Dynamic vision interface for TurtleBot

Auteur : Courtoy, Boris
Promoteur(s) : Franci, Alessio; Sacré, Pierre
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20445

University of Liège - School of Engineering and Computer Science

Dynamic vision interface for TurtleBot

Master's thesis completed in order to obtain the degree of Master of Science in Computer Science and Engineering

COURTOY Boris

Supervisors

FRANCI Alessio
SACRÉ Pierre

Academic year 2023-2024

I would like to sincerely thank my supervisors Alessio Franci and Pierre Sacré, without whom this thesis would not have been possible. I enjoyed working with them and truly appreciated their commitment and advice which have greatly contributed to the success of this thesis.

I also want to express my gratitude to Sven Goffin for his assistance, his commitment and his availability during my thesis, and to Hugo Blayes for his help regarding the TurtleBot and the final tests.

Finally, my last thanks will go to my parents, my brother and my friends for their invaluable support throughout the year.

Abstract

The goal of this master's thesis is to develop a ROS2 driver able to retrieve and communicate the different data captured by the event-based cameras from iniVation. The implementation includes a capture node which reads the data from a camera and publishes them to their corresponding topics, as well as two other nodes in charge of the visualization of the data. To assess the quality of the driver, we performed a latency analysis and ensured that it was able to transmit the data in a few milliseconds. Following this, we designed as a proof of concept a way to control a TurtleBot based on the output of the camera. Finally, we completely integrated the camera with the TurtleBot. Facing latency issues, we managed to show that the Raspberry Pi on which the TurtleBot is built was not powerful enough to process all the events captured by the cameras. We eventually managed to overcome these problems by limiting the number of events processed by the Raspberry Pi.

Contents

Chapter 1

Introduction

With autonomous and intelligent systems being a main engineering field of study nowadays, the concern of having efficient and suitable sensors has never been more relevant. Indeed, as most of those systems rely on sensors retrieving data from the physical world and sending them to actuators in charge of modifying their behavior, using sensors of better quality would result in a more accurate representation of the world thus enhancing the accuracy and capabilities of the system.

In the context of this master's thesis, we will work with event-based cameras. These sensors represent a remarkable technological outcome of neuromorphic engineering, which is the engineering field studying the human brain in order to take advantage of its mechanisms to design brand new technologies relying on what is called brain-inspired computing. In the specific case of the event-based camera, the main idea is to reproduce the behavior of the human eye and in particular of the retina within an image sensor. In practice, these modifications completely alter the output of the camera replacing frames with events capturing a change in brightness in a particular location. This brings several advantages such as a better temporal resolution or a higher dynamic range.

Our objective with this thesis is to develop a robotic interface allowing the communication between an event-based camera and a robot. To do this, we will use the cameras from iniVation[1] in combination with a TurtleBot3[2]. We will implement this compatibility layer using the Robot Operating System (ROS)[3], which is among the most popular frameworks used in robotics. It is widely used in both the academic world for educational purposes and in the industry to develop professional robotic projects. The framework relies on message passing through a node and topic architecture. Each node manages a specific part of the robot, and the different nodes are able to communicate together by publishing or subscribing to topics. IniVation already developed a ROS1 package for their cameras, but as for now, nothing is available in ROS2, which is the latest ROS version. To overcome the lack of an iniVation ROS2 package, we decided to implement our own ROS2 driver to manage the integration of their cameras in any ROS2 project. We believe that our driver

could be used on one hand for educational purposes within the neuromorphic engineering laboratory of the university, and on the other hand by any developer willing to integrate neuromorphic vision through the iniVation cameras within a project. Doing so, we managed to create a driver establishing a reliable communication with the TurtleBot, and which is able to communicate batches of thousands of events in a few milliseconds.

The content will be organized in the following way. We will first present the event-based technology in detail. We will start with a review of the human eye in order to fully capture the biological inspirations behind those sensors, before presenting their underlying principles and architecture.

Following this, we will explain the implementation of our driver, covering the different building blocks on which it relies on, as well as the different nodes and topics composing it.

Afterward, we will go into more details concerning the material and setup used during the implementation and the tests of the package.

Finally, we will present the different experiments performed to assess the quality of our driver as well as the results obtained. We will start with a latency analysis to be sure that the data are retrieved in a sufficiently low amount of time. We will then develop as a proof of concept a small application allowing us to control the TurtleBot depending on the events captured by the camera. To conclude, we will try to completely integrate the camera with the robot by running the driver on its Raspberry Pi.

Chapter 2

Event-based camera

In this chapter, we crawl in depth into the details of functioning of an event-based camera. We will first provide background information in order to gather a better understanding of the event-based principles. We will then introduce the technology and compare it to a frame-based camera in order to highlight the advantages such sensors can provide. We will then conclude by presenting a few examples of applications in which event-based sensors can be of practical interest.

2.1 Background

Event-based cameras also called neuromorphic cameras are visual sensors trying to mimic the human sight's mechanisms. They first appeared under the name of "Silicon retina"[4]. In particular, while being inspired from the human eye behavior, these sensors are designed to specifically take advantage of dynamic vision.

The first step of this thesis is to introduce a sufficient biological background to fully understand the inspirations and the principles of the event-based technology. This will be done through a review of the human eye and retina mechanisms. In addition to this, we will also provide background information concerning image sensors.

2.1.1 Eye and retina

The human eye is composed of several parts with each of them having their own utility in the sight process[5]. A very simplified view can be observed in Fig.2.1.

Light is entering the eye through the cornea which is a transparent layer covering the front of the eye. It is working first as a protection layer enclosing the eye and keeping it safe from external factors, and secondly, thanks to its transparency, as a refractive layer receiving light and making it converge into the pupil.

The pupil is a small hole in the eye controlling which quantity of light is allowed to reach the retina. Its size is regulated by the iris, a thin membrane that can be

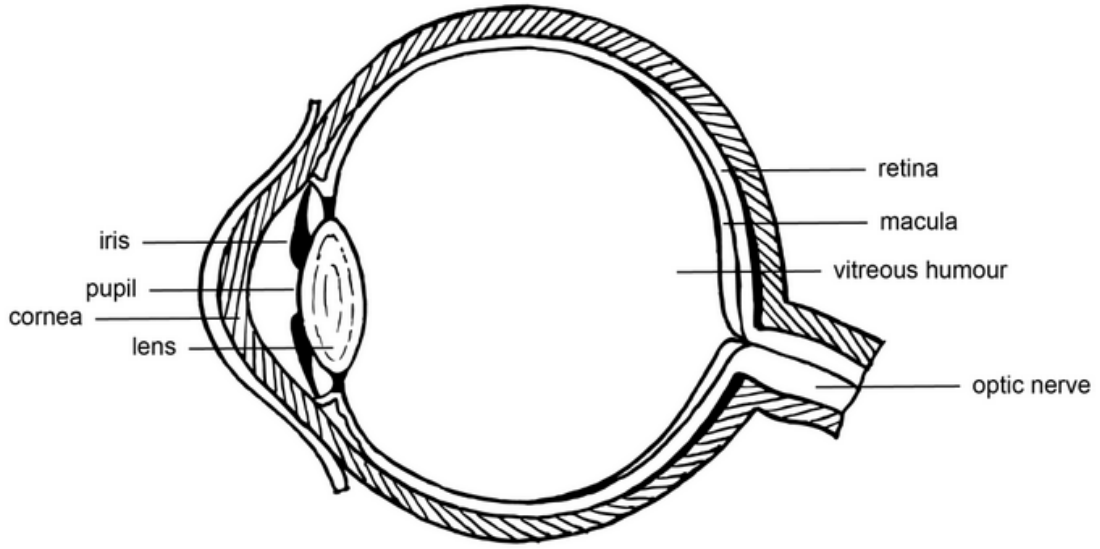

Figure 2.1: Simplified eye overview [5]

contracted or dilated.

