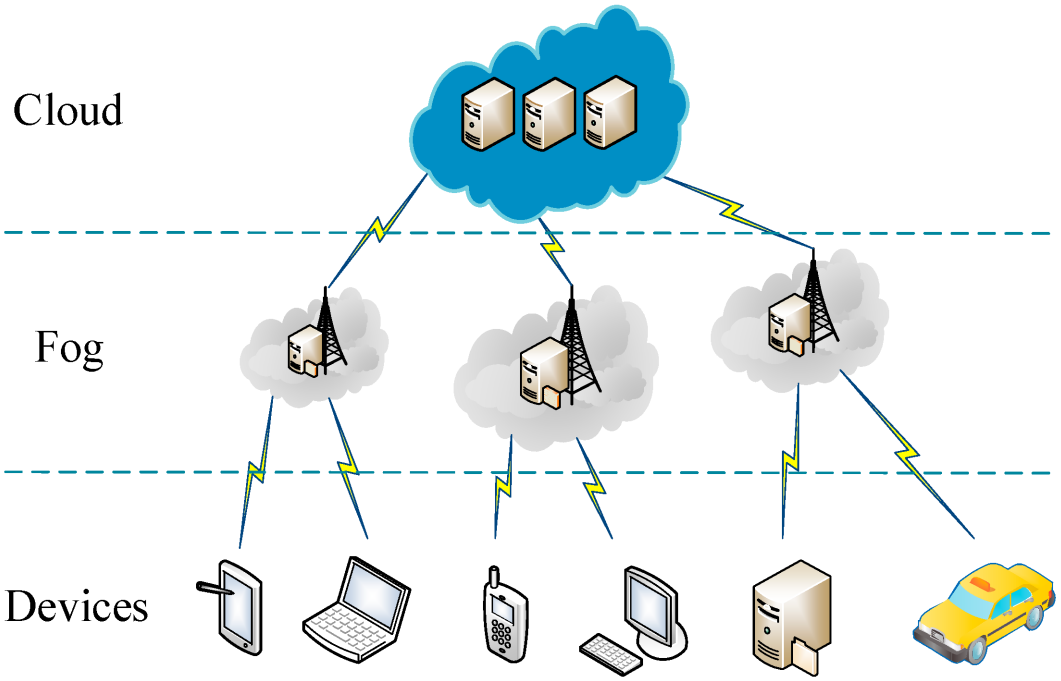

Figure 11: Devices on the Edge level communicate upwards to Fog which has a peer-to-peer connection and can communicate horizontally [5].

In this method the clients can directly access all the infrastructure items of the Cloud, such as network capabilities, storage, processing power;

This is established via virtualization [6] so the amounts of resources can be changed based on the momentary needs.

- Platform as a Service (PaaS);

  PaaS allows consumers/users to develop their projects and applications using a platform situated on Cloud with its own resources.

- Software as a Service (SaaS).

  SaaS is an environment which allows users to release their applications and projects so they can be accessed through the Internet.

We are proposing a SaaS solution as it would fit our needs, by having an already initialized environment so we can just deploy any number of robots.

[6]Virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources.

3.4 Data handling for GIGEL: towards an autonomous robot linked with Cloud

The message queue processing component will consist of a broker which will be responsible for managing multiple queues for different applications and data types. RabbitMQ is an open source message broker which supports messaging in an asynchronous regime with various protocols, message queueing, acknowledgement for every successful delivery and routing to brokers and queues in a flexible way. It also supports deployment as clusters for high availability and throughput [15].

The auto-scalable microservices component will consist of a Kubernetes deployment that will manage microservices which will run in containers. Kubernetes is a system provided by the open-source community for automating and making deployment much easier, more scaling, and management of containerized applications. It manages and groups containers with dependencies that make up an application into various logical units for the purpose of managing and discovering called pods and each pod will run multiple microservices as containers. Microservices can have various roles which can range from interacting with the message queues, service discovery or starting jobs on the processing platform [2].

The storage component will consists of Cassandra, which is a distributed NoSQL system with the purpose of managing databases and handle huge amounts of data across many machines and, at the same time, providing scalability and high availability. It is based on a decentralized architecture without central points of failure and it ensures fault tolerance by having data replication to multiple nodes [7]. Each entry in the database will abide to the following data format:

$$DataFormat = (id, robot_{id}, timestamp, location, context, data) \tag{6}$$

The processing platform component will consist of multiple specialized distributed processing frameworks that will have implemented multiple algorithms. For classical MapReduce processing and for a distributed filesystem, Apache Hadoop can be deployed. For specialized machine learning algorithms and data analytics, Apache Spark

can be deployed and can be used together with HDFS and YARN from Hadoop. In order to have better resource management, Apache Hadoop is a framework that enables the processing of large datasets over a distributed cluster of machines. It is designed to scale up very fast from single servers to collections of thousands of machines organized in clusters, each machine offering local computation and storage. It ensures a high-availability mechanism where the library is designed to detect and manage handle failures at the application level. Apache Hadoop comprises the HDFS[7], Hadoop YARN which is the solution for resource management and job scheduling and Hadoop MapReduce which is a programming model that parallelizes data processing [14].

