Directed Study - Buoys for Shark Recognition

Live Processing of a Distributed Camera Network

Professor:
Kia TEYMOURIAN

Author:
Pierre MOREAU

Abstract:
The shark crisis in the island of La Reunion requires innovative measures to secure sensible surfing and recreation shores. The CRA research center is developing a detection algorithm able to identify harmful sharks on underwater video images. This paper studies the Information System's architecture and implementation to make this project possible.

Please note that this project will be continued, and this document only wraps the part of this project supervised under the directed study with professor Kia Teymourian.

Boston University

Le Centre de Ressources et d'Appui pour la réduction du risque requin

September 15, 2019

Contents

1 **Project Description** **4**
1.1 Motivation . . . 4
1.1.1 Limitations of existing work . . . 5
1.2 Initial targets . . . 5
1.2.1 Computer Science Tasks . . . 5
1.2.2 Engineering the IoT device . . . 6
1.3 Timeline . . . 7
1.4 Funding . . . 7

2 **Technical Requirements Specifications** **8**
2.1 Project presentation . . . 8
2.2 Functional Requirements . . . 8
2.3 Interface Requirements . . . 9
2.3.1 Hardware interfaces . . . 9
2.4 Environment . . . 9
2.4.1 Weather . . . 9
2.4.2 Terrain . . . 10
2.4.3 Connectivity . . . 10
2.5 Technical specs . . . 10
2.5.1 Standards and compatibility requirements . . . 10
2.5.2 Buoy configuration and barrier length . . . 10
2.6 Performance Requirements . . . 10
2.6.1 Speed requirements . . . 10
2.6.2 Reliability requirements . . . 10
2.6.3 Scalability requirements . . . 11

3 **Architecture Research** **12**
3.1 Dumb vs Smart buoys . . . 12
3.2 Network bandwidth . . . 13
3.2.1 Goodput coefficient . . . 13

4 **Development** **15**
4.1 Network technologies . . . 15
4.1.1 Equivalent number of connected cameras . . . 15

4.1.2 Comparison 16
4.1.3 Results 18
4.2 Architecture 18
4.2.1 Kafka 19
4.2.2 Our Kafka-centered design 22
4.2.3 The fatal flaw 25
4.2.4 NiFi 25
4.2.5 Zabbix 28
4.2.6 MXNet 29
4.2.7 System Requirements 29

5 Evaluation **32**
5.1 First implementation tests and measures 32
5.1.1 Test : Video acquisition and compression 35
5.1.2 Test : Network communication speeds 35
5.1.3 Component list and necessary budget 36
5.1.4 Unfortunate turn of events 36
5.2 NiFi implementation and simulations 37
5.2.1 NiFi Implementation 37
5.2.2 NiFi Tests 39
5.2.3 NiFi Test Results 40

6 Future Work **43**

7 Appendices **45**

Bibliography **49**

List of Figures

4.1 Kafka Broker and Partition Architecture. An example with 3 nodes, 2 partitions, 2 consumer groups, and a replication factor of 3 20
4.2 Our overall architecture design 24
4.3 the interface of NiFi's software [5] 26
4.4 the NiFi nodes inner workings [5] 27

5.1 Pertinent selection from "Comparison of 125 Open Spec, Hacker Friendly Single Board Computers", LinuxGizmos.com [12] 33
5.2 Illustration of the test subsection 5.1.1 35
5.3 Illustration of the test subsection 5.1.2 35
5.4 Hardware order list for first tests 36
5.5 NiFi architecture for the tests 37
5.6 NiFi Simulation Test Results 41

7.1 NiFi flow on the buoys 46
7.2 NiFi flow on the server 47
7.3 NiFi alert and log flow on the server 48

List of Tables

3.1 Summary of "dumb" vs "smart" buoys 13

4.1 Wireless communication technologies and their bitrate in Mbps [9, 14, 23, 2] 17
4.2 Wired communication technologies and their bitrate in Mbps [9, 14, 23, 3, 1] 17
4.3 Arbitrary notation of some characteristics consequences ranging from -5 (bad) to 5 (good) 18
4.4 Minimum System Requirements with "dumb" buoys 30
4.6 Minimum System Requirements with "smart" buoys 31

5.1 Min and Max processing times between implementations 40

Chapter 1

Project Description

This chapter is a general introduction and presentation of this project.

Contents

2.1 Project presentation **8**
2.2 Functional Requirements **8**
2.3 Interface Requirements **9**
  2.3.1 Hardware interfaces 9
2.4 Environment **9**
  2.4.1 Weather 9
  2.4.2 Terrain 10
  2.4.3 Connectivity 10
2.5 Technical specs **10**
  2.5.1 Standards and compatibility requirements 10
  2.5.2 Buoy configuration and barrier length 10
2.6 Performance Requirements **10**
  2.6.1 Speed requirements 10
  2.6.2 Reliability requirements 10
  2.6.3 Scalability requirements 11

1.1 Motivation

The island of La Reunion is facing a huge shark attack crisis since 2016. With an average of 2 attacks each year, half of them deadly. The last one in date only happened a week ago, at Saint Leu, where a surfer was killed during a shark attack

[10]. The consequences are not only horrible for the victims, but devastating for this small island living on tourism, having an unemployment rate of 24% [18].

The Centre de Ressources et d'Appui pour la réduction du risque de requin (CRA) [15] is leading a research initiative for underwater shark detection devices based on cameras and Artificial Intelligence. Those cameras should detect sharks in real time and send alerts to the nearby beaches and surf spots.

The machine learning algorithm is being developed by Etienne Meunier, also part of his directed study here at Boston University. I would be in charge of designing and prototyping a device to host underwater cameras, including the complete Information Systems that deals with data acquisition, distributed computing, network communications, and data analytics.

Such a device would not be used exclusively for this purpose but would also be useful for a multitude of underwater marine studies based on image recognition.

1.1.1 Limitations of existing work

Intrusive methods like fishing sharks and safe nets or repulsive systems like magnetic wave emitters are not proving good results [30]. Efforts are shifting towards autonomous shark-finding technology. A sonar systems developed by SharkTec [32] or Xblue [16] are showing promising results, but at too steep of a cost. A deployment of 6 sonars have an estimated cost of about $15 000 [25]. To cover up for a large area, the goal is to implement a cheaper scalable solution using regular cameras instead of high-tech sonars.

1.2 Initial targets

1.2.1 Computer Science Tasks

Being able to integrate on a device:

- ☐ An array of sensors (cameras) and other input probes
  - Data acquisition
  - Video stream processing (OpenCV, Spark, Kafka..)
  - Centralized or distributed computing
- ☐ Computing resources for the detection algorithm
  - Develop information system to deal with the data pre-processing and input of the algorithm
  - Distributed/Mobile computing

  - Information systems integration
- ☐ Network communications [7]
  - Network design, architecture, components
  - Mesh networking, cellular networking, IP and RF protocols
  - Data compression and bandwidth, Network security, network management [26]

Design and develop the overall Information Systems:

- ☐ Design and implement the Information System's infrastructure
  - Centralized systems management
  - Database management (Centralized/distributed Database management systems, Relational and non-relational models) [7]
- ☐ Design the network infrastructure to handle all input data from a network of probes, and the communication strategies
- ☐ Implement the data analytics software
  - Design the Data Analytics dashboards (Python, React, Tableau Software, PyTorch..)
  - Implement the alerting events and mechanisms (Python, Flink, Kafka ..)

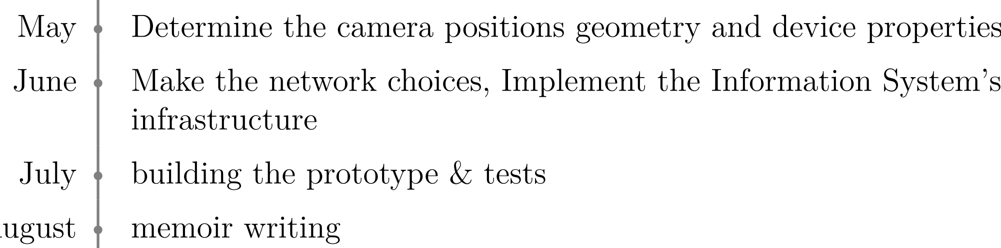
1.2.2 Engineering the IoT device

Design and Prototype the device:

- ☐ Make the right hardware choices
  - cameras, computer, communication devices and technologies [26]
- ☐ Design the device
  - Buoy/underwater net/anchor [7]
- ☐ Implement a reliable power source for autonomous use
  - Solar panel / Wave energy
- ☐ Test device in Charles River

1.3 Timeline

This project should be done over the course of 2 directed studies, in Summer 1 and Summer 2 2019.



1.4 Funding

Such a project requires a lot of hardware parts and even cloud resources, for which the CRA research center will be validating and paying the budget. I would, however, need to have access to the tinkerLab and maker spaces available at BU.

Chapter 2

Technical Requirements Specifications

The purpose of this part is to specify the system we will design in complete and technical details. It should identify the set of functions the system must provides and constraints and performances it must conform to.

Contents

3.1 Dumb vs Smart buoys **12**
3.2 Network bandwidth **13**
3.2.1 Goodput coefficient 13

2.1 Project presentation

As it was presented in the introduction of this document, the project is composed of underwater cameras connected to an artificial intelligence algorithm that analyses the videos in real time and send alerts when it detects sharks.

2.2 Functional Requirements

1. The system records underwater images of the shores
2. The videos are analysed in real time to detect sharks
3. The system saves some of the footage for the archives
4. The system should be easy to deploy and setup

5. When deployed, the system should be able to work in the day for 8h straight. It is not needed for the system to be totally autonomous several days in a row
6. Camera videos should be stabilized
7. The cost of the buoys should be kept low, so that the system is cost effectively scalable.

2.3 Interface Requirements

2.3.1 Hardware interfaces

1. The cameras should be encapsulated in watertight compartments, attached or inside a buoy floating at the surface of the see. Electronics should be well protected from the elements, but also easily accessible to service them.
2. The camera windows should be easily cleanable, and batteries easily rechargeable if any.
3. All hardware should be rugged to withstand the sea weather conditions and vigorous handling.
4. As the buoys will likely be working in chain, they should be easily swapped should any buoy fail, to allow fast maintenance and lower downtime.
5. The elements present in the water should be of high visibility.

2.4 Environment

The project is planned to be used at the island of La Reunion, to secure beach areas from sharks.

2.4.1 Weather

The island benefits from a tropical climate softened by the breezes of the Indian Ocean. It is always summer in La Reunion with temperatures ranging from 20°C to well over 30°C by the coast. The water temperature in the lagoon varies from 20°C to 25°C. Rain showers come in short, but heavy patches [13].

The sunrise goes from 5:30 AM in december to 7 AM in winter. It sets between 7 PM and 5:50 PM, giving around 12h of daylight. Clouds tend to form in the mountains during the day.

However, the sea conditions will likely have lots of currents and waves, as it will be used around surfing competitions.

2.4.2 Terrain

The island is a volcanic island, and the sea shores are made mostly of black and white volcanic sand. The depths at which the devices will operate are shallower than 15m, mostly around 5 to 10m. At those depth, it may also be sufficient to keep the cameras at sea level.

2.4.3 Connectivity

The island has a good mobile data connectivity with bands ranging up to 4G.

2.5 Technical specs

2.5.1 Standards and compatibility requirements

No standards or technologies to be compatible with have been identify up to the moment of publication of this report [1].

2.5.2 Buoy configuration and barrier length

The virtual barrier sould be able to scale up and down between 50m and 1km in length.

2.6 Performance Requirements

2.6.1 Speed requirements

1. The system should allow a **1 min advance** from the moment it notifies of the presence of a shark, and the moment the shark could approach.

2.6.2 Reliability requirements

1. The system should be highly reliable
   (a) If a shark has been detected, the notification should guaranteed delivered
   (b) The system should keep working even though parts of it degrades
   (c) If parts of the system fails, it should be detected and the users should be notified

[1] *This usually concerns enterprise databases and tools to interface with.*

2.6.3 Scalability requirements

1. The network, processing servers, and storage should be easily and cheaply scalable up to being able to handle the real-time treatment between 50 and 400 cameras [2] .