Right after the pupil, light crosses the lens which is in charge of refracting it before it is projected onto the retina on the back of the eye through the vitreous humour.

The retina is full of photosensitive cells converting the light beams striking them into eletrical signals in a phenomenon called phototransduction[6]. These signals are sent to the brain through the optic nerve which establishes the connection between the eye and the brain.

We can mention different types of photoreceptors located into the back of the retina. First, the rods reacting to dim light allowing us to see in darker environments, and secondly, the cones managing higher light levels and taking care of our color perception.

Once these photoreceptors receive light signals, they convert them into electrical signals that are sent through a bunch of cells and synapses, eventually reaching the ganglion cells on the front of the retina [6]. A summary of this communication pipeline is shown in Fig.2.2 with the inner surface of the retina being on the left and the outer surface on the right.

These ganglion cells take the processed signals as input and send them to the brain through their axons forming the optic nerve. It has to be noted that the area of the retina from which the axons of the ganglion cells converge is called the optic disc and do not contain any photoreceptors, causing a blind spot in the human eye. Based on the information it has received, the brain is able to reconstruct the final image corresponding to what is actually seen.

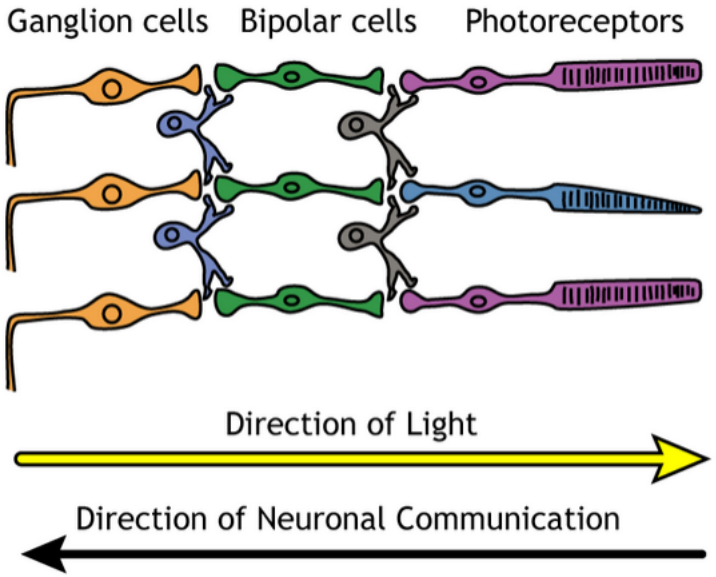

Figure 2.2: Retinal cells communication [6]

There is one important point in the context of this thesis when it comes to human sight mechanisms : the only cells firing action potentials are the ganglion cells. An action potential can be very simplified as a spike of current triggered by a neuron when it has reached its excitability threshold[7].

The interpretation we can give to such behavior is that since the travelling distance of the signal from the photoreceptors to the ganglion cells is rather short, the information goes from one side to the other in a continuous way in the form of an analog signal. However, when the information has to cross the optic nerve, as the travelling distance is way longer, a spiking signal is preferred.

Most importantly, since the action potential is fired when the ganglion cell input reaches a certain intensity threshold, this means the information is being sent to the brain only when the changes of light perceived by the photoreceptors are strong enough. This makes the brain focused mostly on brightness variations instead of absolute brightness. The ability to specifically target variations in an observed scene is what we call dynamic vision.

2.1.2 Standard camera

Even while being different, a regular camera is still very similar to the eye. Indeed, light is entering the camera through a lens sharing several properties with the elements in the front of the eye (diffraction, focus and light quantity regulation) before crashing onto photoreceptors. These photoreceptors will once again convert the light in electric signals that will be processed in order to generate an image. However, the key difference compared to the human retina lies in the way these photoreceptors are being triggered.

Biologically, the photoreceptors from the retina are being triggered independently from each others and in an asynchronous way, in the sense that as soon as it is activated, it forwards the electrical signal to the next bipolar cells. In the case of a frame-based camera, photoreceptors are used to accumulate charges depending on the intensity of light. These charges are being accumulated over a small period of time, the integration time, which is directly responsible for the frame rate of the camera.

The image sensors used to capture light are mainly of 2 types : CCD or CMOS. [8][9][10][11]

2.1.3 CCD sensor

A CCD (Charged Couple Device) sensor is composed of an array of photosites (also called pixels[1]) which are the components responsible for the conversion of light into electrical charges and their storage. Charges generated are directly proportional to the amount of light received. Once the integration time is over, the charges have to be processed to generate an image, and the photosites have to be freed in order to capture light from the next frame. To do so, CCD sensors rely on a shift register.

The idea is to use a dedicated storage area that will receive the different charges before converting them into voltage. Charges are transferred from their row to the adjacent one, and the ones on the edge of the array are transferred into the shift register. This one is then sending each charge through a pipeline composed of a charge-to-voltage converter node, an amplifier and an analog-to-digital converter. The final signals are then processed to generate the corresponding image. The whole mechanism is driven by a dedicated circuit. This shifting architecture can be observed in Fig.2.3.