Apache Spark is used for big-data processing, handling very large scales of data collections, also being a very fast platform. It is based on an advanced DAG execution engine that ensures data flow in an acyclic way and in-memory computing based on the RDD[8] abstraction. Spark can be used for machine learning, data analytics, SQL and streaming. Spark uses the abstraction of RDD which is a collection of read-only objects divided across a set of devices (physical machines) that can be rebuilt if any of the partitions is lost. By default, RDDs are lazy and ephemeral (discarded from memory after used) but the persistence can be changed to be cached in memory or saved in a distributed file system. RDDs can be built from existing files, collections of objects, existing RDDs through functions or by altering the persistence of an existing RDD [16].

As an alternative in-memory computing platform to Apache Spark, Apache Ignite can be used.

Apache Ignite is an in-memory distributed database, caching and processing platform for transactional, analytical and streaming workloads which is capable of delivering in-memory speeds at petabyte scale [4].

Apache Mesos is a distributed system kernel that abstracts compute resources such as CPU, memory, storage from the machines in order to ensure resource sharing between multiple computing platforms. It ensures good scalability, high availability and fault-tolerance. Apache Mesos comes with a number of frameworks and applications stacks that can take advantage of its resource sharing capabilities and by using Mesos it is

[7]Hadoop Distributed File System

[8]Resilient Distribution Dataset

possible to multiplex distributed processing frameworks and to achieve higher scalability [9].

3.5 ROBIN-Cloud project

ROBIN-Cloud is a distributed computer platform for pervasive systems in the Cloud-Edge context.

Its main purpose is defined by the following items:

- creating of a support platform for collecting data from sensors of support systems for robots (e.g., IoT);
- providing processing and learning mechanisms combining Cloud models with devices close to the source of collection;
- providing support libraries for processing intelligent/semantic data;
- development of web-based services to economic agents interested in using the data stored at platform level.

Some of the ROBIN-Cloud project's objectives which concern the use of *Data Capsules* in an Edge-Fog-Cloud system are:

- design and build of a support platform for collecting and systematization of captured data from IoT systems and robots over the IoT-A ARM architecture;
- design and implementation of processing and distributed learning mechanisms based on sensor data in hybrid Cloud-Edge technology (components Complex Event Processing and Context Broker);
- developing a library of semantic data processing algorithms on Cloud platforms;
- designing and creating techniques, methods and algorithms for searching and extracting knowledge from the ROBIN-Cloud Platform Data Store;
- Cloud development of machine learning support (deep-learning algorithms, deep-reinforcement learning, Support Vector Machine etc.) specific to natural language processing, autonomous machines, and collaborative robots in the social environment;

- intelligent and adaptive planning by creating a set of algorithms for: Cloud processing, SLAM pairing, Autonomous machine route planning, NLP, etc.

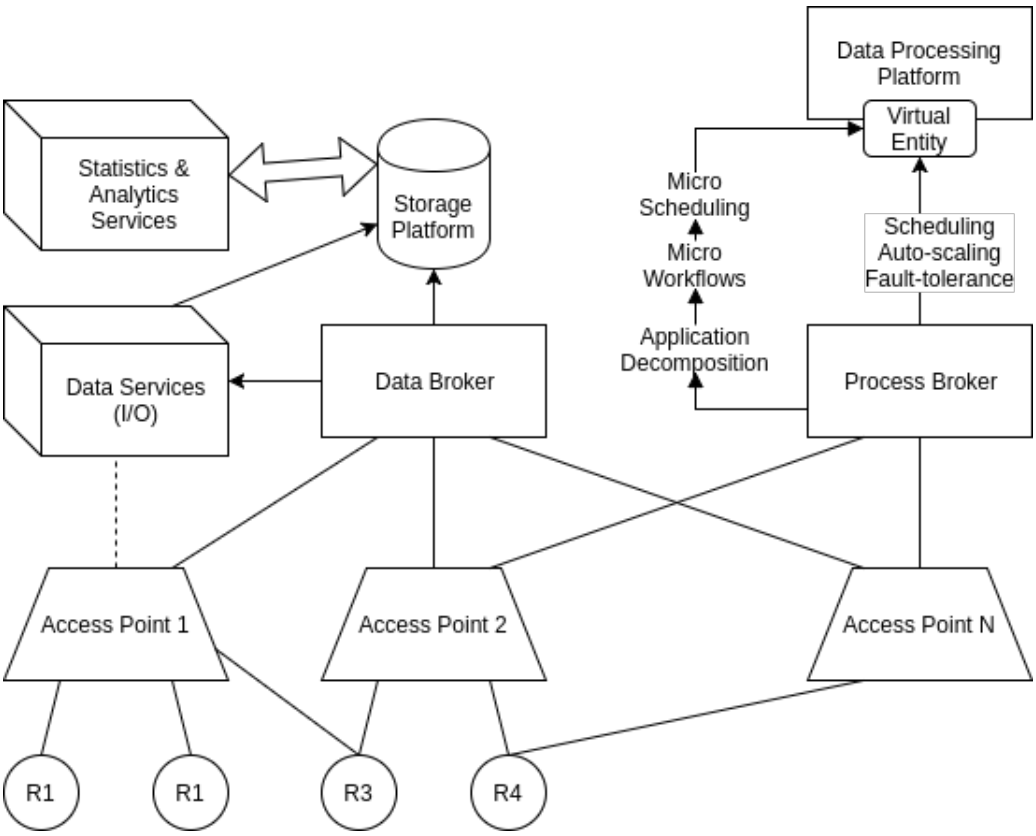

Figure 12: High level view of ROBIN-Cloud architecture.