[2]Based on the fact that underwater cameras have a sight between 5 to 10 meters, and we can expect each buoy to include 3 or 4 cameras at different orientations, buoys should be placed every 10 meters. For a barrier of 500m, this makes a chain of 200 cameras. *Some research is still needed here to validate those results.*

Chapter 3

Architecture Research

This chapter focuses on evaluating the best technologies to use for this project, and how their implementations impacts performance and usability.

Contents

4.1 Network technologies **15**
4.1.1 Equivalent number of connected cameras 15
4.1.2 Comparison 16
4.1.3 Results 18
4.2 Architecture **18**
4.2.1 Kafka 19
4.2.2 Our Kafka-centered design 22
4.2.3 The fatal flaw 25
4.2.4 NiFi 25
4.2.5 Zabbix 28
4.2.6 MXNet 29
4.2.7 System Requirements 29

3.1 Dumb vs Smart buoys

We are presented with the choice of making either **"dumb" buoys**, that uploads the images to be analysed on the network. Those buoys are cheap, but require expensive and complex infrastructure.

On the other hand, we can use **"smart" buoys**, that runs the AI algorithm directly in the buoy, so that they only have to send small amounts of data (alerts, monitoring data, pings, some images in case of a positive detection). The buoys needs

to be powerful enough to run the AI inference on all its cameras, so that makes it a lot more expensive. However, the overall infrastructure complexity and costs are a lot better. Moreover, we can expect the price of the computer to be relatively a small portion of the buoys total costs, as long as it's power consumption don't become hard to handle.

	"dumb" buoys	**"smart" buoys**
Description	inference done in a heavy processing cluster on the network	inference done localy in each buoy for its own cameras
Network Bandwidth per camera	enormous : 2 fps	tiny : pings, rarely an image
Scalability	hard but cheap	easy but expensive
System's Useability	complexe	easy

Table 3.1: Summary of "dumb" vs "smart" buoys

3.2 Network bandwidth

The maximum rate that can be sustained on a link are limited by the Shannon-Hartley channel capacity for these communication systems, which is dependent on the bandwidth in hertz and the noise on the channel. Each networking communication technologies has it's limitations regarding the amount of information that can be transferred on the network. This first section surveys the existing networking technologies to find the best alternatives for this project.

3.2.1 Goodput coefficient

The useful information throughput that goes through a channel is always lower than the physical bitrate. Protocols, encryption, noise, medium instability adds up overheads and re-transmissions, which takes up bandwidth usage. For instance, the usual TCP protocol used in internet communications requires a 3-way handshake at each transaction, which adds up overhead. When packets are lost or corrupted, it also reduces the useful bandwidth, even more if they need to be sent again.

For instance, the Wifi technologies using the 5Ghz bands only have a short range of sight. This basically means that each buoy should be having a Wifi repeater in order to bounce the signal up and down the chain. However when acting as a repeater, a Wifi equipment divides its bitrate by 2 since the information is travelling twice on its channel, from the source node and then to the next buoy.

For estimation purposes, we define the goodput coefficient to be a percentage of the physical bitrate approximating the **Effective Bitrate** that we could leverage. This coefficient is mostly highly dependent on the physical conditions, distances, and

protocols used, so we tried our best to replicate the industry's estimations to guide our decision.

Chapter 4

Development

This chapter focuses on developping scenarios and leveraging existing technologies.

Contents

5.1 First implementation tests and measures **32**
5.1.1 Test : Video acquisition and compression 35
5.1.2 Test : Network communication speeds 35
5.1.3 Component list and necessary budget 36
5.1.4 Unfortunate turn of events 36
5.2 NiFi implementation and simulations **37**
5.2.1 NiFi Implementation 37
5.2.2 NiFi Tests 39
5.2.3 NiFi Test Results 40

4.1 Network technologies

4.1.1 Equivalent number of connected cameras

In the case of "smart" buoys, all the networking technologies are good enough to support more than 200 cameras.

In the case of "dumb" buoys, the images should not be processed on the buoys, but rather on a powerful computing cluster in the cloud or on shore. We can estimate the number of images and equivalent cameras that the network can handle.

The data we will be transferring on the connections are still pictures from HD cameras.

Full HD video images have $1920 * 1080 = 2Mpx$. A 100% JPEG compression with 24 bits per pixel gives a file size of 3.38 Mb. A 90% compression almost divides the file size by 2 giving a still of 1.69 Mb [17].

Using this first approximation, we can draw a gross number of images that can be sent across the network. Knowing that each camera is supposed to film at 2 frames per seconds, we also deduce the equivalent maximum number of connected cameras.

4.1.2 Comparison

Even though our numbers are not fully reliable, it's important to make estimates at some point so that we can draw conclusions and move forward.

The tables 4.1 and 4.2 summarize our studies across the different mediums, either with wireless or wired technologies. All bitrates are in Mbps.

Considering that we plan to deploy around 200 cameras, all cellular networks have an upload rate too small, so it filters out everything except Wireless 802.11ad, Ethernet, USB and thunderbolt. Now the maximum cable length of the serial communications (USB and Thunderbolt) render those technologies useless in our scenario.

In order to guide our choice between Wireless Wifi or Ethernet, we further developped the characteristics and consequences of using those technologies in the table 4.3.

Either of those technologies are all vulnerable to cutting (waves for wifi, cable break for ethernet), and have a Single Point Of Failure (SPOF) should a buoy crash, as it stops transmitting all the data from buoys down the line. This vulnerability should be monitored all the time so that users are alerted should it occur.

10Gb and 40Gb Ethernet connections assures reliability and scalability. Those connections can be either optical or copper based. The fiber cables are really fragile and needs to be handled with great care. The copper based version, with a max length of 100m (vs 40Km for the optical fiber), as a major cost advantage over the optical fiber option. The Category 6a Ethernet cable and equipment allows 10 Gbps transfer rates over 100m cabling. This is a very cost effective solution [20, 1].

Having autonomous Wifi buoys, they need solar panels or betteries to function properly. The convenience of those independant devices also comes at the dependence of having to recharge them at night, whereas the wired buoys can just supply power in the cables.

Technology	download rate	goodput coefficient	Effective Bitrate	Range	Approx max frames per sec.	Equiv. nb of cameras (2fps)
EDGE (type 2 MS)	1.894		1.894	Long	1.1	**0**
LTE (4×4 MIMO)	326		326	Long	192.8	**96**
5G	900		900	Long	532.5	**266**
Wireless 802.11b	11	60%	6.6	100m	3.9	**1**
Wireless 802.11g	54	60%	32	100m	19.1	**9**
Wireless 802.11n	600	30%	180	30m	106.5	**53**
Wireless 802.11ac	1 300	30%	390	30m	230.7	**115**
Wireless 802.11ad	7 000	30%	2100	30m	1242	**621**
Bluetooth 3.0	25		25	60m	14.7	**7**
Bluetooth 4.0	25		25	60m	14.7	**7**
Bluetooth 5.0	50		50	240m	29.5	**14**

Table 4.1: Wireless communication technologies and their bitrate in Mbps [9, 14, 23, 2]

Technology	physical bitrate	goodput coefficient	Effective Bitrate	Max Length	Approx max frames per sec.	Equiv. nb of cameras (2fps)
Ethernet	10	80%	8	100m	4.7	**2**
Fast Ethernet	100	80%	80	100m	47.3	**23**
Gigabit Ethernet	1 000	80%	800	100m	473.3	**236**
10Gb Ethernet	10 000	80%	8000	100m	4733.7	**2366**
40Gb Ethernet	40 000	80%	32000	100m	18934.9	**9467**
100Gb Ethernet	100 000	80%	80000	100m	47337.2	**23668**
USB 2.0	480	80%	384	5m	227.2	**113**
USB 3.0	5 000	80%	4000	3m	2366.8	**1183**
USB 3.1	10 000	80%	8000	3m	4733.7	**2366**
USB 3.2	20 000	80%	16000	3m	9467.4	**4733**
Thunderbolt 2	20 000		20000	3m	11834.3	**5917**
Thunderbolt 3	40 000		40000	2m	23668.6	**11834**

Table 4.2: Wired communication technologies and their bitrate in Mbps [9, 14, 23, 3, 1]

Characteristic	10Gb Ethernet	40Gb Ethernet	100Gb Ethernet	Wifi 802.11ad
Reliable	5	5	5	-2
Vulnerable to cutting	-3	-3	-5	-1
Single Point Of Failure (SPOF)	-5	-5	-5	-5
Scalable	4	5	5	1
Availability of equipment	5	1	1	4
Praticity	0	0	-1	5
Needs batteries or solar panels	0	0	0	-3
Cost	4	2	-3	3
TOTAL	**5**	**0**	**-8**	**4**

Table 4.3: Arbitrary notation of some characteristics consequences ranging from -5 (bad) to 5 (good)

4.1.3 Results

"smart" buoys : All presented networking technology is supposedly good enough to handle 200 cameras analyzed by smart buoys.

"dumb" buoys : Either the Wireless 802.11ad and 10Gb Ethernet are both networking technologies suitable for our use case. In a subsequent test phase, we will be implementing both to run some tests and measures in order to further validate the viability of both those implementations.

Certainly improvements could be made on the compression phase to allow better results, however at the moment 10Gb Ethernet on Cat6a cables is the most cost effective applicable and reliable solution that has margin for error and further scaling.

4.2 Architecture

Considering 4 cameras per buoy every 10m for 500m yields 200 cameras. If we take 200 cameras with that 90% JPEG compression, it generates almost 680 Mbps every seconds, which is 2.3 Terabits of images every hour. Obviously that volume should be further compressed when considering storage, once the images have been processed. For instance, by skimming the images to be saved as not all images are useful in the archives.

However, the processing unit should be able to handle that processing rate and hold sufficient space in RAM to load those images to process. And the number are even higher regarding the processing of the image. When they are loaded in RAM, each frame takes up to 1.33 Gbits in space.

Those numbers requires a wicked fast, real-time processing architecture. However since the number of cameras is likely to be a variable, the architecture needs to be easily scalable to accommodate the costs with the project's size.

4.2.1 Kafka

For reliable and efficient real time processing, there is a need for a cost-effective, scalable and fault tolerant distributed system. Kim and Jeong [21] detailed a distributed architecture based on the open source Apache Kafka to process video streams in real time : "Although the GPU is well suited for high-speed processing of images, it still has limited memory capacity. Using Kafka to distributed environment allows overcoming of the memory capacity that cannot be accommodated by one node. In particular, since image data can be stored in the file system, it is advantageous to handle large-scale images without data loss." [21].

If you're not familiar with Kafka, I greatly recommend you to read the hackernoon post from S. Kovlovski [22] that details it thoroughly. I will however present its concepts as well as how they are applied and implemented in our case.

1. the buoys (**`producers`**) publishes image frames (**`messages`**) to a node (**`broker`**) in the Kafka cluster. Those messages are all stored in the same Kafka collection (**`topic`**).
2. The Kafka cluster consists of several nodes (`brokers`). The big input topic is divided into smaller **`partitions`** that are assigned to the brokers for load balancing and replication *(see section 4.2.1)*.
3. Processing nodes subscribe to the Kafka brokers using the stream api, and pull data to process as capacity is available. The data will always be pulled in the order it came in, and Kafka will assure a "at least once" treatment for the image.
4. The processing nodes then publishes a new message that encompasses : the image metadata, the result of the labelling application, and the file-path of the archive. Those new messages are published in the same Kafka cluster, but in another topic.
5. The output topic is connected to a relational database for long term storage and analytic.
6. The messages in this output topic can then be subscribed to from a consumer like Flink, which will process the result of the detection and trigger notifications if needed.
7. The processed Kafka messages in the input topic will eventually die after being treated to free some space.