[1]Not to be confused with pixels indicating the resolution of a screen or an image

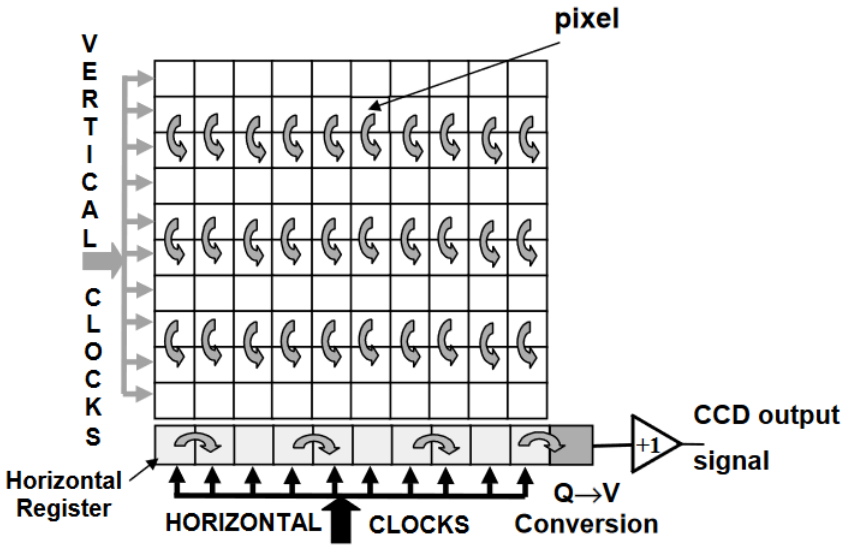

Figure 2.3: CCD architecture [11]

2.1.4 CMOS sensor

A CMOS (Complementary Metal Oxide Semiconductor) sensor is also composed of an array of photosites, but is very different than a CCD one in the way charges are processed. Indeed, this time there is no shift register, each pixel is an active component as it also manages charge-to-voltage conversion and embeds an amplifier. It is also provided with an addressing mechanism allowing a pixel-wise access. This mechanism is composed of a row-addressing module used to retrieve voltages from a selected row, and a column-adressing module composed of a multiplexer and switches allowing to access the signal of a particular pixel in the row. This architecture avoids managing charge transfers from one part of the sensor to the other, instead it is directly working in voltage domain. In addition to this, pixels can be addressed in a random-access manner instead of having to wait for all the previous rows to be shifted. Such an architecture can be observed in Fig.2.4.

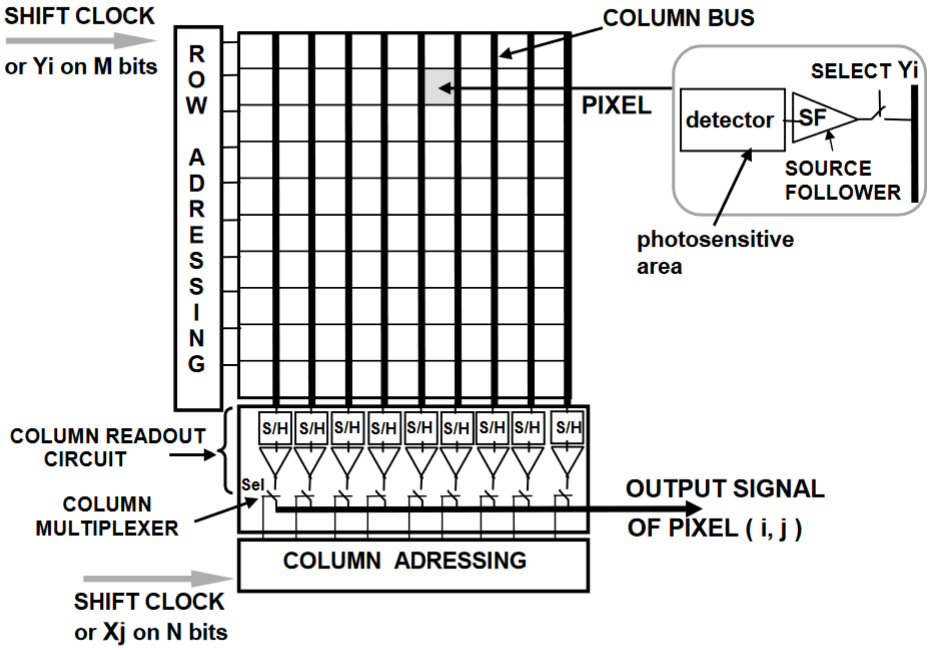

Figure 2.4: CMOS architecture [11]

Initially, CMOS sensors were more sensitive to noise than CCD due to the additional circuitry needed to ensure the pixel-wise architecture, thus producing images of lower quality. Nevertheless, it was showing better results in different aspects such as processing speed or power consumption[12]. CCD sensors were thus preferred in areas with a concern for image quality. However, recent improvements toward CMOS sensors made them catch up with CCD sensors, winning popularity over them for some applications.

2.2 Event-based vs frame-based camera

2.2.1 Motivations and main principles

Despite progress being made years after years in the image capture domain, visual sensors still struggled when being compared to the human eye[13]. Indeed, performance gaps were identified in different domains such as temporal resolution for example. Recognizing this disparity, researchers and engineers have been inspired to design new devices directly inspired from the human eye in the hope of reducing the gaps[4].

Taking advantage of the mechanisms described in section 2.1.1, the main objectives are to get rid of the frame rate by processing pixels independently from each others in

a continuous and asynchronous way in order to obtain a better temporal resolution, and to avoid redundancy by focusing on variations in the observed scene. This is done by replacing the frame-based architecture with an event-based one. Here, instead of generating a new frame every short period of time by processing each pixel, we are interested in events, which characterize a change of brightness in a particular pixel[14].

Events are represented by a 4 elements tuple :

$$(x, y, t, p) \tag{2.1}$$

- x and y are the coordinates of the pixel triggering the event
- t is the timestamp (often in microseconds) at which the event occurs
- $p \in \{-1, 1\}$ is a boolean indicating the polarity of the event : $+1$ meaning an increase in brightness, and -1 a decrease

An event is triggered in a pixel when its relative change in brightness is of sufficient magnitude, which is the case when the variation exceeds a specific threshold. Mathematically, brightness is defined as the log of the photocurrent generated by a pixel.

$$L = \log(I) \tag{2.2}$$

Considering a threshold C, an event is thus generated if :

$$\Delta L(x, y, t) = pC \tag{2.3}$$

With ΔL being the difference in brightness between the last triggered event and the brightness at the current timestep.

$$\Delta L(x, t) = L(x, t) - L(x, t - \Delta t) \tag{2.4}$$

At each time and in each pixel, a brightness comparison is performed and an event is triggered if the variation exceeds the threshold. This way, there is no need to generate the whole picture all the time and the only pixels processed are the ones being subject to sufficient variations.

2.2.2 Hardware architecture

As event-based sensors heavily rely on pixel-wise access, they share similarities with CMOS sensors. However, their pixel architecture is different as it is embedding the threshold mechanism[15]. An abstract view of such a pixel can be observed in

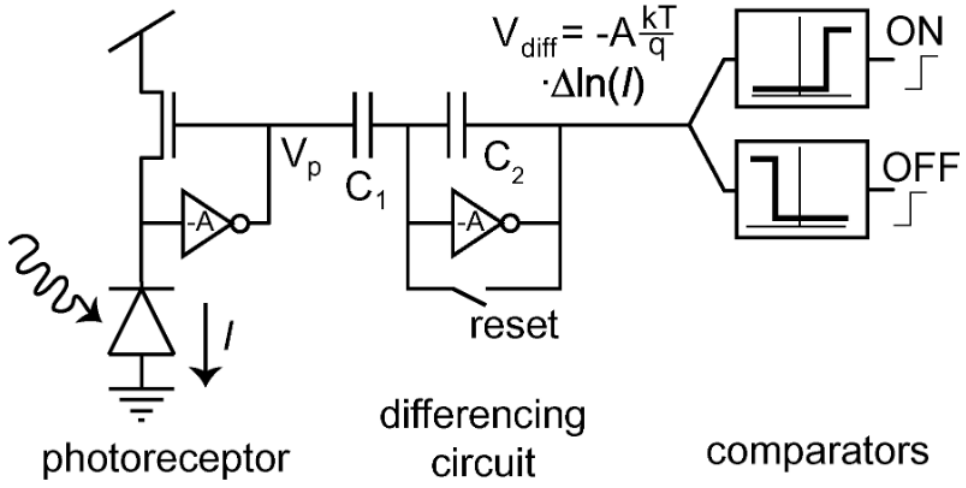

Figure 2.5: Abstracted event based pixel architecture [15]

Fig.2.5. Detailing the exact pixel circuit at a transistor level is outside the scope of this thesis, but the complete schema is left to the reader in appendix A.