4 IMPLEMENTATION DETAILS

4.1 Technologies

4.1.1 Building an autonomous robot: GIGEL

The hardware piece that represents the brain of this project's proof of concept is a Raspberry Pi 3 Model B, on top of which we set up a BrickPi board for the purpose of making the interactions with LEGO components easier. It uses an ARMv7 Quad Core Processor which runs at a maximum speed of 1.2GHz, which is enough and well-suited for the purpose of data collection, little processing and mostly sending Data Capsules to Fog. It contains a WiFi chip which helped us, firstly by letting us develop remotely, and secondly, by enabling various communications with different Access Points around the robot. It also has a Bluetooth Low Energy chip which can be used mostly for retrieving location data from Beacons or sending data in the network around. The four USB[1] 2.0 ports come in very handy as we connect additional microphones for the purpose of noise removal, or some extra speakers etc. One of the most important features is the availability of a CSI[2] port which we use to connect the Raspberry Pi camera. For web camera debugging we use the HDMI[3] port that comes in very handy. 64GB micro SD[4] storage and 1GB RAM suffice any robot in this architecture's needs.

BrickPi does an amazing job by bypassing all of the networking issues and usual limitations on a RaspberryPi board, also providing several ports for LEGO sensors, from which we use UDS[5] for example.

[1] Universal Serial Bus
[2] Camera Serial Interface
[3] High-Definition Multimedia Interface
[4] Secure Digital
[5] Ultrasonic Distance Sensor

4.1.2 Machine learning

All of the machine learning algorithms are executed in Cloud and the main framework chosen for these tasks is TensorFlow (in Python). We chose it because of its quick but steady development in the open-source area, so we could rely on many other people's work in our search for the best parameter tweaking.

We divide these ideas into two big ways of using data:

- **online** real-time data processing;
- **offline** latent data processing.

The online processing comprises the following:

- voice recognition;
- noise reduction;
- text to speech.

The offline processing comprises the following:

- facial recognition;
- object recognition and captioning.

4.1.2.1 Voice recognition

In order to recognize different patterns (e.g., "Remember this", "Exploration mode"), we need not only to use the Internet connection when it is available, but also provide an offline alternative. Some of the APIs[6] that we use and combine are:

- CMU Sphinx is a framework which does not need to know any information about the speaker, it can record and analyze continuously and it is mainly a recognition system. Behind the scenes, it heavily relies on HMMs[7] and on SLMs[8], for example an n-gram one (works offline);

[6] Application Programming Interfaces
[7] Hidden Markov acoustic Models
[8] Statistical Language Models

- Snowboy Hotword Detection uses a Deep Neural Network architecture to learn your hotwords and recognize them whenever needed (works offline);
- Google Cloud Speech API and Google Speech Recognition enable developers, and students in our case, to convert any kind of audio sample to text using neural network models. There are several supported languages, but we mostly rely on English, and its API is facile. (works both online and offline);
- Wit.ai is a very simple to use speech recognition as a service. It was acquired by Facebook some years ago so its development is steady at least (works online);
- Houndify API is a good alternative for developers because of its speech to text results using the Speech to Text Only domain. Houndify domains are used to respond to an audio or text query, including a transcription (works online);
- The IBM Speech to Text is a free API that produces transcripts of spoken samples in several formats. It can change the sampling rate or the broadband so it is customizable and definitely useful for our needs (works online);
- Microsoft Bing Voice Recognition converts real-time or latent audio samples from files or from another audio devices to text. The API is very useful because it can be used in phone, desktop or web apps (works online);

Among the Python libraries that we use for acquiring and analyzing data from the microphone, PyAudio is the one that we extensively use. As for the main way of audio encoding, we use FLAC[9]. There is a trade-off between the quality of the record and its size. We tend to give up on the storage constraints for bigger quality, as the noise removal or voice recognition algorithms mostly rely on clean data.

4.1.2.2 Noise reduction

Giving the fact that the LEGO motors used for the proof of concept are very loud, we need to make sure noise does not get into the voice recognition process. The first idea is to try separating the audio sample into relevant and not so relevant parts. We can do that by implementing a PCA[10] machine, which is a mathematical (or statistical) procedure used to transform a set of human or machine-discovered observations into a

[9]Free Lossless Audio Codec

[10]Principal Component Analysis

set of **principal components**, which are some linearly uncorrelated variables. It mainly uses orthogonal transformations so the computing is done in an efficient way, no matter the input.

Let's consider $\mathbf{X}$ a matrix representing the audio data. It has n_v lines representing the variables and n_o columns representing the observed data for each variable. We note $\mathbf{P}$ as the PCA matrix which is made of n_v lines and n_{ev} columns that represent the selected eigenvectors. Let's consider $\mathbf{C}$ the coefficient matrix representing each eigenvector's weight regarding each observation, so we can consider C_{jk} the weight of the j^{th} eigenvector regarding the k^{th} observation. X notes the columns of the matrix X stacked upon each other. Its covariance is kept in V and W has the same size as X and represents the weights of the matrix if the noise measurement would be independent [1].

Our main target is to minimize $\sum_{variable\ i, observation\ j} [\mathbf{X}_{ij} - PC]_{i,j}{}^2$.