High Availability vs Performance : Partitions, and Replication factors

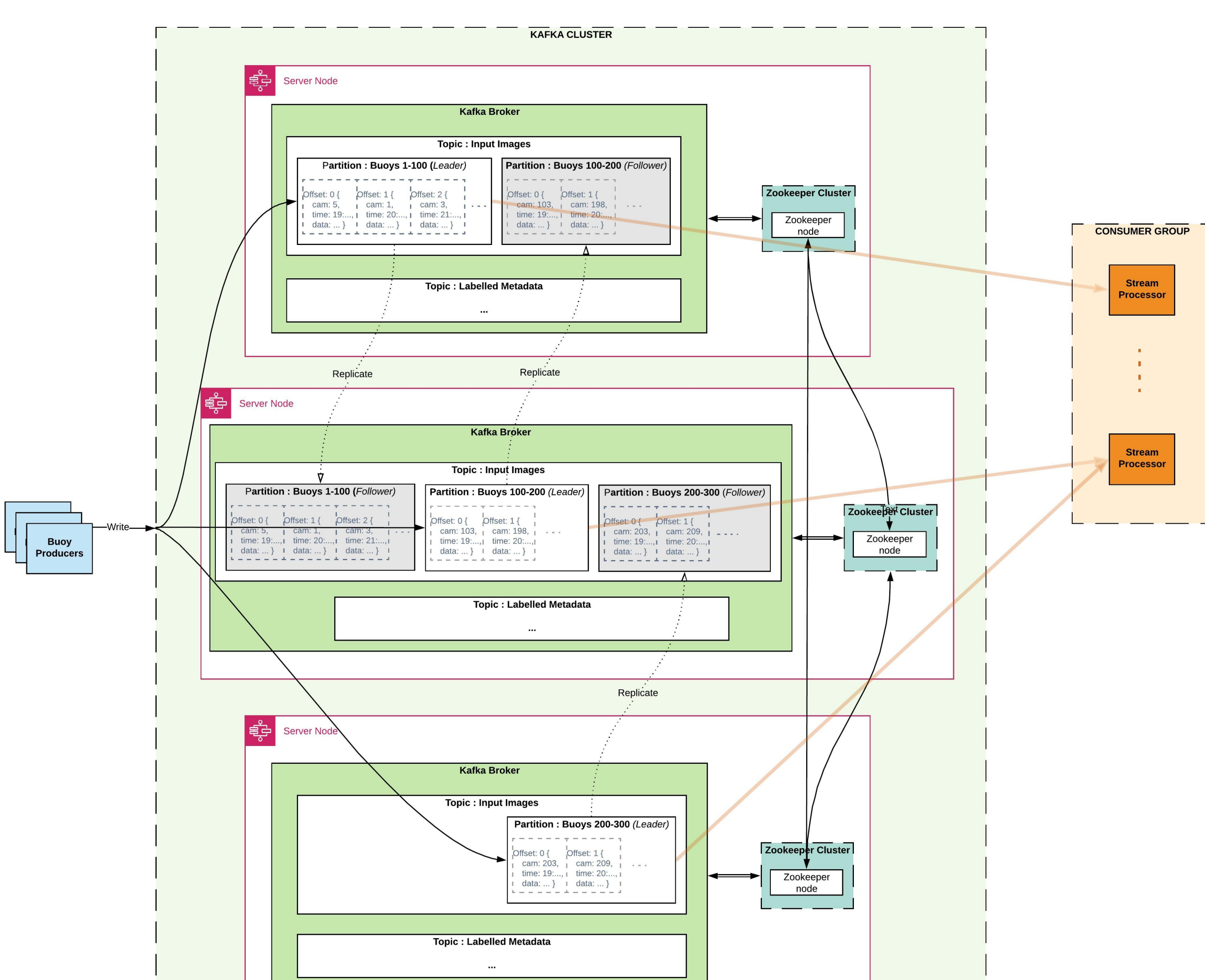


Figure 4.1: Kafka Broker and Partition Architecture. An example with 3 nodes, 2 partitions, 2 consumer groups, and a replication factor of 3

Kafka is designed with failure in mind. At some point, a system will fail, and Kafka's distributed architecture allows for different levels of resilience. It is also designed to be easily scalable by parallelizing it's architecture. The key concepts here are **`partitions`** and **`replications`**.

Partitions for performance

Kafka divides big topics into `partitions`. This allows it to distribute the loads more efficiently. A message is assigned to a partition via a record key if present, or a round-robin by default. We'll likely be using the round-robin assignment strategy, but for the clarity of the argument, we'll represent the partition via the buoy keys : {buoys 1 to 100} and {buoys 100 to 200}. Multiple partitions scales a topic across servers to parallelize writes, and consumers consume messages in parallel up to the number of partitions. Each partition can only be assigned to 1 consumer in a consumer group, so that the messages don't get processed twice by 2 consumers. This means that the partition number is the maximum unit of parallelizing. Multiple partitions can be assigned to a consumer however. Furthermore, in order to have real-time processing, the consumer needs to be able to process all it's assigned partitions incoming data as fast as they are produced. If not, then it's lagging and messages are waiting to be catch-up, which requires more processing power.

So say I have a total of $TMbps$ data to be processed each second, and each consumers can only handle $\frac{T}{C}Mbps$, then I need at least C consumers, and if the data is partitionned equally, at least $P = C$ partitions. However, in the real world, we can't expect each consumer to be equally capable, neither the data flow to be constant, neither the partitions to be totally equal. This is why we need a thinner granularity in the partitions sizes so that they can be assigned in a more balanced way to the consumers, according to their capacity. However, partitions also means more replication latency, rebalances, and open server files. By a conservative estimate, one partition on a single topic can deliver 80 Mbps [11]. This means that our 200 cameras data rate should require at least $\frac{680Mbps}{80Mbps} \approx 9$ partitions. To make those partitions as equally as possible, and to assign them more efficiently, good practices [24] says we should use the round-robin assignment policy.

Replications for High Availability

First, we explained how Kafka keeps the messages written in cache until they are consumed. This guarantees that once a frame is published into Kafka, it is persisted until consumed at some point even if a processing node or application should fail [4].

To account for the failure of a Kafka node however, we need replications so that the data is backed up in another node. In the event of a Kafka node crash, Kafka will reconfigure itself with the remaining nodes. This needs a replication factor ≥ 2 so that a broker with the replicated standby partition (follower) can become leader straight away and take on the stream.

In our case, we shouldn't worry much about the split-brain scenario, that mainly occurs in a cluster spread across different availability zones. A deployment architecture that can tolerate the failure of F machines, should count on deploying

$2 * F + 1$ machines. For such, it needs to have at least 3 nodes to account for the event of the failure of 1.

Replication is great, but we need to ask the right questions for our use case. Considering that messages are consumed in real time, even without replication, if a Kafka broker fails, the system readjusts automatically by redirecting the input messages to the remaining nodes, and messages still continue to get processed in real time. So as long as, in the unlikely event of a broker failure, we can afford a few seconds of images lost (while the network reconfigures), so we shouldn't focus much on solving this problem. As more replication drastically increases network, processing, and disk loads on the Kafka nodes, we shouldn't have a replication factor greater than 2 in our use case, but mainly lots of partitions.

4.2.2 Our Kafka-centered design

The overall following breakdown is illustrated in Figure 4.2. The inspirations and details for this design comes from Data Works Summit conferences [33, 34, 31, 28] :

1. The buoys convert all those frames into serialized objects [1], that encompasses the pixel data and the camera localization and metadata
2. The camera Stream Channel in the buoy publishes those events, and sends them to the Kafka broker in the Input Topic, using the KafkaProducer client
3. The input message is assigned to a partition using the round-robin allocation strategy
4. The kafka broker acts as a buffer queue to store the data while it's waiting to be processed. Kafka stores that data in the file system, which improves durability and overall performance when loads or availability varies. It also garantees the order of the messages in a single partition for a given topic. This also guarantees fault tolerance since Kafka replicates the saved messages to the broker [8].
5. The processing nodes pulls the message in order from the Kafka broker, and reconstructs the frame matrix to feed it to the detection application. This yields optimal performance for large-scale data by pulling only the messages in its processing capacity.

[1] As a study [29] by Sidkar, Teymourian and Jermaine suggests, there are better format than JSON to serialize and compress the data. The JAVA KRYO object for instance should be 2 times lighter than the same JSON object. We'll surely use this format later, but the JSON format is far easier and explicit to use at first in order to do our first implementation tests.

6. Once the labelling is done, the application decides if it should send it for long term storage on the archive database. If so, it compresses the image, sends it to the database filesystem, and adds the filepath to the image metadata.
7. The computing node then publishes a new message in a second topic : "Labelled Metadata", of the original message stripped from the frame data, and enriched with the labellization outcomes and filepath.
8. The output topic is connected to a relational database for long term storage and analytic.
9. The messages in this output topic can then be subscribed to from a consumer like Flink, which will process the result of the detection and trigger notifications if needed.
10. The processed Kafka messages in the input topic will eventually die after being treated to free some space.

Note : Apparently, as Kafka is primarily designed for text messages of small sizes, the configuration will need to be changed to accept larger JSON messages (1.7 Mb). Those parameters in the Producer Config are respectively ***`batch.size, max.request.size, compression.type`*** *[8]*

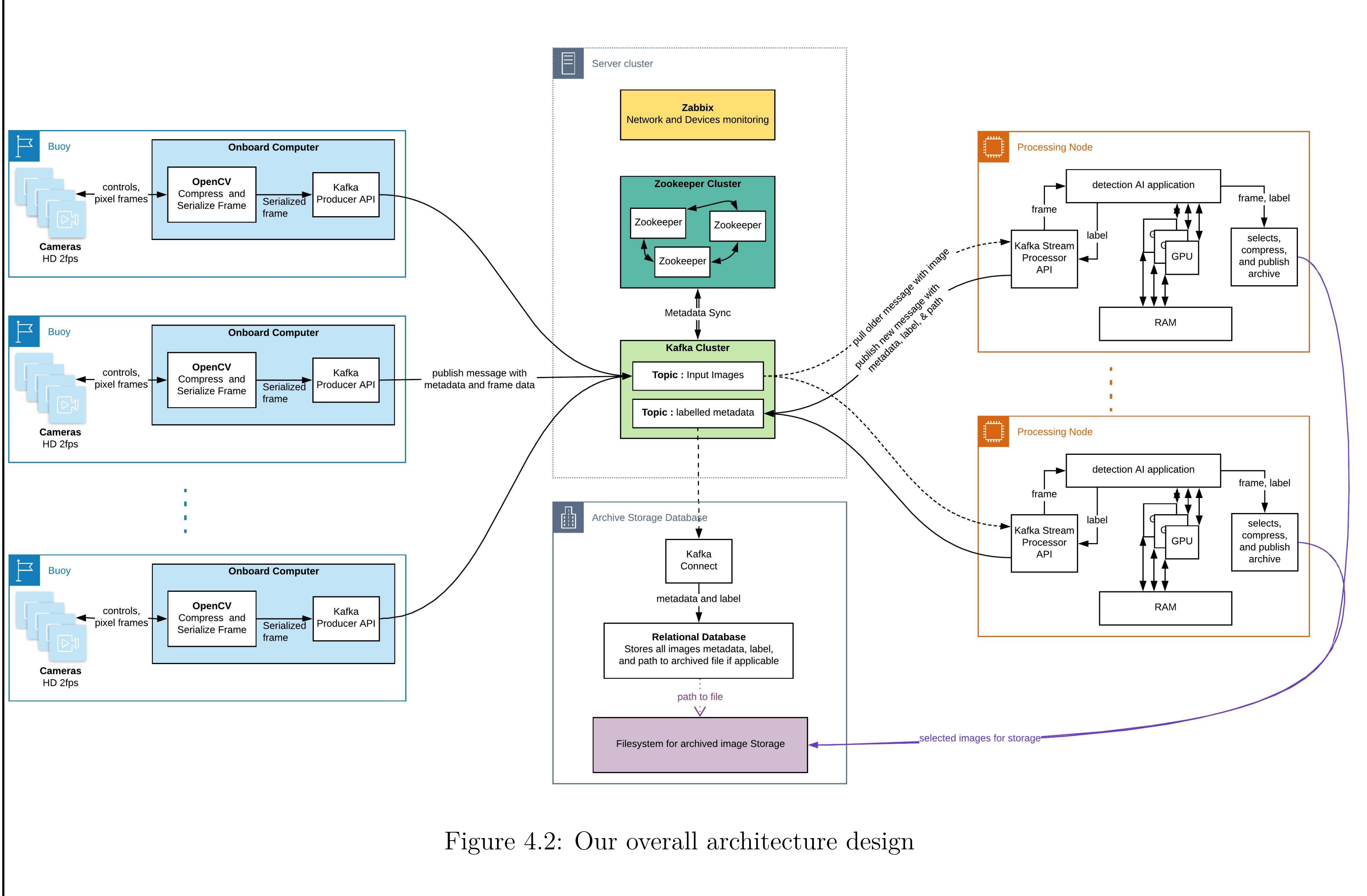


Figure 4.2: Our overall architecture design

4.2.3 The fatal flaw

The Kafka streaming mechanisme is designed around accumulating data, waiting to be processed. Its use case is mainly around variable flowrates, where bursts of input data should be hold onto while the systems scales up and adjusts and processes the data. **No data is ever loss with Kafka, but lags can appear**. As such is is mainly oriented to cloud-like infrastructures with data persistency and close-to-real-time processing. This could be our best solution in the situation where the computing would be done in the cloud, with elastic processing availability.