The first part of the pixel is very similar to what can be found in a regular CMOS pixel as it is composed of a photoreceptor followed by an amplifier. However, this amplifier is inverted and it performs a logarithmic conversion of the photocurrent generated. With V_s being the initial voltage and V_p the output of the photoreceptor part of the pixel, we have :

$$V_p = -AV_s \tag{2.5}$$

The differencing circuit is used to amplify changes in voltage with high precision. With V_{diff} being its output current, we obtain through the conservation of electrical charges :

$$C_1 \frac{d}{dt}(V_p + \frac{V_{diff}}{A}) = C_2(\frac{V_{diff}}{A} - V_{diff}) \tag{2.6}$$

As the amplifier factor A is very high, we have :

$$C_1 \frac{d}{dt}(V_p + \frac{V_{diff}}{A}) \approx C_1 \frac{d}{dt} V_p \tag{2.7}$$

$$C_2(\frac{V_{diff}}{A} - V_{diff}) \approx -C_2 V_{diff} \tag{2.8}$$

And thus :

$$C_1 \frac{dV_p}{dt} = -C_2 \frac{dV_{dif}}{dt} \tag{2.9}$$

Which brings over a period of time :

$$\Delta V_{diff} = -\frac{C_1}{C_2} \Delta V_p \tag{2.10}$$

It is also provided with a reset mechanism in charge of forcing V_{diff} to a reference voltage after the generation of an event.

The last part takes as input the amplified signal and compares it with respect to the global thresholds which are offsets from the reset voltage. If the input signal exceeds a threshold, the corresponding event is being triggered.

A typical scenario is described in Fig.2.6.

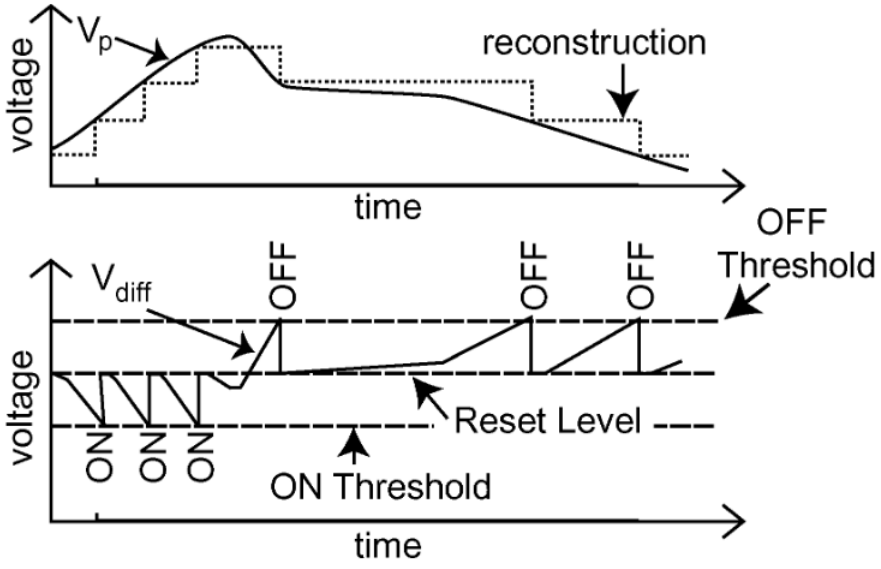

Figure 2.6: Event generation scenario [15]

The upper graph represents the evolution of the output voltage of the photoreceptor V_p while the lower one is about the output voltage of the integrating circuit V_{diff}. V_p is directly proportional to the amount of light perceived by the photoreceptor, and the variations in V_{diff} are inversely proportional to the variations in V_p thanks to 2.10.

During the whole process, V_{diff} is compared to the ON and OFF thresholds. As soon as it exceeds one of these thresholds, which is the result of a brightness variation of sufficient magnitude, an event is generated. Afterward, the reset mechanism is triggered and V_{diff} is set to its reference value.

In the end, V_p will rise and fall depending on the ongoing brightness of the pixel, while for the differencing circuit we have

$$V_{diff} \in [R - C^{ON}, R + C^{OFF}] \tag{2.11}$$

because of the reset mechanism activated after each event, with R being the reset voltage and $C^{ON,OFF}$ the threshold values.

2.2.3 Output comparison

Event-based cameras are data-driven sensors, in the sense that their output depends on the variations observed. Indeed, considering a perfectly noise-free scenario, a scene without any changes would not trigger any event. In opposition, filming a high movement scene would result in a huge amount of events generated.

In the case of a standard camera, none of this matters. It will always generate a certain number of frames per second depending on the frame rate. In the end, the same number of pixels, which depends on the resolution of the camera, is being processed by the camera.

Comparing the outputs of those 2 sensors, we have on one hand for the event-based sensor a stream of events (x, y, t, p) arriving at a high dynamic rate depending on the movements and changes of brightness in the scene. On the other hand for the frame-based sensor, images of dimension $h * w * c$ are captured synchronously depending on the integration time. Such a comparison can be observed in Fig.2.7.

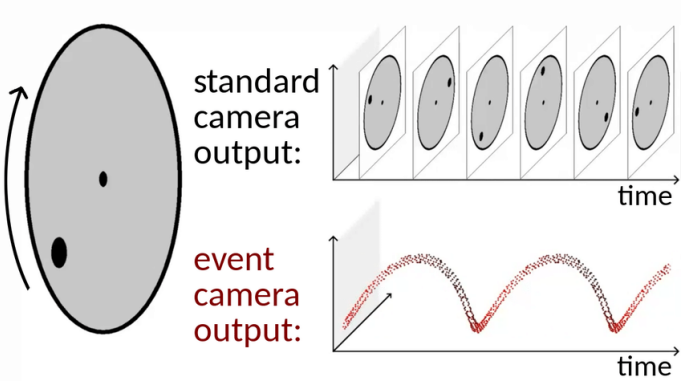

Figure 2.7: Event and frame comparison [16]

This figure illustrates the main differences in terms of output, it showcases a black dot rotating on a circle. Here, the standard camera is capturing a lot of redundancy, as the interesting part is the movement of the dot. It is also completely blind to what is happening between the captured frames. In comparison, the event-based camera is able to track the whole movement of the dot thanks to its better temporal resolution while completely neglecting the outer disk.