4.1.2.3 Text to speech

In order to achieve both offline and online text-to-speech functionality we use pyttsx3, which is a Python TTS[11] library. We use Python3 and rely on this library's very low delay rate. There was no need to implement another algorithm instead of using one that's already open-source because it would imply lots of user knowledge which is very difficult to acquire.

4.1.2.4 Facial recognition and object captioning

The purpose is to recognize faces and objects so it can act according to its prediction in an assistive way. We use the official TensorFlow models to train our DNNs.

As our proof of concept, GIGEL, captures video data, we use a machine learning trained neural network to analyze every 32 frames per second (this ratio cand be easily modified).

Here are two examples of facial recognition trained with four total pictures,

[11] Text To Speech

two with glasses and two without glasses, two for each of us.

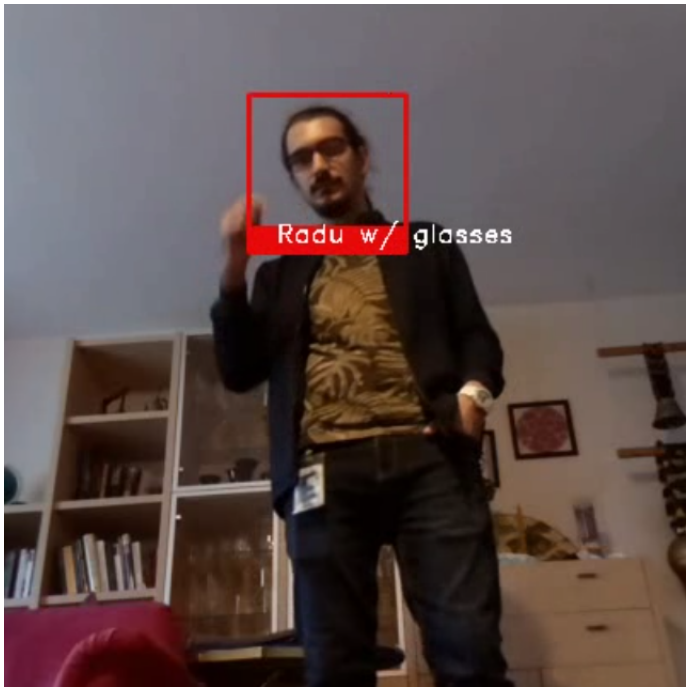

Figure 13: Radu's facial recognition (with glasses).

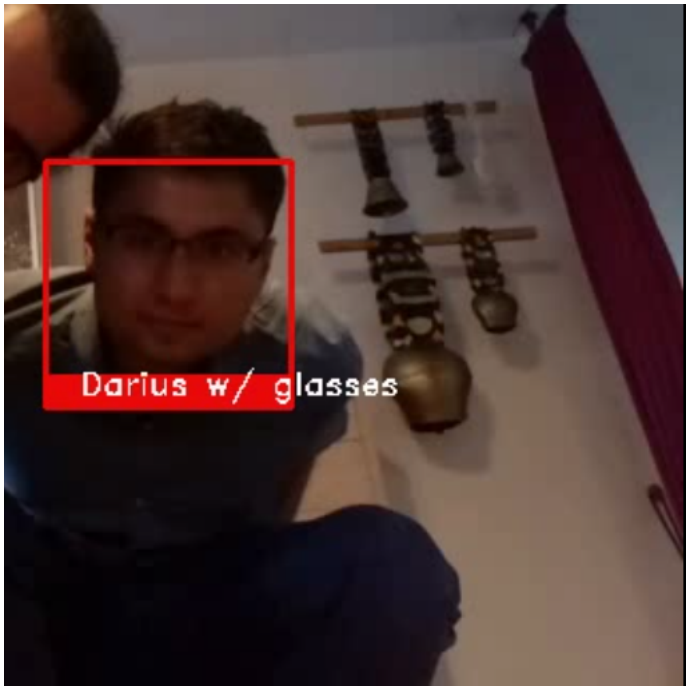

Figure 14: Darius's facial recognition (with glasses).

Thinking about our robot and its functionalities, the assistive keyword has enormous importance, as it could help and enhance the lifestyle of any user. Judging by the fact that we use a DNN, we can train it with multiple features, as having glasses or

not.

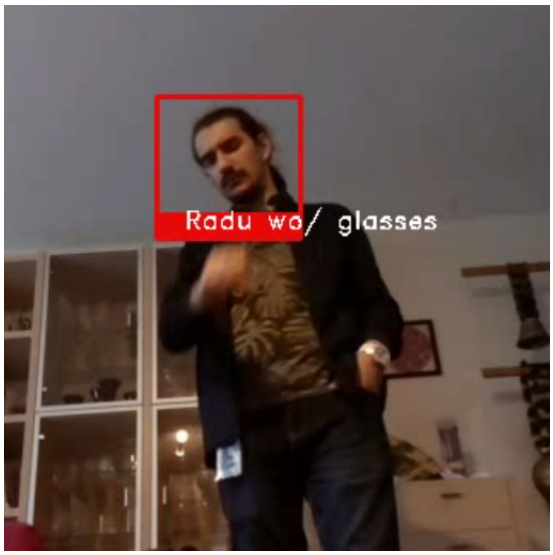

Figure 15: Radu's facial recognition (without glasses).