However, I think **we should rather be concerned with the speed of the data. In the event of accumulation of untreated images, we rather skim a few images out to reduce the load, but keep on processing images without accumulating delay**, as long as we keep all traceability. There is actually a recent framework designed around that, also Open Source by the Apache foundation, it's called **NiFi** [31].

4.2.4 NiFi

NiFi is also an open source data flow tool from the Apache Fundation, with great performance, horizontally scalable and pluggable architecture. However, NiFi doesn't replicate data like Kafka. If a node goes down, the flow is redirected to another node, but the queued data in the failed node will have to wait until the node comes back up. It is however able to configure back-pressure threshold, prioritized queueing, and data traceability. Its key functionalities are [5]:

Guaranteed Delivery : The data is persistently stored. And as long as the system is online (or repaired back online), all data will eventually reach it's delivery.

Back Pressure with specific Quality of Service : Where data must be processed and delivered within seconds to be of any value, it's possible to configure specific latency, throughput, loss tolerance, back pressure and pressure release mechanisms.

Traceability : Easy monitoring of flows, queues, and forensics. Everything is logged.

Modularity : The web-based graph interface allows easy supervision of the flows and routings. Processes and flows can be easily modified, tested, and deployed at any points or any scale of the pipeline.

Edge intelligence : NiFi handles the flow from the first mile of data ingesting in the buoys, to the last miles of shark notifications via instant messaging. By managing all that flow with one tool and combining inputs from different

sources allows you to intelligently and dynamically transfer data. We can leverage the bi-directional communications to the buoys to make them respond to the changing environmental conditions in real-time.

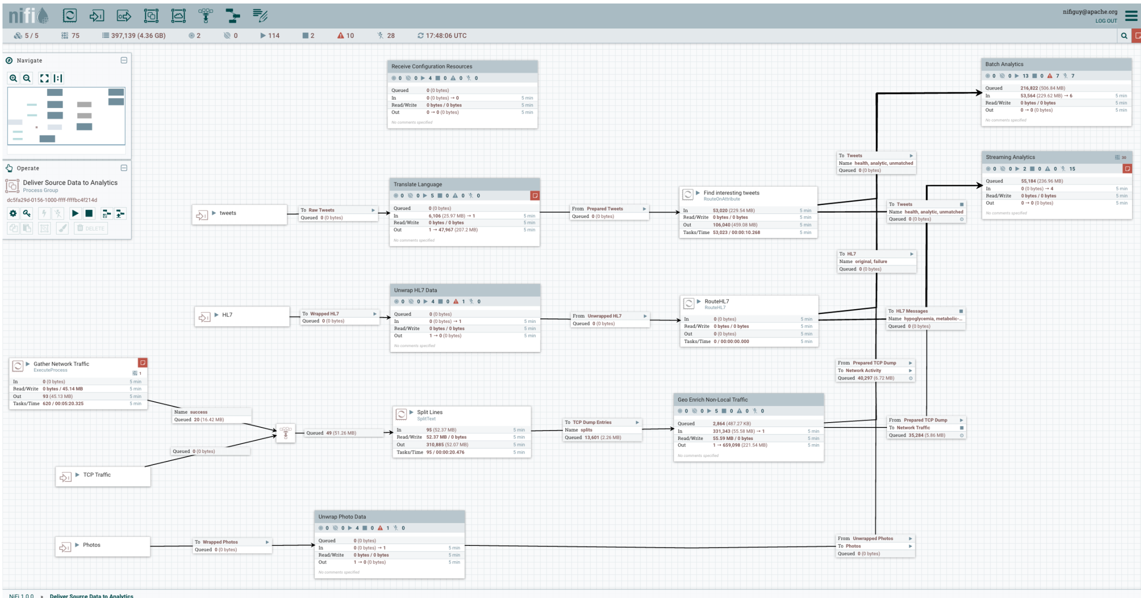


Figure 4.3: the interface of NiFi's software [5]

NiFi is mainly a dataflow tool, and as such has many other great other functionalities. As it's implementation is used to create a fault-tolerent production pipeline. It's simple web-based graph interface and wide coverage makes it easy to modify, test, update and deploy changes at any points and any scale of the pipeline. That flexibility allows the following use cases :

- Create new input connections with input data such as GPS and weather
- Push data to databases or storage
- Connect instant messaging apps like emails or slack to send and get notifications
- Run SQL queries on the network
- No need for python management scripts, cron jobs, automate tools and update mechanisms. Everything can be handled through miNiFi and NiFi
- Flows can be versionned
- Make changes live in the models, or in the network

- Run experiments live on some devices, and deploy model upgrades
- Update and deploy image processing flows or the Machine Learning model
- Scale up and down the network
- **Should the buoys become powerful enough to run the model, it is easy to move some parts of the model to test buoys, or even migrate all the model into the buoys.**

All that abstraction is great, but it also means that it's easy to create congestion and bad routes that would clutter the underlying hidden physical layers with unnecessary data flows and processes. It's important to understand how the data is managed on the network by the tool to use it correctly.

NiFi's Repositories

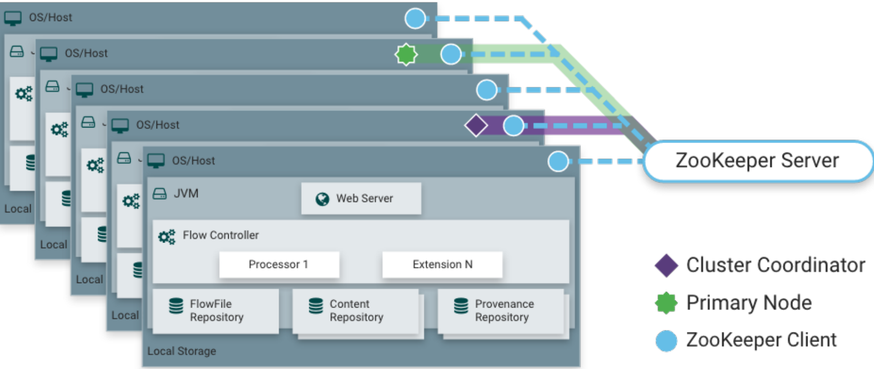


Figure 4.4: the NiFi nodes inner workings [5]

NiFi is a high-level tool, and most of it's implementation is hidden to the user. However if we want to design a performing system, we need take an in-depth look into it's implementation and design decisions.

Flow-based design. Based around FlowFiles. The Apache NiFi documentation describes a FlowFile as : *"A FlowFile is a data record, which consists of a pointer to its content (payload) and attributes to support the content, that is associated with one or more provenance events"* [6].

The payload of the message is the content and its attributes. In our case, it might be the pictures or sensor's data. All the FlowFile's attribute are stored in key-value pairs, and provenance records the transformations and life events of the FlowFiles.

This design is used to optimize the storage, traffic, and usage of those different parts. They are each assigned to a dedicated repository in the Host's file system.

FlowFile Repository : Metadata for all the current FlowFiles in the flow. FlowFiles being processed are loaded in a hash map in the JVM memory. They are then stored on disk in a Write-Ahead log, which is immutable.

Content Repository : Content for current and past FlowFiles. Usually the largest repository, especially when dealing big payloads such as images. Content written is immutable. The motivation is to strip it from the metadata so that it doesn't have to be loaded in memory each time the metadata is processed in the network, and that multiple FlowFiles can share the same pointer to the common content. As such it is good practice to extract key information from the content into the FlowFile's attributes that would be useful for other processor to use, so that it doesn't need to read through the content.

Provenance Repository : History of FlowFiles. New provenance events are created each time an event occurs for a FlowFile, which makes them snapshots of that FlowFile. Those are mainly used for auditing, monitoring, and debuging the flows.

Immutability : The immutability of the Content and Metadata means that the data can't be updated. It has to be read, copied and transformed, and then update the FlowFile's pointer to the new content. It makes the content storage acts like an "immutable versioned content store" [6]. This takes advantage of OS caching to avoids large complex graph processing, and improves replay capability, random read/write performance hits, and reasoning. Immutability also garantees integrity in the events of system failures. The metadata or content are loaded in a hash map of the JVM memory to be processed, and then written to disk in the repositories. With the immutability paradigm, only periodic checkpoints written on disk are enough to restore the node's state in the event of failure. The corrupted data in the JVM's will gets cleaned during the automatic garbage collection.

More in-depth details are given in NiFi's advanced documentation [6].

4.2.5 Zabbix

With all the connected buoys, cameras, processing nodes, Kafka broker etc... we need a centralized tool to monitor everything. Zabbix is an open-source tool designed to monitor the network architecture, as well as all connected devices. It can automatically add devices and monitor their performance. The web interfaces allows the system administrator to configure alerts, clusters, and reports. The Zabbix server should better be installed on a server node in the server cluster, so that it is in the network itself. Here are some points that we plan to achieve with Zabbix :

- Scan the topology of the network, it's changes
- Monitor the Network bandwidth, performance drops, and packets errors
- Scan the devices present on the network, and if any comes offline
- Monitor the buoys, see if that their system is online, with all necessary resources to function correctly
- Monitor the Kafka consumers, especially their lag and fetch rate
- Monitor the processing nodes, their GPU usage and RAM performance
- Monitor the database and storage filesystem
- Record the system-level telemetry (CPU, Mem, IOPS, Disk%, ...) of all buoys, processing nodes, Kafka nodes, and database nodes
- Create visualization dashboards of the system's health
- Send some alerts with different levels of gravity according to the problems detected on unhealthy systems

4.2.6 MXNet

In order to run the inference, which is the labellisation of the image using the ML algorithm, the model needs to be exported and available on the processing nodes.

An apache project, cloud ready, and runs on tiny nodes for edge computing, like in our buoys, supports ONNX - update the model on 1 device, test it, and deploy the changes to all devices qusing nifi, model server. It runs as great on a raspberry pi for image labellisation than on a distributed multi GPU server

4.2.7 System Requirements

"dumb" buoys system requirements

The minimum system requirements for a "dumb" buoy setup are presented in Table 4.4.

Nodes :	Buoys	Servers	Compute	Training	Database
RAM	1GB +	32 GB ++	16 GB ++	32 GB ++	8 GB
CPU	not much relevant	Decent	Good	not relevant	Decent
GPU	-	-	-	Nvidia +++	-
Storage	64 GB	50 GB	50 GB	500 GB	1000 GB
How many	xN	x2	x1	x1	x1

Table 4.4: Minimum System Requirements with "dumb" buoys
+ signs means the higher the better

Buoys : With the dumb buoy setup, the objective is having the cheapest buoys. We can accomplish this by using cheap Raspberry Pies for the buoy, which cost not much and are plenty sufficient for this job.

Server nodes : As a rule of thumb, a Java Virtual Machine (JVM) needs 8 Gb of RAM. In the big data stack, almost all applications/services are in fact a JVM, and it's still true for Kafka Broker, NiFi nodes, Zookeeper, Flink, Hadoop nodes... We can start our system with 1 server node (that could then be scaled-up to 3), 1 processing node (that could be linearly scaled up), and 1 database node. However we are going to compare NiFi and Kafka implementation.

Compute nodes : The trained machine learning model takes 3 GB of space in RAM. To which we add up 1.3 Gbits for each image. A decent computation node should have at least 16 Gb of Ram (the higher the better). To label the images, the GPU is not relevant as we don't handle many images. We need a good CPU with many cores. Again, the bigger the better.