2.2.4 Advantages

Event-based sensors show a lot of advantages when compared with standard cameras [14]. First of all, we can mention the ones illustrated in the previous section. Indeed, event-based sensors are completely outperforming standard cameras in terms of temporal resolution, allowing them to capture very fast variations, including high speed movement which would not be fully captured if limited by a frame-based device. Typically, events are computed in microseconds, which would correspond to a camera capturing thousands of images per second.

In addition to this, they are also way more data efficient in the sense that the camera is only focused on the variations in the scene. It is thus getting rid of redundant information. As the output rate of the camera depends on the variations and not on an integration time, and since pixels are independent from each others, the amount of data processed will be considerably reduced in a low variation scenario, while it will also provide high precision with high level variations.

Another advantage is related to the dynamic range which is the ratio between the highest and the lowest brightness value a camera is able to detect. Indeed, because of their particular architecture, event-based pixels are showing a significantly higher dynamic range, allowing them to capture relevant information in extremely dark or overly bright scenes. In comparison the dynamic range of high quality frame-based sensors is around 60dB, while event-based cameras can reach dynamic ranges exceeding 120dB.

In conclusion, by processing pixels independently and focusing on brightness changes event-based cameras are able to produce a higher temporal resolution allowing to capture scenes in a more detailed way, particularly in the case of high variations such as high speed movements. They are able to do this in a data efficient manner while still being efficient in brightness saturated environments.

2.2.5 Drawbacks

Even if this technology brings a lot of benefits, it also comes with two major issues. The first one is related to its price. Indeed, event-based cameras are way more expensive than standard frame-based cameras. Not only this makes them harder to obtain, but this price difference is also translated in their resolution, as event-based camera typically have smaller resolutions than frame-based sensors[1][17].

The second counterpart that must be mentioned directly concerns the output. Events and frames being two fundamentally different kinds of data, event-based sensors bring with them brand new processing problems. More specifically, one significant challenge posed by event-based cameras is their incompatibility with traditional image processing algorithms. Indeed, conventional techniques usually manipulate frames and are unable to deal with events. It is thus necessary to develop new algorithms able to extract and exploit all the information contained in an event stream. One could argue that it would be possible to gather the events retrieved in order

to regenerate a frame using an accumulation mechanism and to apply conventional techniques. However, doing this would have little to no interest as it would cancel most of the interesting properties of the events and would certainly not be data efficient.

2.3 Applications

Event-based cameras can be a game changer technology for fundamental computer vision tasks[14] such as feature detection or tracking, allowing to capture scenes in between regular frames. Initially, objects were considered as blobs of events appearing in the same neighborhood and new events were associated to their nearest blob. This first model was however struggling to capture more complex shapes. Other methods such as iterative techniques or kernel-based strategies were thus developed to overcome these problems. For these kinds of problems, event-based sensor are particularly interesting in high speed movements scenarios or in situations during which background variations are not likely to happen, such as video surveillance[18] for example.

These sensors could also be attractive in more targeted applications. We could mention optical flow estimation which is about computing the velocity of different objects, as they could measure very high speed flows taking advantage of their asynchronous and temporal properties.

Neuromorphic cameras have also been used in medical applications. Not only they provide convenient properties that can be used for example in high-speed particle tracking[19], they also have been integrated into medical devices taking advantage of their bio-inspired properties[20].

As being high quality sensors, robotics is also one of their main field of application. The low latency provided can be very efficient in SLAM for example, in order to have very frequent update concerning the state of the world. Overall, the temporal resolution as well as the lower power consumption and the high dynamic provided by those sensors could drastically improve the quality of different applications, especially when having to deal with high speed behaviors. For example, self-driving cars would be an example of autonomous systems taking advantage of all those properties.

Now that we have introduced the event-based technology, the following chapters will be focused on the implementation of our driver and the results we obtained.

Chapter 3

Interface implementation

This chapter will be focused on the driver we have developed for the iniVation cameras. In particular, we will first present the main building blocks of our driver, being the Robot Operating System and the dv-processing library from iniVation. The former is used to create a package applied in robotics, while the latter is there to manipulate the camera.

Once we have introduced these libraries, we will detail our implementation, starting from ROS1 and moving to ROS2[21].

3.1 Robot Operating System

The Robot Operating System (ROS)[3] is *"a set of software libraries and tools that help you build robot applications"*[22]. In other words, it is an open source C++ and python library providing tools allowing the design and the implementation of robotic applications.

We decided to work with ROS as it is the most popular framework when it comes to robotics. Being fairly used in both the robotics industry and as an academic tool, it provides a standardized communication interface for robotic projects. In addition to this, it is already integrated within different software, and a lot of robots are provided with their own ROS support. In our case, ROS packages are available for TurtleBots, and we will use Gazebo as simulation software which enables a ROS interface.

In practice, ROS works using a message passing system. A ROS package is typically made of a set of nodes, topics and messages. Everything is provided by the library to create, manipulate and combine these concepts.

3.1.1 Nodes

Nodes in a ROS package correspond to an active piece of software taking care of one particular job in a robotic application. They are able to communicate together by sending messages through topics. For example, one sensor is managed by a node, which will send its collected data to another node managing a robot component which will take a decision depending on what it has received.

This architecture brings several advantages. First of all, from a modularity point of view. Indeed, as each node corresponds to a process managing one part of the robot, it is thus possible to break the whole robot behavior into smaller components. Each of them taking care of one particular task, this makes it easier to modify a specific part of the robot without modifying the others. In addition to this, nodes are portable and can be moved from one pipeline to another very easily in order to share components between different robots.

Following the same idea, it also offers the possibility to make the different nodes perform their task concurrently. The different computations can be managed asynchronously by running nodes in parallel, allowing fast and efficient processing within the pipeline.

3.1.2 Topics

A topic acts as a mailbox connecting several nodes with a FIFO queue of defined length. Each topic is bound to a specific message type it is able to manage. A node can either publish or subscribe to a topic. In the case where a topic is full and still receives messages, it will drop the older messages to leave a slot for the newer ones. A topic can have multiple publishers and subscribers. When several nodes subscribe to the same topic, each subscriber receives every message sent to that topic.

In practice, topics also work as an abstraction layer. Indeed, all nodes are publishing/subscribing to topics, while having no idea about which nodes they are communicating with.

3.1.3 Messages

Messages are particular data structures containing the different fields needed to represent the information transmitted between different nodes. They are defined by .msg files, and ROS is taking care of translating those files into source code allowing the manipulation of such messages.

Fields can be composed of standard built-in type such as int or float for example, but they can also contain more complex data such as arrays or even other messages.