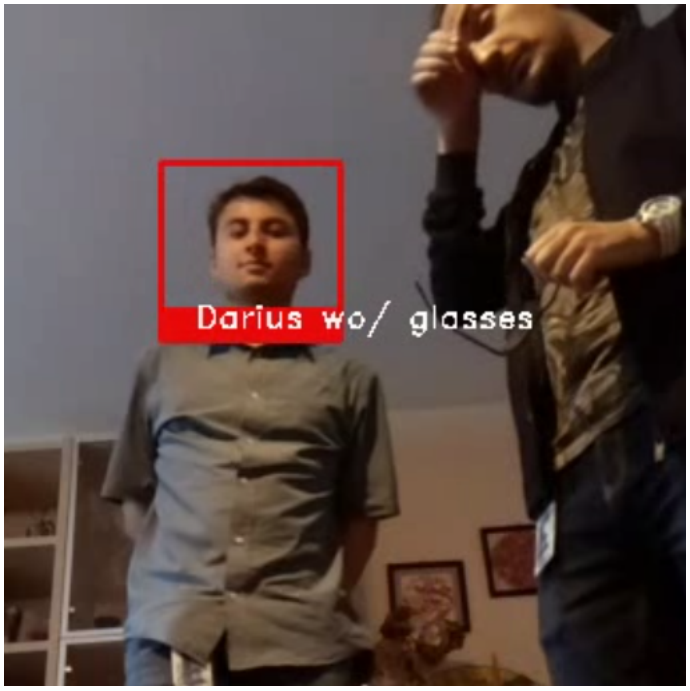

Figure 16: Darius's facial recognition (without glasses).

4.2 Implementation of Data Capsule in Edge devices

We defined the Data Capsule earlier at equation 2 and because of the discrepancy in its fields and the raw behavior data can adopt, we concluded in the need of a

language-neutral mechanism for serializing heterogeneous data. With this in mind, the best approach was to use a ProtoBuffer[12]. It is created and maintained by Google who extensively uses it, it's faster and much easier to implement and adapt to various data differences than any other serializing platform (e.g., XML). It can be used in C/C++, Java, Python, and many other programming languages. It can also be simply combined with gRPC[13], which is also developed by Google.

```
message DataCapsule {
    required Timestamp timestamp = 1;
    required Location location = 2;
    required Metadata metadata = 3;
    required RawData data = 4;

    message Timestamp {
        int64 seconds = 1;
        int32 nanos = 2;
    }
    message Location {
        double latitude = 1;
        double longitude = 2;
    }
    message Metadata {
        repetead string key = 1;
        repetead string value = 2;
    }
    message RawData {
        optional bytes data = 1;
    }
}
```

Notice that we define a special message type for all the fields we needed inside

[12]Protocol Buffer

[13]gRPC Remote Procedure Calls

a Data Capsule. Each of the numbers in the figure mean a tag inside the message for faster lookup. We define a timestamp as a correlation between the number of seconds and the number of nanoseconds, the location as a combination of latitude and longitude, the metadata as a map, ”repeated” meaning zero or more fields of the respective type and we define RawData as a sequence of bytes of random length.

We also need to define a gRPC service that can be called from any robot, simulation or not.

```
message SendRequest {
    required DataCapsule dataCapsule = 1;
}
message SendResponse {
}
message RetrieveRequest {
    optional Timestamp timestamp = 1;
    optional Location location = 2;
}
message RetrieveResponse {
    repetead DataCapsule dataCapsules = 1;
}
service DataCapsuleService {
    rpc Send (SendRequest) returns (SendResponse);
    rpc Retrieve (RetrieveRequest) returns (RetrieveResponse);
}
```

We can populate with Data Capsules by calling ”Send” which contains a SendRequest (made up of a DataCapsule) and it does not respond, but we define the response for further compatibility purposes. Also, we can retrieve one or more Data Capsules by calling ”Retrieve” and a timestamp or/and a location. This list can easily be extended, as you can see in the figure above.

4.3 Data collection workflow for GIGEL

The main workflow of a Data Capsule starts when a microservice decides to send data to Cloud. At that moment, a Data Capsule Builder takes the raw data and transforms it into a Data Capsule under the ProtoBuffer message model. If there is no Cloud connectivity, the Data Capsule is saved locally as a FIFO set and is it checked again after some delay if the connection is up. If it is up, gRPC sends the Data Capsule to the closest Access Point and from there on, ROBIN-Cloud takes place.