Training nodes : In order to train the model, and tests some compression algorithms as well as allowing some continuous improvements, we can expect to add a training node to train the model. It is different than a standard computing node such that all processing is done on a GPU, and all the training set needs to be available from storage. The processor, in this case, is not relevant.

"smart" buoys system requirements

The minimum system requirements for a "smart" buoy setup are presented in Table 4.6. Here nothing much changed except for the "Compute" nodes that are not needed anymore, and the buoys.

Nodes :	**Buoys**	**Servers**	**Training**	**Database**
RAM	4GB +	32 GB ++	32 GB ++	8 GB
CPU	quad core ++	Decent	not relevant	Decent
GPU	-	-	Nvidia +++	-
Storage	64 GB	50 GB	500 GB	1000 GB
How many	xN	x2	x1	x1

Table 4.6: Minimum System Requirements with "smart" buoys
+ signs means the higher the better

Buoys : This time the buoys needs to process their own images. The inference is mainly using RAM and CPU. The model takes 3GB in RAM, so that's already a minimum requirement to have. To label the images, the more CPUs the better. At decent quad-core should do the job fine.

Chapter 5

Evaluation

This chapter focuses on tests, measures, and implementation of the project.

5.1 First implementation tests and measures

We first searched the market for the available single board computers. Those would be an ideal choice to use on the buoys as they are usually cheap, can run linux, and don't consume much power. Our research is visible in Fig.5.1.

Board	Price ($)	Vendor	Processor	Cores	3D GPU	MCU	RAM	Storage	LAN	Wireless	USB ports	Power	Expansion	OSes
Pine H64 Model B	45	Pine64	Allwinner H6	4x A53 @ 1.5GHz	Mali-T720	no	3GB	opt.	GbE	WiFi, BT	2	5V DC jack	Pi 40	Linux, Android
Raspberry Pi 4 Model B	55	Rpi Trading	Broadcom BCM2711	4x A52 @ 1.5GHz	VideoCore IV	no	4GB	no	GbE	WiFi/BT	5	5V USB-C Q3, 5.5-20Vin	Pi 40	Linux
Rock Pi 4	75	Radxa	Rockchip RK3399	2x A72 @ 1.8GHz, 4x A53 @ 1.4GHz	Mali-T864	no	4GB	empty eMMC	GbE	opt. WiFi/BT	5	5V USB-C Q3, 5.5-20Vin	Pi 40, M.2	Linux, Android
Odroid-N2	90	Hardkernel	Amlogic S922X	4x A73 @ 1.8GHz, 2x A53 @ 1.9GHz	Mali-G52	no	4GB	empty eMMC socket	GbE	opt.	5	7.5-20Vin, 12V/2A DC jack	Pi 40	Linux, Android
RockPro64	80	Pine64	Rockchip RK3399	2x A72 @ 1.8GHz, 4x A53 @ 1.4GHz	Mali-T864	no	4GB	empty eMMC	GbE	opt. WiFi/BT	4	5V/3A type H	PCIe x4, Pi 40	Linux, Android
NanoPi M4	95	FriendlyARM	Rockchip RK3399	2x A72 @ 2GHz, 4x A53 @ 1.5GHz	Mali-T864	no	4GB	opt. eMMC	GbE	WiFi, BT	5	DC 5V/3A input or USB	Pi 40 and other	Linux, Android
Rock960 Model C	99	Vamrs	Rockchip RK3399	2x A72 @ 1.8GHz, 4x A53 @ 1.4GHz	Mali-T864	no	4GB	empty eMMC; opt. M.2	no	WiFi, BT	3	8-18V input with 12V, 2A DC jack	96Boards; opt. M.2	Linux, Android
Jetson Nano Developper Kit	99	Nvidia		4x A57 @ 1.4GHz	128x Maxwell		4GB	opt	GbE	opt.	5	5V 4A DC Jack	Pi 40	Linux
Khadas Vim2	100	Khadas	Amlogic S912	8x A53 @ 1.5GHz	Mali-T820 MP3	yes	3GB	32GB	GbE	WiFi,BT	3	5-9V input and USB-C	40-pin custom	Linux, Android
Renegade Elite (ROC-RK3399-PC)	100	Libre Computer	Rockchip RK3399	2x A72 @ 2GHz, 4x A53 @ 1.5GHz	Mali-T864	no	4GB	opt. eMMC to 128GB	GbE w/PoE	no	5	12V USB-C 15W	other	Linux, Android
NanoPC-T4	110	FriendlyARM	Rockchip RK3399	2x A72 @ 2GHz, 4x A53 @ 1.5GHz	Mali-T864	no	4GB	16GB eMMC	GbE	WiFi, BT	4	12V/2A DC jack	M.2, Pi 40	Linux, Android
Khadas Vim3	139	Khadas	Amlogic A311D	4x A73 @ 2.2GHz, 2x A53 @ 1.8GHz	Mali-G52 MP4	yes	4GB	32GB eMMC	GbE	WiFi,BT	3	5V USB-C Q3, 5.5-20Vin	40-pin custom	Linux, Android
Rock960 Model A and B	139	Vamrs	Rockchip RK3399	2x A72 @ 1.8GHz, 4x A53 @ 1.4GHz	Mali-T864	no	4GB	32GB eMMC; opt. M.2	no	WiFi, BT	3	8-18V input with 12V, 2A DC jack	96Boards; opt. M.2	Linux, Android
Odroid-H2	160	Hardkernel	Intel Celeron J4195	4x Gemini Lake @ 2.3GHz	Intel UHD Graphics 600	no	4GB and up	8GB eMMC and up; M.2, SATA	2x GbE	no	4	14-20V DC Jack	M.2, other	Linux
UP board	149 or 169	Aaeon	Intel Atom x5-Z8350	4x Cherry Trail @ 1.44GHz/1.92GHz	Intel HD 400	no	4GB	32GB or 64GB eMMC	GbE	no	5	5V DC jack	Pi 40	Linux, Android
UP Core	149 or 169	Aaeon	Intel Atom x5-Z8350	4x Cherry Trail @ 1.44GHz/1.92GHz	Intel HD 400	no	4GB	32GB or 64GB eMMC	no	WiFi, BT	1	5V DC jack	Pi 40	Linux, Android
Khadas Edge/Edge-V	149 to 230	Khadas	Rockchip RK3399	2x A72 @ 2GHz, 4x A53 @ 1.5GHz	Mali-T860	yes	4GB	32GB to 128GB eMMC	GbE (Edge-V)	WiFi, BT	4	5-20V USB-C	MXM with FPC or Pi 40 with M.2 (Edge-V)	Linux, Android
UP Squared	149 to 339	Aaeon	Intel Celeron N3350, to pentium Quad	2x Apollo Lake @ 1.1GHz/2.4GHz	Intel Gen9 HD 500/505	no	4GB or 8GB	32GB to 128GB eMMC, SATA	2x GbE	no	4	5V DC jack	other	Linux, Android

Figure 5.1: Pertinent selection from "Comparison of 125 Open Spec, Hacker Friendly Single Board Computers", LinuxGizmos.com [12]

As the decision has been made to run the algorithm in the buoys, we mainly axed our research to compare the embedded computers powerful enough to run the inference. As we said before, to run the inference we mainly need RAM and processing power. The GPU is not relevant. Since the model takes at least 3GB in RAM, we looked for boards that had at least 4GB RAM.

From that comparison, the best choice is definitively the **Odroid N2**, since devices under $75 aren't powerfull enough. In the $75-$130 area, the Odroid N2 is definitively the best of all, and it would wost at least twice the price to get a more powerfull processor, which might be, in the end, usefull, but we'll stick to this board for our tests, as it is a reference point in many benchmarks, and it will give us the data we need to know where to further develop.

The next goal is to test the implementation designed above, and validate the technical choices. For our first prototypes and tests, we can deal with only a few cameras, and even cheaper networking technologies. We need to tests implementations and makes measures first before beggining to scale up. For such, we will work with some little cheap computer boards (Raspberry Pi 3B+), a few usb cameras, and some Gigabit Ethernet and Wireless 802.11ac networking equipment.

Raspberry Pi (RPi) model 3B+ ~~is the latest to this day~~, is a micro-computer board that has a 1.4GHz 64-bit quad-core processor, dual-band wireless 802.11ac, Bluetooth 4.2, Fast Ethernet and usb-2 sockets [27]. We can install Linux and run our own program to prototype the buoy. It's ethernet traffic goes through the USB-2 chip, whith shares the throughput with all other USB-2 devices. So even with a Gigabit adaptor, the Ethernet speeds of the RPi won't go over 400Mbps. However this doesn't look like an issue since only the data from that buoy's cameras need to go through that port. All the other transmitted videos directly hops between switches on the network, without bouncing on each connected RPis. The wireless hardware on the RPI 3B+ however is measured with download speeds only up to 45Mbps [19].

Today, June 24th 2019, the Raspberry Pi foundation released the model 4 of their board. It now supports full throughput Gigabit Ethernet, USB-3, and Bluetooth 5. This board stays at the same price, even considering it's upgraded feature and new Quad Core Cortex-A72 1.5GHz processor. This is a great board. It even has upgraded models with 2GB and 4GB RAM, all of those versions under $55. However the device is out-of-stock, so I'll have to start some tests with the earlier model 3B+.

The Raspberry Pies, in the end, are the best alternatives to create "dumb" buoys. They are not powerful enough to run the AI algorithm at this stage. However I can still use a couple of Raspberry Pi 2 B+ that I already have laying around in my room to run some implementation tests and measures.

5.1.1 Test : Video acquisition and compression

1. Connect 2 usb cameras to a RPi to monitor and record the cameras
2. Test compression methods and measure the file sizes
3. Measure the ressources and time it takes to preprocess the videos

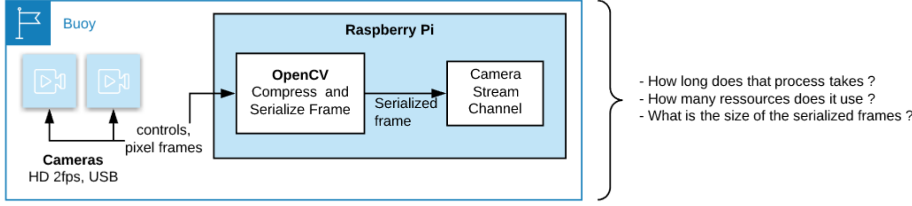


Figure 5.2: Illustration of the test subsection 5.1.1

5.1.2 Test : Network communication speeds

1. Create a network with 2 RPi each feeding live camera feeds to a computer
2. Measure the network speeds between Wifi 802.11ac and Gigabit Ethernet speeds over different loads and distances

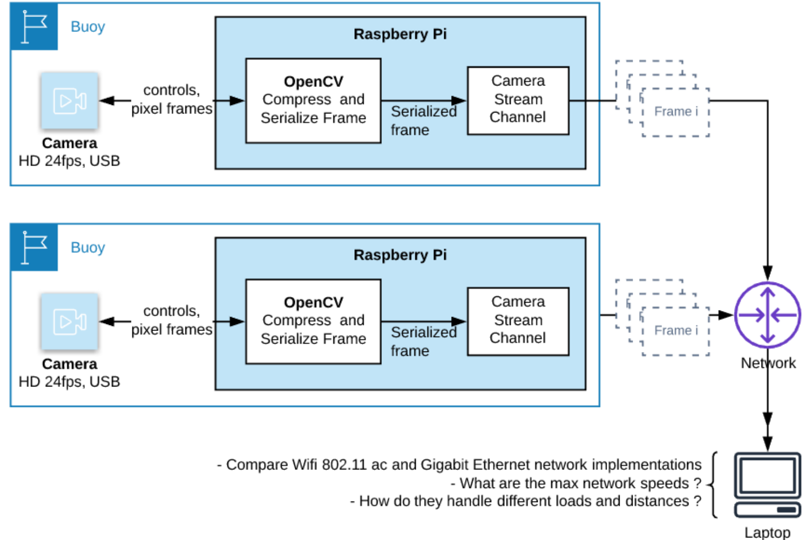


Figure 5.3: Illustration of the test subsection 5.1.2

5.1.3 Component list and necessary budget

I can use my own router at home to try some tests, but there is still hardware needed to implement the buoy, like ethernet Cat6. cables and a USB HD Camera. The Figure 5.4 lists the required components to buy before starting the next steps.