3.2 Dv-processing library

The second building block of our driver is the dv-processing library[23] from iniVation. It offers a C++ or Python API with convenient data structures and algorithms for the manipulation of the cameras. We will use it mainly to retrieve and store the data from the camera relying on 2 data structures : the CameraCapture and the EventStore.

A CameraCapture is a class that can be used to manage the camera. We can use it to find any iniVation camera plugged in, modify its characteristics and read its collected data.

An EventStore is another class which is used this time as a way to store the events retrieved from the camera. Each EventStore contains a set of several events. Similarly to the event definition, we can find among its fields :

- A one dimensional array representing the timestamps of the events, computed as integers
- A 2xN array representing the coordinates x and y of the N events stored
- A one dimensional array representing the polarities of the events

It is subject to one main constraint, being that all the events must be stored monotonically with respect to their timestamps in the same EventStore, in order to capture the chronology of the scene. In addition to this, it is also provided with a set of methods used to manage and modify its content.

3.3 ROS1 package

Even if our goal is to develop a ROS2 driver for the cameras, we first decided to work with ROS1 using the Noetic distribution[24] for 2 main reasons. The first one is that since ROS2 is significantly more recent than its predecessor, it is occasionally perceived as less stable and also lacks the extensive documentation available for ROS1.

The second reason is that even if iniVation lacks a ROS2 package for their cameras, they however developed a ROS1 package allowing the integration of their cameras within ROS1 projects. These packages are available on their gitlab repository[25].

Our methodology for the implementation was the following. We initially developped our driver under ROS1 Noetic in order to first get a better grasp with both ROS and the dv-processing library, while trying to reproduce results of the same quality as what iniVation provided in their package. Once we developed a fully functional ROS1 driver, our goal was to port it to ROS2 and to assess its quality.

3.3.1 C++ vs Python

Before diving into the implementation, there is a few words to say regarding the choice of the programming language. Indeed, as both ROS and the dv-processing offer a C++ and a Python interface, we had to choose between them. We initially went for a Python implementation. While there was no problem collecting the events and visualizing them in the same node, we faced some serious performance issues when sending data from one node to another.

To send the events from one node to another, we need to convert them into ROS messages by looping over all of them. As millions of events can be generated per second, relying on Python becomes very quickly inefficient and leads to important latency.

In order to llustrate the latency problems, we studied the evolution of the time elapsed between the capture of an EventStore in a first node, and its reception in a second node over 2 scenarios of a one minute duration. The first scenario is a static scenario with as little events generated as possible, while the second is a regular one with more variations. The graphs corresponding to those scenario are accessible respectively in Fig.3.1 and Fig.3.2. Even if the number of batches is similar, they completely differ in terms of size as the motion captured generated way more events than the static scenario.

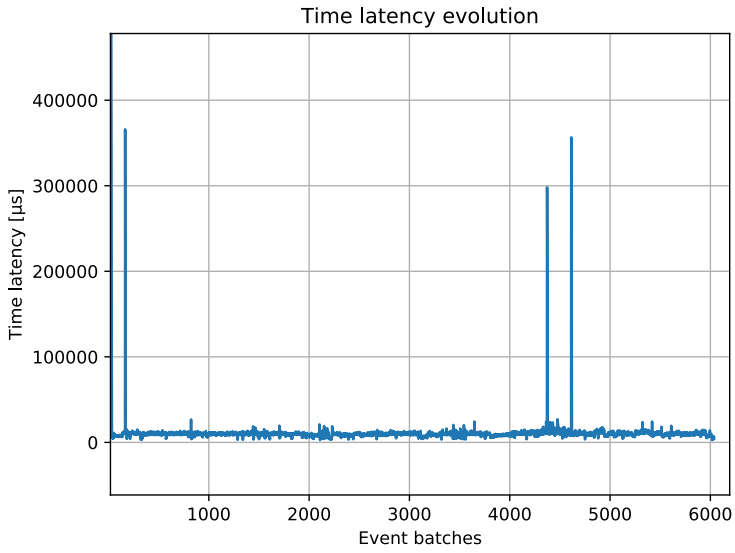

Figure 3.1: Static scenario processing latency using Python

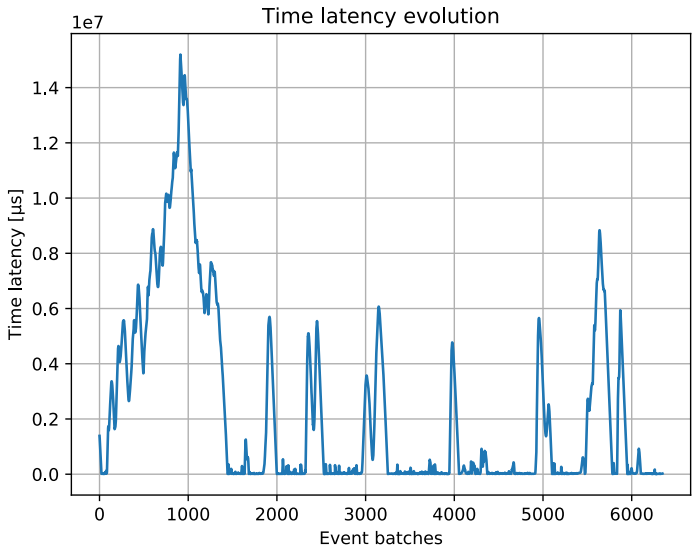

Figure 3.2: Regular scenario processing latency using Python

In the first graph, we can see that even if a few spikes are sometimes generated, mainly because of electrical noise or intern ROS/camera factors, the overall latency remains very low. However, in the second situation, we can see that spikes of bigger magnitude appear way more frequently. Each spike is this time generated by an object moving in front of the camera. As the number of events triggered directly depends on the variations in the scene, the processing time is being increased accordingly. In the end, we obtain a very low latency of a few milliseconds in a static scenario, but this value can be increased up to several seconds depending on the quantity of events captured. This would make the driver completely unusable in a real case scenario as the information would be outdated.

In order to reduce that latency, we switched from Python to C++, as the latter is well known for outperforming the former in terms of execution speed. By doing so, we fixed the latency problem and managed to communicate up-to-date data between nodes. In the end, C++ was much more suitable when dealing with high data rate sensors as we need the highest processing speed possible to manage the events.

3.3.2 Package architecture

We first have to define what we expect from our driver. From a ROS perspective, we want to be able to retrieve the data from the camera in one node, and to send them to another node which will manage their processing. In addition to this, we also want to display those data so that any user would be able to visually interpret them.

In practice, cameras from iniVation have 4 different output data streams :

- An event stream managing the events perceived by the camera
- A frame stream, as one of their model is able to capture both frames and events
- An IMU stream, as their cameras are provided with an inertial measurement unit
- A trigger stream which can be used as a synchronization mechanism with external devices

What we want is first of all a node able to retrieve data from those streams and to publish them into corresponding topics, and secondly, nodes subscribing to these topics and processing a visual representation of the data.