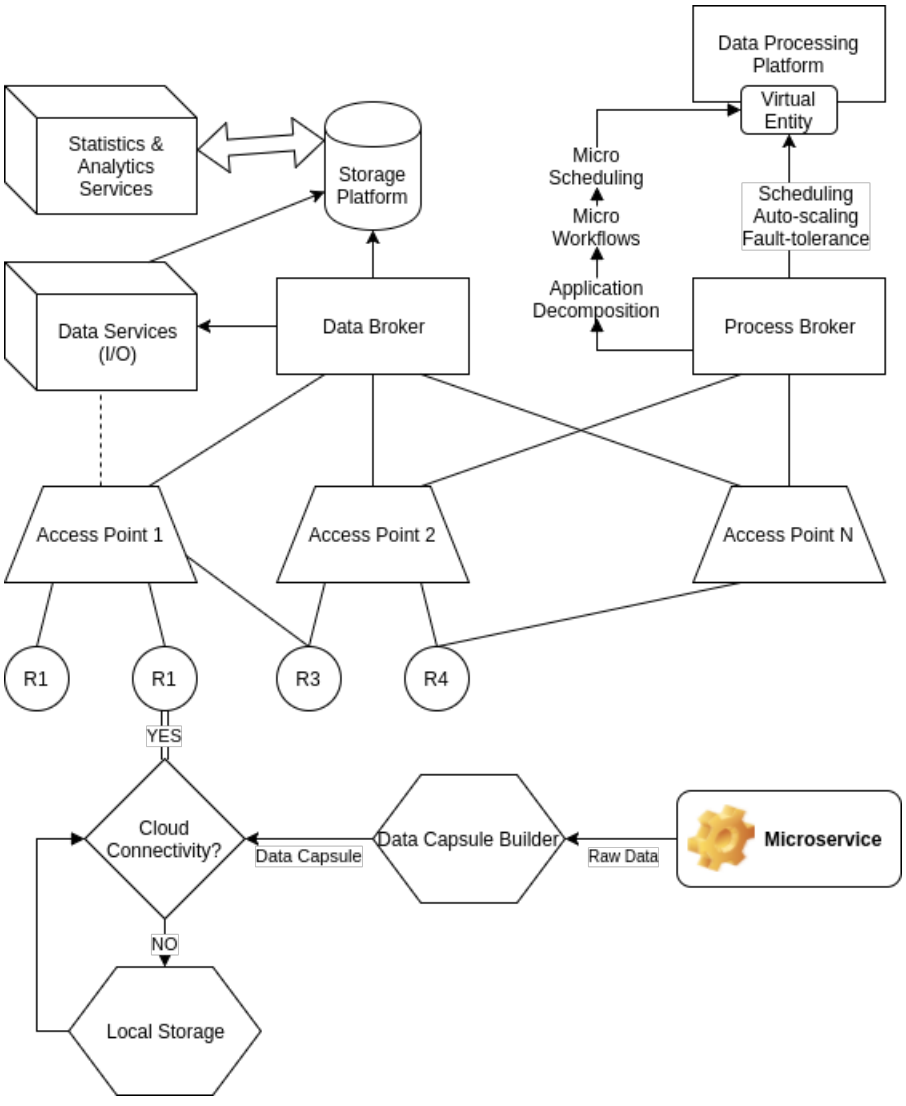

Figure 17: Data collection workflow.

In Cloud most of the processing is especially made for Machine Learning algorithms. The Data Capsule Storage is periodically divided into three sets:

- Training set: used for training the weights of the Neural Networks;
- Validation set: used to tweak the parameters of the training so it has a minimum difference between the accuracy on predicted training examples and predicted validation examples;
- Test set: used for the final accuracy, an unbiased result because all of the tweaks were made against another data sets in different batch sizes or learning rates.

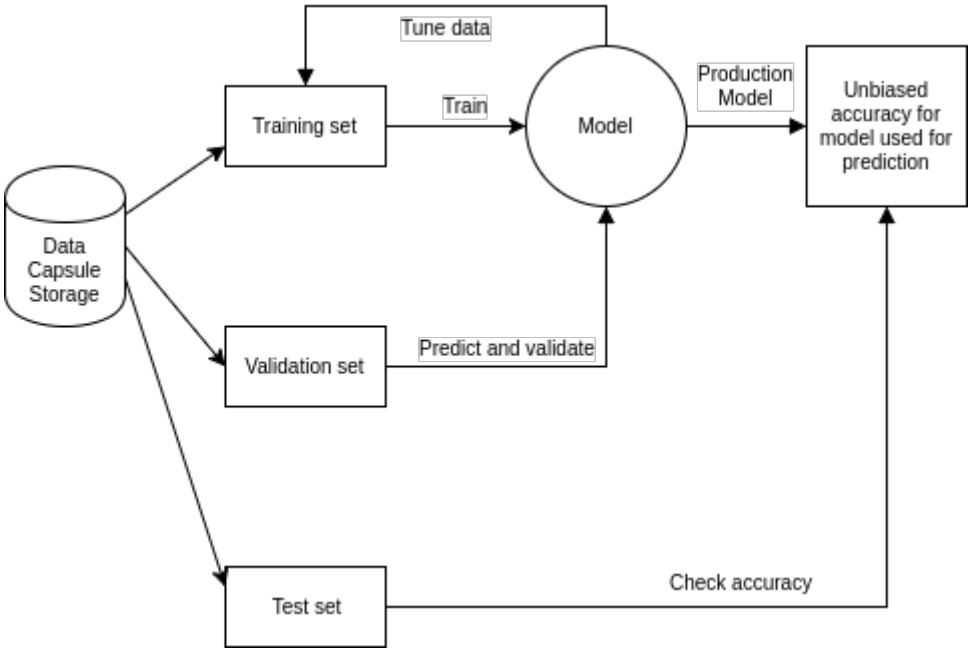

Figure 18: Machine learning platform on Cloud.

5 EXPERIMENTAL RESULTS

5.1 Evaluation metrics

We need to compare different serializers and the latency generated by each of them, both on the serialization and the deserialization phase. For that, we have chosen three technologies, Protocol Buffer, Wire (which is mainly used in the .NET framework) and Jackson (a framework used in most of the Java-based architectures including other framworks as Java Spring etc.).