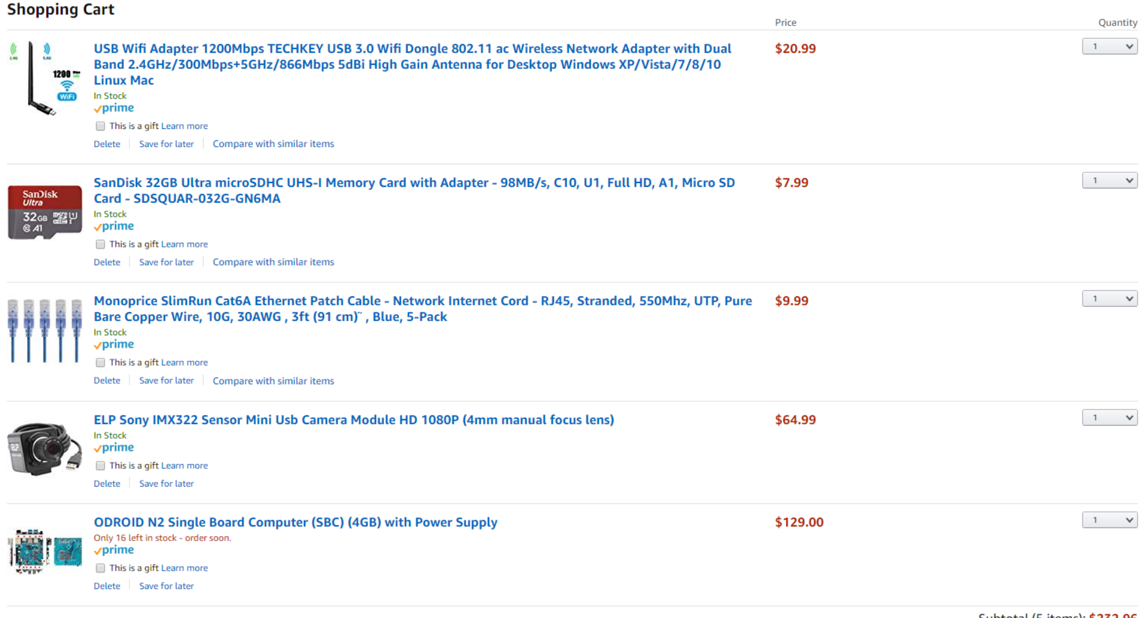


Figure 5.4: Amazon.com shopping list of hardware for the first tests and in prevision of later further tests

By doing those first tests, we can measure the performances of each block, so that we can better guide the hardware needed for the next phases. We looked for the cheapest options while keeping the required technical specifications to match our tests and later prototypes, as those components will likely be reused later on.

5.1.4 Unfortunate turn of events

As specified before, this project was in collaboration with the CRA research center at La Reunion. My contact was the director of the center, Mr Eric Chateauminois. He made the order on Amazon for the necessary components for those tests, but those never arrived. Unfortunately, the director then retired from his position, and I was not able to contact Amazon on behalf of him, and the other researchers at the center would not deal with this issue.

I decided to postpone those tests and simulate those environments with virtual servers, which is the focus of the next sections of this chapter.

5.2 NiFi implementation and simulations

As detailed above, NiFi was the best solution to manage the data flow for this project. So I designed 2 solutions in order to compare the "smart" and "dumb" configurations.

5.2.1 NiFi Implementation

All of this implementation is installed on custom Virtual Machines (VM) in the cloud. This allowed me to configure and modify them easily as suited for my later performance benchmarks.

The architecture is quite straightforward, with the buoys dealing with the capture, and the server dealing with the centralization, alerts and archives. With the NiFi user friendly interface, the whole process is easily controllable and monitored. The flows are pictured in the appendix. With NiFi, the buoys can all be managed via the server's web interface using templates.

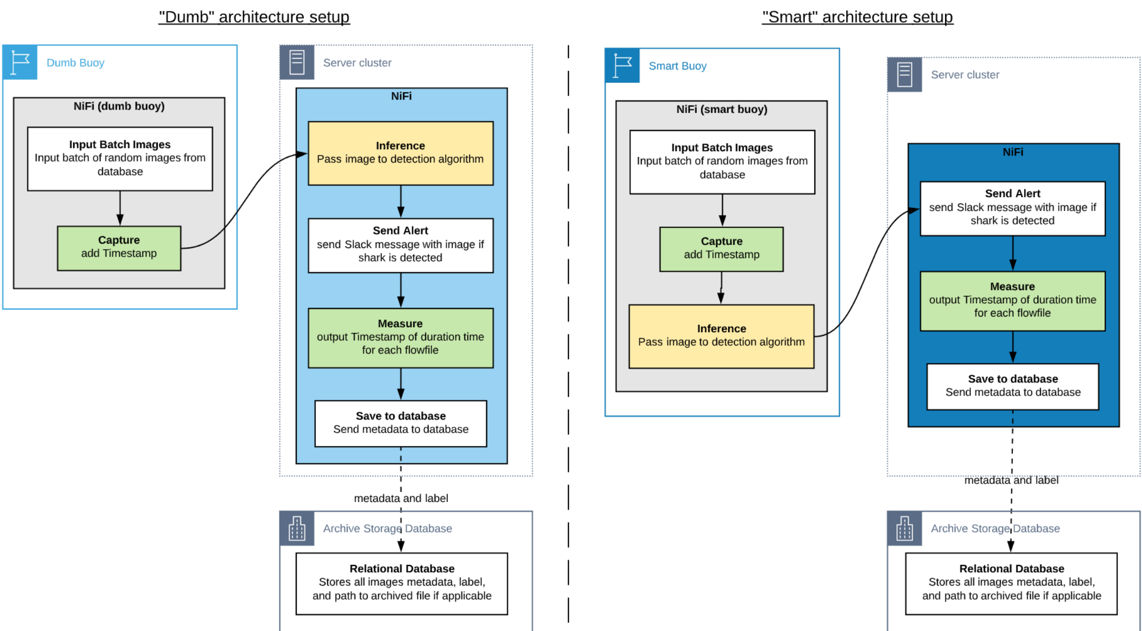


Figure 5.5: NiFi architecture for the tests

Site-to-Site communication : In order to send data from the buoys to the server, I'm using NiFi's Site-to-Site protocol. It's the recommended protocol from the documentation. In this configuration, the buoys are the clients, and the server is the server. The buoys sends data to a Remote Process Groups, which is then received

on the server on an input port. This allows me to maintain attributes and manage my input ports separately. Since the buoy might also have an unknown or changing IP address when being turned off or using 4G, this also simplifies the future configurations by having to use only the server's fixed IP in the configurations.

Input Data : The flow is easily managed in NiFi and I used diverse NiFi processors and features to simulate the input video data since I didn't have the cameras. I wanted to avoid writing custom code so that this project could be easily transferred to the research center at La Reunion, and usable for people without programming experience.

Inference : Usually when we learn to develop Machine Learning algorithm, we forgot about how to deploy them in production. We are mainly training and re-training the models. However I can't use the same tools here. The inference is cheaper than the training since all the training dataset doesn't need to be read, we just need the pre-trained model, and there's only 1 image being processed. For such, we don't need to use a GPU. We don't want either to use a custom script loading the model, and running the inference for each image. We would rather have a local server with the model pre-loaded in RAM, to which we can just make some calls with the file path, and it would return the labels.

In order to simplify processes in the future, we wanted a standard interface for the model, and asked that the model would be given in the universal Open Neural Network Exchange (ONNX) format. Etienne Meunier developed the model using PyTorch, and the direct PyTorch integration isn't greatly supported by NiFi and I wanted to avoid writing custom processors hard to maintain in the future. The ONNX format is an open source exchange format widely compatible, and I also chose to use the open source Apache MXNet project for the inference service, for compatibility and serviceability.

MXNet was also chosen because it proposed a lightweight C++ installation that could as easily run without Java with the MiNiFi C++ installation on a lightweight buoy.

Alert : I didn't had any specifications regarding the types of alerts to send, so I created a Slack bot that would connect to NiFi and send alerts to subscribed people. With Slack, this is a easy process, and the message can be sent in real-time and formatted to include the image, some text, and even some actions so that the reader can give some feedback or look for more details.

It's easy to add any type of alerting, and I even added an other type of alerts that sends me SMS with a succinct text message and an URL to check out for more details.

Database : Even though NiFi is a great monitoring and traceability tool, with advanced database to store the flowfiles metadata, I setup a custom PostgresSQL database on a separate VM to upload there the flowfile's attributes for further analytics.

Measuring : For our tests and analytics, we need to measure exactly how each image takes before the corresponding alert can be sent. This is why we added tome custom processors in NiFi to add attributes.

5.2.2 NiFi Tests

The goal of the following tests is to compare different buoy and network configurations. Using Virtual Machines (VM), we are enabling and disabling CPUs, as well as throttling the network. When dealing with large batches of images, and comparing between a "smart" and "dumb" buoy in those configurations, we want to gain insights on the feaseability of those implementations.

Test Procedure

1. Using a custom script, I configure the buoy VM into the desired CPU and network configuration
2. A variable is set into NiFi with a `test-id` to later match the entries to their test in my database
3. I use a NiFi processor to load a batch of 100 or 1000 images from the CRA's video database, and hold them in queue
4. The test is then run and all the start time is instantly appended each flowfile's attributes
5. Depending on the test, the flowfiles are sent now or later to the server (dumb vs smart)
6. The image is processed on an local MXNet server hosting the inference model, and the result is appended in the flowfile's attributes
7. If a shark is detected, an alert is immediately sent in Slack, with the attributes and the image
8. The flowfile's total duration is then computed and appended in its attributes
9. The flowfile's attributes are then added to a analytics postgres database running on another VM instance

Test Specifications

The buoys and main server are running Ubuntu 18.04.3 LTS, nifi-1.9.2, Java 8, and mxnet-model-server 1.0.5.

- the buoy is a VM on Google Cloud, on network us-east4, with configuration n1-highcpu-8 (ie. 8 VCPUs Intel(R) Xeon(R) CPU @ 2.20GHz, 7.2GiB RAM)
- the main server is a VM in the MIT, with only 4 CPUs (Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz) but 16GiB of RAM

I am then disabling CPUs on the buoy, and throttling the network speeds according to tables 4.1 and 4.2. All the batch's images (100 or 1000) are preloaded in NiFi's queue and asked to be processed immediately.

5.2.3 NiFi Test Results

The Figure 5.6 compares the average flow-file's duration between all the different configurations. The lower the better. The duration displayed in seconds in the average time an image spends in the processing since the batch has been opened up until after the alert has been sent. This is an average over 100 or 1000 images.

A simple and boring boxplot shows that those times are evenly distributed between 0 second and 2x the average. We can already see in NiFi during the tests that the bottlenecks are the inferences, at which the whole batch is queued before being processed. It's amazing to see that the fastest time to process the first image is always below 1 second (cf Table 5.1). **This can give us a baseline ≤ 1 second for the latency** . And it's worth noting too that the max processing time is of course and thankfully less than $batch_size * min_processing_time$.

	"dumb" buoys	**"smart" buoys**
Average Min. Processing Time	0.17 s	0.89 s
Max. Processing Time (batch 100)	25.88 s	10.80 s
Max. Processing Time (batch 1000)	261.34 s	109.43 s

Table 5.1: Min and Max processing times between implementations

On figure 5.6, I plot the comparison of 24 configurations, evaluating batches of 100 and 1000 images across 3 different network speeds, and limiting the number of processors.