As the trigger and IMU streams do not need any visualization, we will only implement visualization nodes for frames and events. Finally, the architecture of the driver can be observed in Fig.3.3. Nodes are represented by ellipses while topics are represented with rectangles. We managed to obtain very good results using this architecture in ROS Noetic.

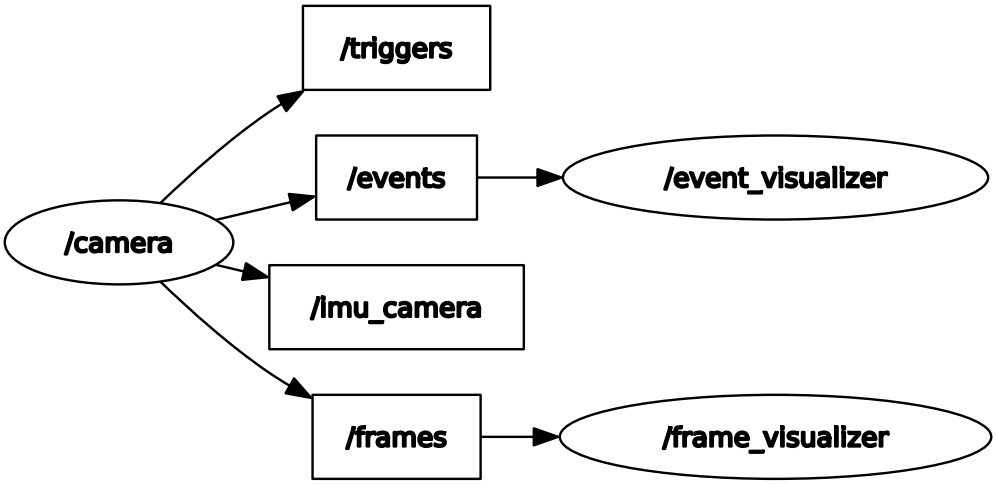

Figure 3.3: Driver ROS architecture

3.3.3 Camera node

The first node is the one managing the camera. Everything needed for the instanciation of a node is provided by ROS through a NodeHandle object. Once the node is initiated, we use it to listen to and discover any iniVation camera plugged into the computer and create its corresponding CameraCapture object.

Following this, we initiate the publishers that will be used to write data to topics. Each publisher needs 3 parameters to be initialized : a message type indicating which data structure it will be in charge of, an integer indicating the size of its

queue and a string representing the name of the topic. We have defined the queue size to 1000 to ensure no data are being lost in the pipeline.

Once everything is initialized properly, we have to implement the behavior of the node. In ROS1, this behavior will be contained within a specific loop, that will be running over and over as the node is spinning. ROS has mechanisms allowing us to select and tune the loop rate, defining the number of iterations per second. In the case of a node subscribing to a topic, the behavior is relying on a callback mechanism. The behavior is contained within a function named the "callback function" which will be called every time the node receives a message from the topic.

We decided to choose a loop rate of 100 which is equal to one iteration every 10ms. The reason behind this value is purely empirical, as this is the highest rate at which event batches were retrieved.

As long as the node is spinning and the camera is still plugged in, the node retrieves data from the available streams. It will then convert them into convenient ROS messages before publishing them to the corresponding topics.

The first stream being managed is the event stream. At each iteration of the loop, we retrieve the last events captured by the camera. In the implementation, events are recovered as batches in the form of EventStores. As the camera is capturing events, they are stored within the device until being read. As soon as it is done, we retrieve the batch containing the events, and the camera is being flushed. The next time we read data from this camera, we will retrieve the events generated starting from the previous reading time.

Now that we are able to recover the events of the camera, we have to publish them to the corresponding topic. To do so, we have to convert them into ROS messages. There exists a lot of built-in ROS messages, however, none of them was perfectly suitable to manage events. We thus designed a tailored message named EventBatch used to store the events read. It contains 6 different fields :

- ts : An array containing the timestamp of each event in the batch
- x : An array containing the position on the x axis of each event in the batch
- y : An array containing the position on the y axis of each event in the batch
- polarity : An array containing the polarity of each event in the batch
- size : An integer indicating the number of events in the batch
- resolution : A 2 elements array representing the resolution of the camera capturing the events

Each EventStore recovered from the camera is thus being converted into an EventBatch ROS message, which is then being published to the "events" topic.

We initially planned to design a message representing a single event, but we eventually kept the batch architecture for 2 main reasons. The first reason was to avoid

flooding the message traffic. Indeed, as EventBatches are typically composed of thousands or several thousands of events, clustering the events as batches would considerably reduce the number of messages created and published to the topic. The second reason is because it preserves the original architecture of the EventStore obtained when reading data from the camera.

The second stream this node is taking care of is the IMU stream. An Inertial Measurement Unit is defined as *"an electronic device that measures and reports acceleration, orientation, angular rates, and other gravitational forces. It is composed of 3 accelerometers, 3 gyroscopes, and depending on the heading requirement, 3 magnetometers."*[26]. It is mainly used to keep track of the orientation and position of the device.

Data from the IMU are represented as an IMU object with fields containing the linear accelerations and the angular velocities. Similarly to the event stream, IMU are retrieved as batches. However, we will not rely on a custom message sending them by batch, as we will use the IMU message from the sensor_msgs library[27]. We assumed that relying on existing messages from popular ROS packages instead of defining custom ones would make our package more easily portable toward other projects. The IMU message contains the following data :

- The orientation represented by a quaternion message from the geometry_msgs library[28] and its corresponding covariance matrix
- The angular velocities as a 3 elements vector and the corresponding covariance matrix
- The linear accelerations as a 3 elements vector and the corresponding covariance matrix

However, the IMUs from the iniVation cameras are only able to collect angular velocities and linear accelerations. At each loop iteration, the corresponding fields of the IMU messages are being completed with IMU data from the camera, while the other fields are being filled with 0 or -1. Afterward, the message is published to the "Imu_camera" topic.

The trigger stream is used to retrieve batches of Trigger objects containing the timestamp and the type of trigger captured in the form of an integer. We defined a custom message representing a batch of triggers containing 2 fields : a timestamp array indicating the timer at which triggers have occurred and an integer array containing the trigger types.

The last data stream being managed is the frame stream. Depending on the frame rate of the camera, each time a frame is available, it is retrieved as a Frame object from the dv-processing library, which contains the image as an opencv matrix. We then convert this matrix into an Image message from the sensor_msgs library, using the cv_bridge library[29] which offers efficient and convenient conversion mechanisms between ROS and opencv. This message is then published to the "frame"

topic.

3.3.4 Event visualizer

As mentioned earlier, the following nodes will rely on a callback mechanism. Every time a message is available in the topic the node is subscribing to, it will be read and processed by the node.

In the case of our event visualizer node, the EventBatches are being retrieved from the event topic and reprocessed into EventStores. The visualization is done through a slicing and accumulation mechanism. Indeed, as events represent the variations of brightness in the scene in the form of tuples, we have to process them in order to obtain a visual representation that can be interpreted by anyone.