Also, for the machine learning part, we need to see which activation functions work the best in our neural networks and how many samples or what kind of batches to choose for the purpose of lowering the cost functions and make the predictions better.

5.2 Performance analysis

For the Data Capsule we tried three different technologies to serialize the data between sending it in the network and also deserialize it when it is received: Protocol Buffer, Wire, Jackson.

In 10000 runs, we measured the times (nanoseconds) for each serialization/deserialization operation.

Serializer	Serialization (ns)	Deserialization (ns)
ProtoBuffer	821.23	893.81
Wire	1211.04	1327.75
Jackson	1121.13	1969.44

Table 1: Table with serializers comparison.

Table 1 clearly shows that ProtoBuffer works the best for our architecture. It also can be seen in the figure 19.

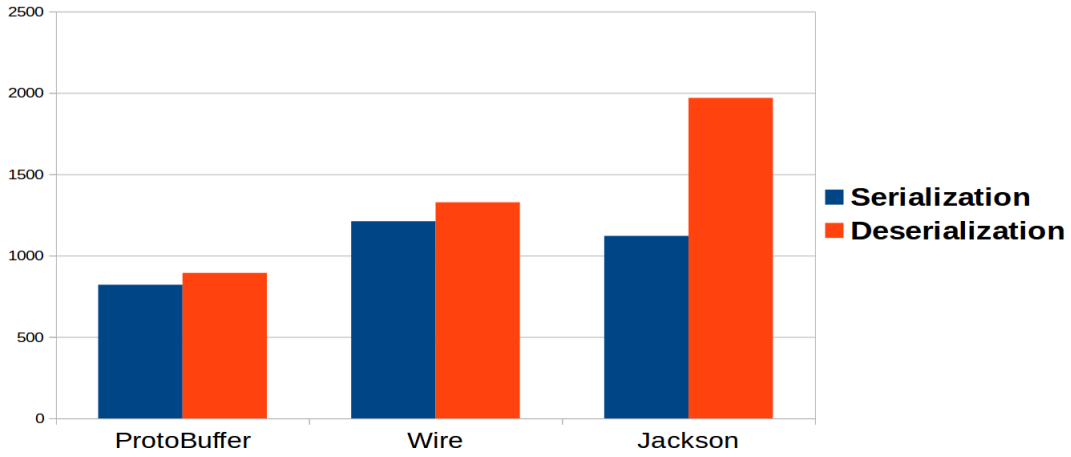

Figure 19: Comparison between serializing technologies.

Talking about the face recognition from a video input, we have a trade-off between analyzing more samples (implying a higher time consumed, but a better accuracy) and analyzing less samples with some latency gains.

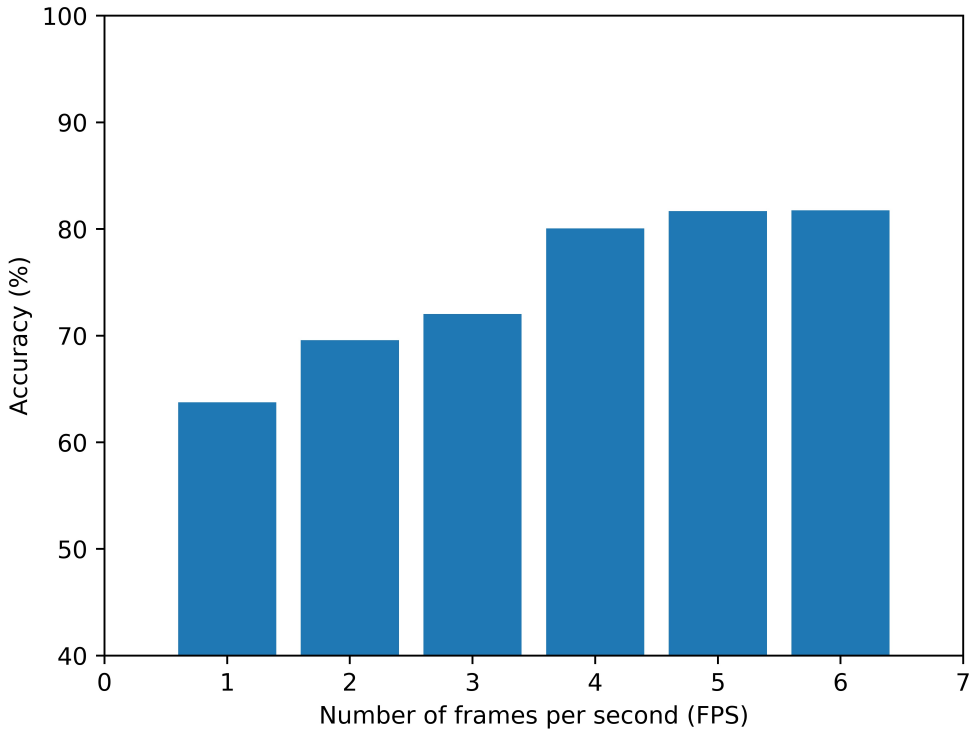

Figure 20: Accuracy of face recognition algorithm.

In Figure 20 the accuracy of the machine learning face recognition algorithm is plotted for 2, 4, 8, 16, 32 and 64 FPS noted as 1, 2, 3, 4, 5, 6. We can see that the accuracy reaches a plateau at 32 FPS so there is no need to put more stress on the CPU to use 64 FPS.