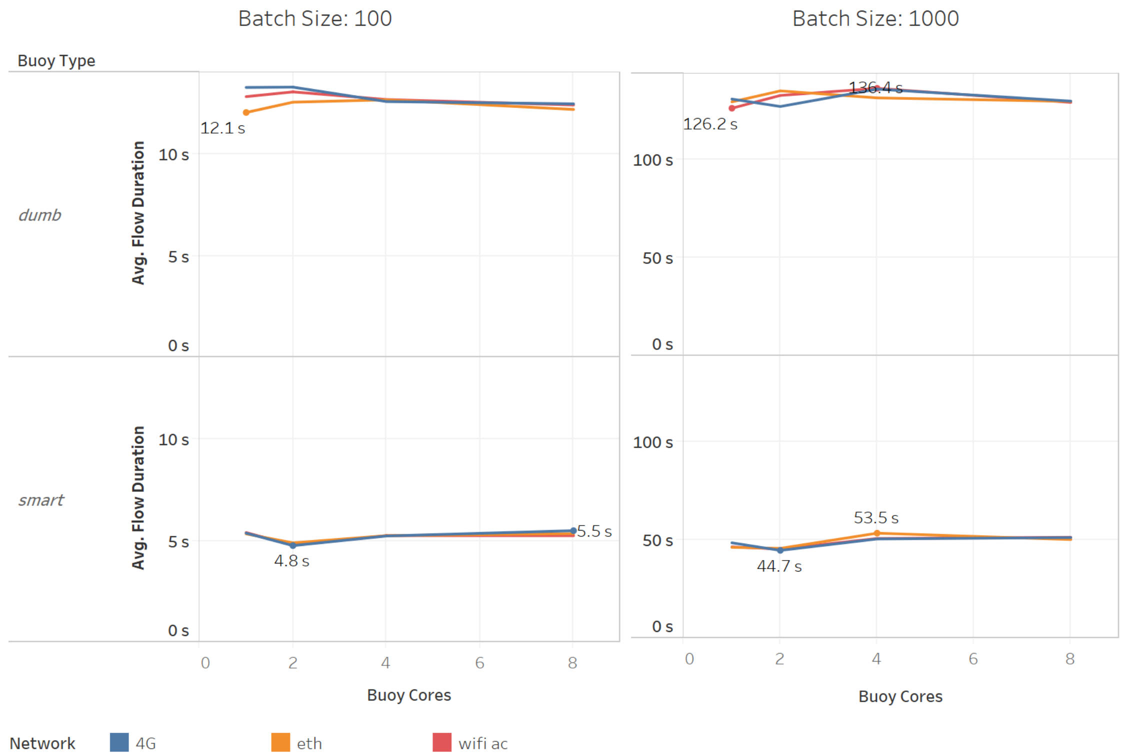


Figure 5.6: NiFi Simulation Test Results

At first glance it looks like the network speed isn't such a problem as we initially expected. Even using 4G, the result are in the same range as with WiFi or Ethernet. We can also notice that a high number of CPU didn't reduce the processing time as we expected. I should clarify right away that I double checked that the network speed was indeed limited and that the CPUs were indeed disabled.

Network speed and compression

We might have wanted to deal more with the compression of our data. Since a lot of images are sent over the network, it seems an important point not to overlook. However results shows that the network isn't the bottleneck at all yet, and NiFi is already doing a great job optimizing the network capabilities already. It's still worth noting that while the network speeds has almost no effect on the "smart" buoy configurations, its effects are increased in the "dumb" buoy setup, which makes sense knowing since there is, as already justified, more traffic with the "dumb" buoy setup.

Batch Size

Interestingly, we see that the processing times are linearly correlated with the batch size. All the average and maximum processing times for the tests with `batch_size = 1000` are 10 times the times for `batch_size = 100`. By subtracting the latency of $\sim$ 1 second, those consistent results gives us a baseline formula to compute the maximum number of cameras we can process depending on the delay we can allow for a specific configuration.

For instance, since the number of cores doesn't create a big enough variation, and matching our baseline of HD cameras at 2 fps, we computed that a "smart" buoy with those configurations should be able to process correctly without delay (ie. only the latency) about 20 frames per second. This corresponds to 10 HD cameras at 2 fps.

A "dumb" buoy, on the other hand, could only handle 8 images per second with this configuration, which is equivalent to 4 HD cameras at 2 fps.

This is really surprising me as I was expecting the dumb buoy configuration to being able to process more images than the smart configuration.

Dumb vs. Smart configurations

The "dumb" buoys have approximately 3 times the delay of the "smart" buoys. This is a lot and can be quite unexpected. It's worth noting that the main server which is running the inference in the "dumb" scenario is having only 4 CPUs, and should be quite in the same configuration as the buoy when the buoy is using only 4 CPUs. Since it still takes 3x longer in those scenarios, we can also remove the RAM from the equation, because the server already has 2x the RAM of the buoy. I expect this delay to be occurring from the overhang of running the inference on images that are originally stored on the buoys. And as expected, that results in more processing time for the "dumb" buoys.

Those comparisons might not be conclusive enough, and more tests and investigations can still be done. However based on the data we have, a "smart" buoy configuration with 4G dongles will be a effective and cost-efficient implementation.

Chapter 6

Future Work

This research document details the considerations of implementing underwater buoys with cameras to detect sharks in real time. I focused primarily on the Information System's Network architecture concepts, limitations, and implementations. We now know that a "smart" buoy setup with 4G connectivity is a suitable effective and cost-efficient configuration.

On a personal side, I am already really happy and proud with this research as I learned a lot about streaming architecture, event processing, in depth kafka and nifi architecture, and deploying a Machine Learning model. This is also a really useful project that will be continued, and I learned a lot about approaching a project with a researcher's mindset.

As the scope of this project is very large, and that I had only 2 months to work part-time on this project, I have let some parts of the projects to be done later.

- ☐ **Data Compression** : the frame objects could greatly benefit from further compression before being sent over the network. A good and fast compression algorithm in the buoys, and decompression in the processing nodes, should increase the capacity of the network in terms of cameras. NiFi would make this step easier should it be the selected architecture.

- ☐ **Security of the system** : Setup kerberos authentication ? SSL or not ? NiFi already handles SSL encryption of the data, and authentication mechanisms should not be complex to set up.

- ☐ **Notification to send when shark is detected** : This can be done in many ways, either in the Relational Database, either with Flink in the Kafka cluster connected to the output topic, or either in NiFi that can connects to messaging apps like Slack and send instant messages with the corresponding image.

- **System monitoring and management** : Zabbix has been presented in subsection 4.2.5. It should be known if it's implementation is useful and should be implemented to monitor system's health.
- **Designing the Buoys** : The design of the underwater buoys is still to be conceived and prototype. More research should be done on the configuration of the underwater cameras.