Also, we compared two activation functions for our neural network: Sigmoid and ReLU and we can observe in Figure 21 that ReLU is better for our purpose. We generated the chart from an average of 100 runs in 100 epochs each run for each of the activation functions.

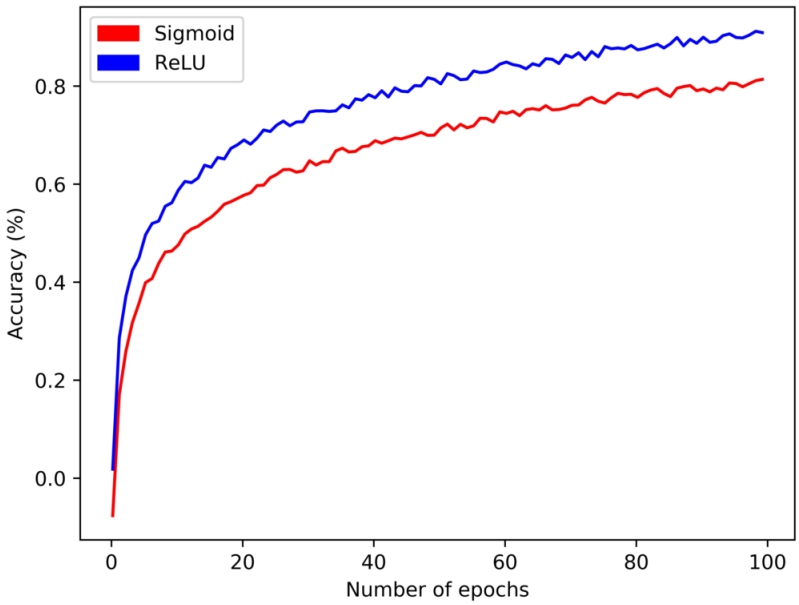

Figure 21: Accuracy of face recognition algorithm.

6 CONCLUSION AND FUTURE WORK

In this thesis we talked about an architecture that is closer to the devices which acquire raw data, we proposed a generic data type to encapsulate heterogeneous data, we created a way to connect to the existing ROBIN-Cloud system and we compared some machine learning strategies for real-time or offline processing in Cloud.

As the experimental results show, we know how to choose some of the parameters used for a better approximation and efficiency regarding the face recognition and voice recognition algorithms.

We also created a valid proof of concept, GIGEL[1], a robot, defined as in chapter 3, which acquires raw data through its sensors (distance, location etc.), serializes it in a Data Capsule and sends it to Fog (which will decide if it can process it or send it upwards to Cloud). Our proof of concept also receives voice commands and can answer us to specific operations, record through a web camera giving us the possibility to use machine learning algorithms on heuristically chosen frames.

As for the future work, there are a number of improvements we can think about now. We only used PCA algorithms for noise reduction and there are other algorithms such as EMPCA[2] or nullifying the noise with a collection of noise records. Also, face recognition can be done on real-time basis but we have to think about a fair trade-off between processing power and battery consumption on every robot. The Data Capsule model can be modified to comprise the current world's needs better, with location getting out of the context or another structure of metadata. Because there is a Message Broker (and a Process Broker on the Fog layer) that handles every capsule coming from an Access Point, there are many scheduling issues that we did not cover in this paper and can be addressed in future work.

[1]Guided Intelligent GEared Legend

[2]Weighted Expectation Maximization Principal Component Analysis

REFERENCES

[1] S. Bailey. Principal Component Analysis with Noisy and/or Missing Data. *pasp*, 124:1015, September 2012.

[2] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

[3] Aleksandar Bojchevski, Yves Matkovic, and Stephan Günnemann. Robust spectral clustering for noisy data: Modeling sparse corruptions improves latent embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 737–746, New York, NY, USA, 2017. ACM.

[4] Thomas W. Dinsmore. *Disruptive Analytics: Charting Your Strategy for Next-Generation Business Analytics*. Apress, Berkely, CA, USA, 1st edition, 2016.

[5] Kai Fan, Junxiong Wang, Xin Wang, Hui Li, and Yintang Yang. A secure and verifiable outsourced access control scheme in fog-cloud computing. *Sensors*, 17(7), 2017.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[8] Lilian Hernandez, Hung Cao, and Monica Wachowicz. Implementing a fog/cloud architecture for supporting stream data management. *CoRR*, abs/1708.00352, 2017.

[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on*

Networked Systems Design and Implementation, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[10] Yun Liu, Qi Wang, and Hai-Qiang Chen. Research on it architecture of heterogeneous big data. *Danjiang Institute of Science and Technology*, 18(2):135–142, Jun 2015.

[11] Pankesh Patel, Muhammad Intizar Ali, and Amit Sheth. On using the intelligent edge for iot analytics. 32:64–69, 09 2017.

[12] Rashed Salem and Asmaa Saad. Fixing rules for data cleaning based on conditional functional dependency. 1, 04 2017.

[13] Palvinder Singh, M-Tech Student, and Anurag Jain. Survey paper on cloud computing. 3:84–89, 04 2014.

[14] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[15] Alvaro Videla and Jason JW Williams. Rabbitmq in action: distributed messaging for everyone. 2012.

[16] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.