Chapter 7

Appendices

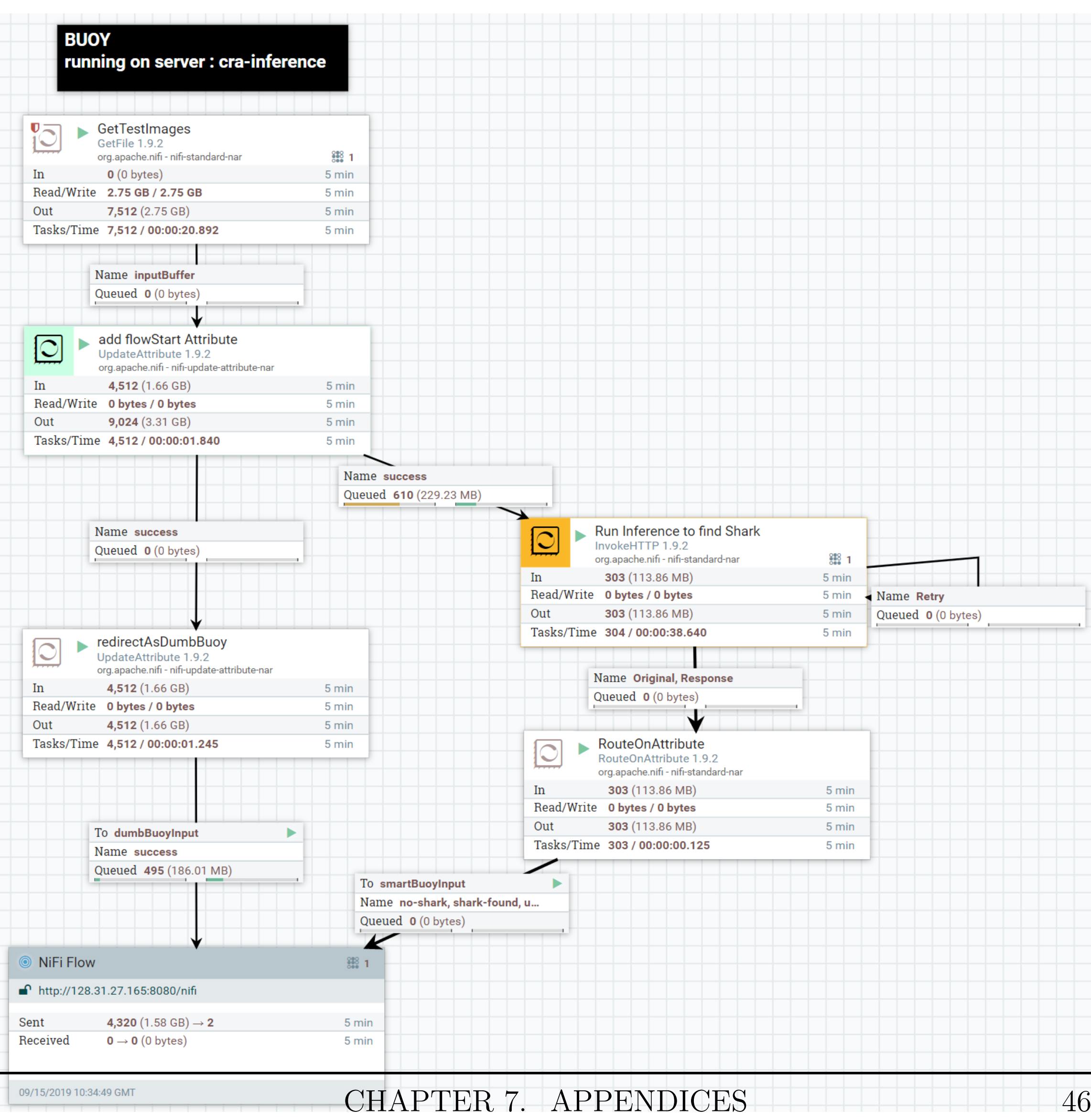


Figure 7.1: NiFi flow on the buoys

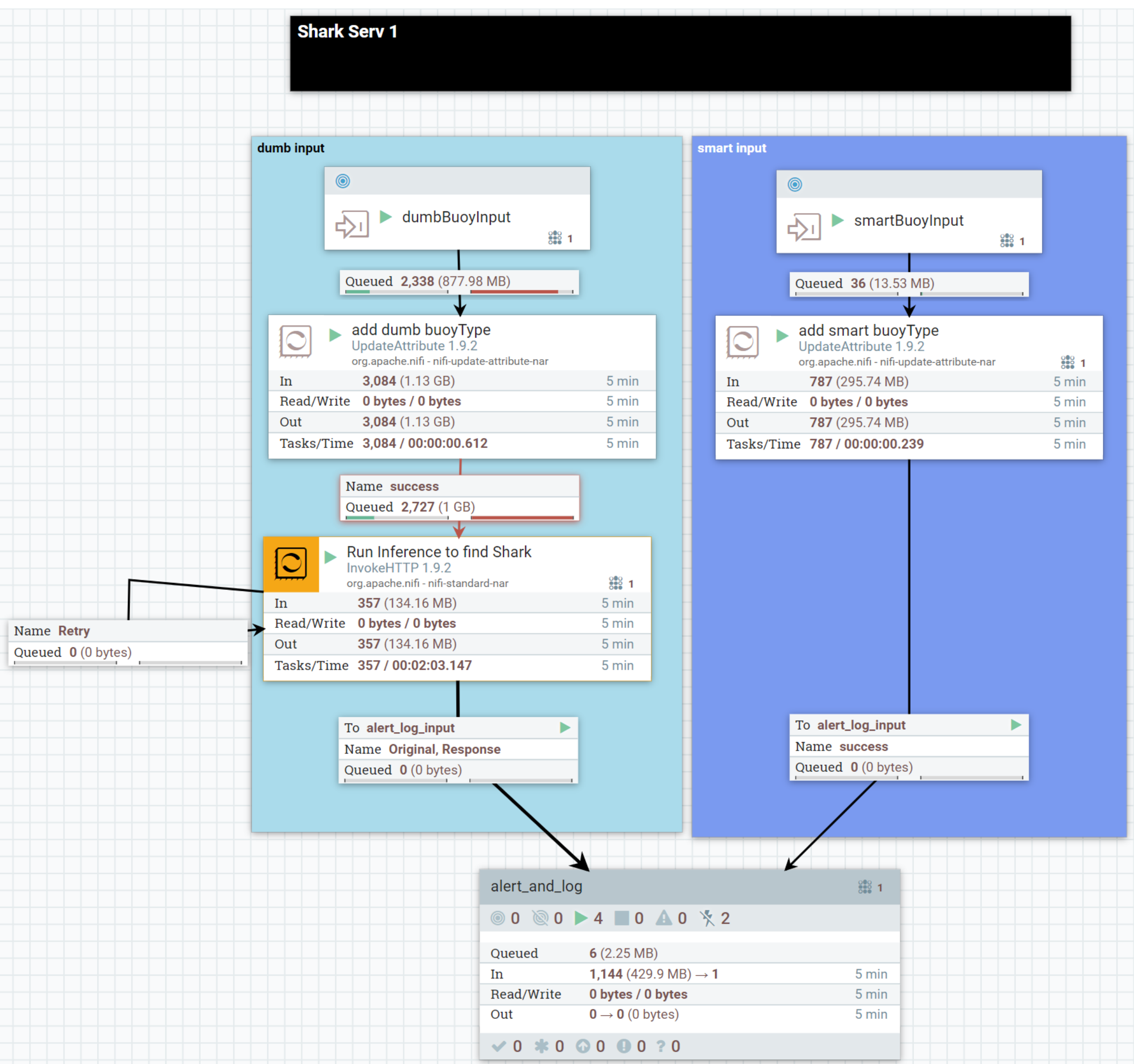


Figure 7.2: NiFi flow on the server

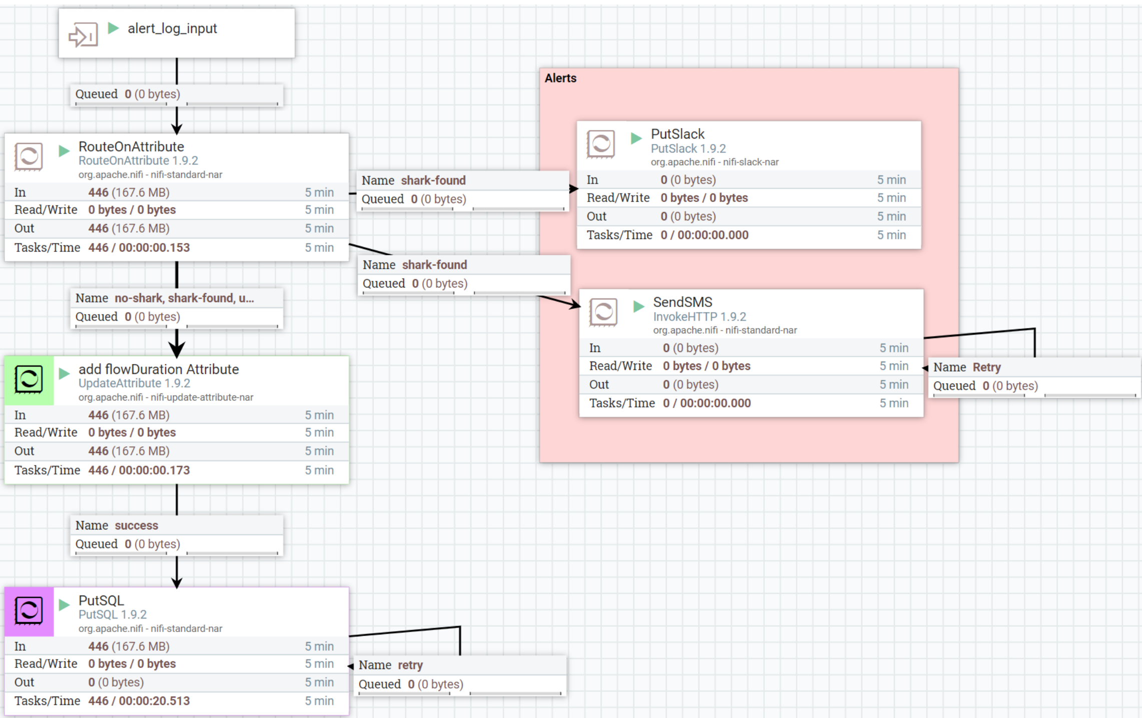


Figure 7.3: NiFi alert and log flow on the server

Bibliography

[1] *10 Gigabit Ethernet*. en. Page Version ID: 899541661. May 2019. URL: https://en.wikipedia.org/w/index.php?title=10_Gigabit_Ethernet&oldid=899541661 (visited on 06/15/2019).

[2] *5G*. en. Page Version ID: 901845141. June 2019. URL: https://en.wikipedia.org/w/index.php?title=5G&oldid=901845141 (visited on 06/15/2019).

[3] *8-N-1*. en. Page Version ID: 896581180. May 2019. URL: https://en.wikipedia.org/w/index.php?title=8-N-1&oldid=896581180 (visited on 06/15/2019).

[4] *Apache Kafka*. en. URL: https://kafka.apache.org/21/documentation/streams/architecture (visited on 06/23/2019).

[5] *Apache NiFi Overview*. Apr. 2019. URL: https://nifi.apache.org/docs/nifi-docs/html/overview.html (visited on 06/24/2019).

[6] *Apache_NiFi_In_Depth*. Apr. 2019. URL: https://nifi.apache.org/docs/nifi-docs/html/nifi-in-depth.html (visited on 07/06/2019).

[7] Scott Arena. *MET CS 625 Business Data Communication and NEtworks Boston University*. Jan. 2019. URL: https://www.bu.edu/academics/met/courses/met-cs-625/ (visited on 06/14/2019).

[8] Amit Baghel. *Video Stream Analytics Using OpenCV, Kafka and Spark Technologies*. Sept. 2017. URL: https://www.infoq.com/articles/video-stream-analytics-opencv/ (visited on 06/15/2019).

[9] *Bandwidth (computing)*. en. Page Version ID: 900066956. June 2019. URL: https://en.wikipedia.org/w/index.php?title=Bandwidth_(computing)&oldid=900066956 (visited on 06/15/2019).

[10] News BBC. *Shark attack: Surfer killed off France's Réunion Island - BBC News*. URL: https://www.bbc.com/news/world-europe-48219268 (visited on 06/14/2019).

[11] Ken Bromhead. *Apache Kafka: Ten Best Practices to Optimize Your Deployment*. Oct. 2018. URL: https://www.infoq.com/articles/apache-kafka-best-practices-to-optimize-your-deployment/ (visited on 06/23/2019).

[12] Eric Brown. *Catalog of 125 open-spec hacker boards*. June 2019. URL: http://linuxgizmos.com/catalog-of-125-open-spec-hacker-boards/ (visited on 06/29/2019).

[13] *Climate*. en. Nov. 2014. URL: https://en.reunion.fr/practical/reunion-island/geography-and-climate/climate (visited on 06/16/2019).

[14] *Comparison of wireless data standards*. en. Page Version ID: 900373760. June 2019. URL: https://en.wikipedia.org/w/index.php?title=Comparison_of_wireless_data_standards&oldid=900373760 (visited on 06/15/2019).

[15] CRA. *Le Centre de Ressources et d'Appui pour la réduction du risque requin, Acceuil*. fr. Apr. 2019. URL: http://www.info-requin.re/le-centre-de-ressources-et-d-appui-pour-la-r70.html (visited on 06/14/2019).

[16] Reunion CRA. *Test II du Sonar Xblue : un dispositif innovant pour la sécurisation de la gauche de Saint-Leu face au risque requin*. fr. Apr. 2019. URL: http://www.info-requin.re/test-ii-du-sonar-xblue-un-dispositif-innovant-pour-a872.html (visited on 06/14/2019).

[17] Peter Forret. *Megapixel calculator | toolstudio*. en. 2018. URL: https://toolstud.io/photo/megapixel.php?width=1920&height=1080&compare=video&calculate=compressed (visited on 06/15/2019).

[18] FranceInfo. *Le taux de chômage augmente à La Réunion de 2 points*. fr. Apr. 2019. URL: https://la1ere.francetvinfo.fr/reunion/taux-chomage-augmente-reunion-2-points-697356.html (visited on 06/14/2019).

[19] Jeff Geerling. *Getting Gigabit Networking on a Raspberry Pi 2, 3 and B+ | Jeff Geerling*. URL: https://www.jeffgeerling.com/blogs/jeff-geerling/getting-gigabit-networking (visited on 06/16/2019).

[20] *ISO/IEC 11801*. en. Page Version ID: 892552170. Apr. 2019. URL: https://en.wikipedia.org/w/index.php?title=ISO/IEC_11801&oldid=892552170 (visited on 06/15/2019).

[21] Yoon-Ki Kim and Chang-Sung Jeong. "Large Scale Image Processing in Real-Time Environments with Kafka". In: 2017. DOI: 10.5121/csit.2017.70120.

[22] Stanislav Kozlovski. *Thorough Introduction to Apache Kafka™*. Dec. 2017. URL: https://hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1 (visited on 06/22/2019).

[23] *List of interface bit rates*. en. Page Version ID: 900218833. June 2019. URL: https://en.wikipedia.org/w/index.php?title=List_of_interface_bit_rates&oldid=900218833 (visited on 06/15/2019).

[24] Tony Mancill and pwpadmin. *Best Practices for Apache Kafka*. en-US. Aug. 2018. URL: https://blog.newrelic.com/engineering/kafka-best-practices/ (visited on 06/23/2019).

[25] Sue Palminteri. *You don't need a bigger boat: AI buoys safeguard swimmers and sharks*. en-US. Apr. 2018. URL: https://news.mongabay.com/2018/04/ai-buoys-safeguard-swimmers-and-sharks/ (visited on 06/14/2019).

[26] Raymond R. Panko. *Business data communications*. Upper Saddle River, NJ: Prentice Hall, 1997. ISBN: 978-0-13-308164-0.

[27] RaspberryPi. *Buy a Raspberry Pi 3 Model B+ – Raspberry Pi*. URL: https://www.raspberrypi.org (visited on 06/16/2019).

[28] Joshua Robinson. *High-Performance Input Pipelines for Scalable Deep Learning*. en-US. Mar. 2019. URL: https://dataworkssummit.com/barcelona-2019/session/high-performance-input-pipelines-for-scalable-deep-learning/ (visited on 06/22/2019).

[29] Sourav Sikdar, Kia Teymourian, and Chris Jermaine. "An experimental comparison of complex object implementations for big data systems". In: Sept. 2017, pp. 432–444. DOI: 10.1145/3127479.3129248.

[30] Michael Slezak. *Shark nets used at most beaches do not protect swimmers, research suggests | Environment | The Guardian*. Feb. 2016. URL: https://www.theguardian.com/environment/2016/feb/09/shark-nets-used-at-most-beaches-do-not-protect-swimmers-research-suggests (visited on 06/14/2019).

[31] Timothy Spann. *Apache Deep Learning 201*. en-US. Mar. 2019. URL: https://dataworkssummit.com/barcelona-2019/session/apache-deep-learning-201/ (visited on 06/22/2019).

[32] Tritech. *Tritech Successfully Deploys Shark Detection Sonar | Tritech | Outstanding Performance in Underwater Technology*. Feb. 2017. URL: https://www.tritech.co.uk/news-article/tritech-successfully-deploys-shark-detection-sonar (visited on 06/14/2019).

[33] Itai Yaffe. *Stream, Stream, Stream: Different Streaming Methods with Spark and Kafka*. en-US. Mar. 2019. URL: https://dataworkssummit.com/barcelona-2019/session/stream-stream-stream-different-streaming-methods-with-spark-and-kafka/ (visited on 06/22/2019).

[34] Itai Yaffe and Yakir Buskilla. *Counting Unique Users in Real-Time: Here's a Challenge for You!* en-US. Mar. 2019. URL: https://dataworkssummit.com/barcelona-2019/session/counting-unique-users-in-real-time-heres-a-challenge-for-you/ (visited on 06/22/2019).