The main idea here is to generate a frame representing a subset of events. To do so, pixels where events are being triggered are colored depending on the polarity of the event. In order to have a sufficient number of events to generate a meaningful frame, we perform what is called a slicing. Events retrieved are clustered into smaller groups, and each groups is used to generate a frame.

Slicing can either be performed on a temporal basis, considering the events generated over a defined time window, or considering a fixed number of events. Both slicing mechanisms are represented in Fig.3.4 and Fig.3.5. The dv-processing library offers convenient slicing and visualization mechanisms. We decided to go for a temporal slicing since it provides a better visualization over time, and we used a time window of 33ms as it is the default parameter and provides good results.

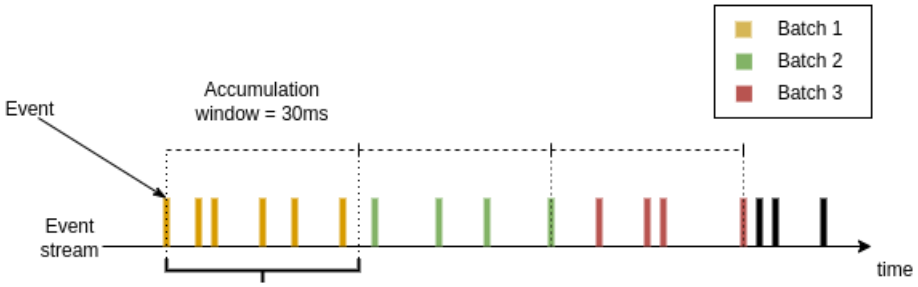

Figure 3.4: Time-based slicing [23]

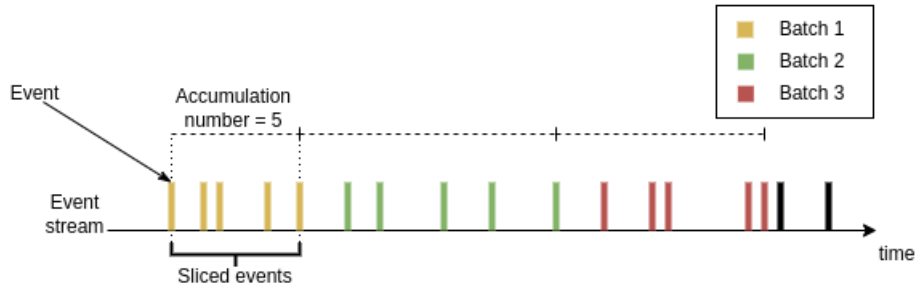

Figure 3.5: Number-based slicing [23]

Once the subset is defined, the frame is generated with respect to the polarities of the events. The polarities of the events are summed with respect to the position at wich they occur. In the end, a color is attributed to the pixel depending on the result. The color of the background is initially set to white. If the sum is positive, this means that brightness at this location has increased, and the color blue is given to that pixel. In opposition, if the sum is negative, this means the brightness has decreased and the color grey is attributed to the pixel. An example of generated frame can be observed in Fig.3.6.

Figure 3.6: Events visualization

3.3.5 Frame visualizer

There is not much to say regarding the last node. Here again, when an Image message is available in the frame topic, we read it and process it within the callback

function. We rely on the cv_bridge library to convert the Image message back to an opencv image which is finally displayed.

3.4 ROS2 package

3.4.1 Distribution choice

The first thing we had to do while porting our driver to ROS2 was to choose the distribution we were going to work with. We initially planned to work with ROS2 Foxy Fitzroy[30]. Even if it is has reached its end-of-life, we wanted to use it since some TurtleBot3 support is provided with this distribution, and because it is the distribution installed on the TurtleBots of the university.

However, this choice of distribution made us face compilation issues. Indeed, the dv-processing heavily relies on C++20 features, making it impossible to compile with lower versions. In opposition to this, the implementation of ROS2 Foxy makes use of std::allocator::rebind which is deprecated in C++17 and completely removed in C++20. The combination of those 2 implementation choices results in an incompatibility between both libraries and brings compilation errors. We initially tried to modify the implementation of the distribution in order to replace rebind appearances with compatible code, but the amount of modifications was way too high and led to inconsistencies.

The solution we found to this problem was to change the distribution and use ROS2 Galactic Geochelone[31] instead. Without any modifications, this distribution still makes use of rebind. However, it is only used once in the whole implementation. Following a github issue[32], we only had to modify the line containing rebind in order to obtain a C++20 friendly ROS2 distribution. This way we managed to compile a ROS2 package using the dv-processing library.

We could also have used ROS2 humble[33], being the latest long term support distribution, as it is C++20 compatible without any modifications. However, ROS2 Humble relies on Ubuntu 22.04 while ROS2 Galactic relies on Ubuntu 20.04. Since the TurtleBots from the university are installed using an Ubuntu 20.04 server, we decided to go for ROS2 Galactic and to stick with Ubuntu 20.04 instead of modifying the Ubuntu server.

3.4.2 Porting from ROS1 to ROS2

From an algorithm point of view, things are very similar in ROS1 and ROS2. Indeed, everything related to the data acquisition, its processing and the different messages used to transmit the information remains the same. The main differences lie in the code architecture, especially at a ROS level.

Using ROS1, there was no conventional way of implementing nodes. Any C++ code using the ROS library could initiate a node thanks to the init function and manage it

through a NodeHandle object. The different topics are then being declared afterward so that the node is able to communicate its messages.

In ROS2, things are slightly different. The implementation of a ROS node now heavily relies on object oriented programming. In order to create a new node, we have to define a brand new class inheriting from the Node class of the rclcpp library (the equivalent of ros.h for ROS2). In addition to the fields used for the behavior of the node, this class should also contain all the publisher and subscriber objects needed for its communication mechanisms.

Another difference in ROS2 is that every node, even the ones not subscribing to any topics, now rely on a callback function which is also defined as a private field. In the end, the main function of a file is just used to create the node and to make it spin. In ROS2, our implementation is thus relying on 3 objects managing the different nodes. In particular, our Camera class is composed of the following fields:

- A timer callback function retrieving, processing and publishing data
- A CameraCapture object from the dv-processing library
- 2 integers x and y representing the resolution of the camera
- A publisher managing EventBatch messages
- A publisher managing IMU messages
- A publisher managing Image messages
- A publisher managing Trigger messages
- A TimerBase object defining the loop rate of the callback function

In practice, the implementation of the callback function is very similar to the spinning loop in our ROS1 package. We do not detail the event and frame visualizers classes here as they mainly contain Subscription objects retrieving data from their corresponding topics.

Now that we have implemented a functional driver, the next step will be to assess its quality and to use it within a complete ROS2 pipeline.

Chapter 4

Experimental setup

In this chapter, we will cover the complete setup in which we experimented our package. More precisely, we will present our computing capabilities, the different cameras at our disposal and we will detail the characteristics of the TurtleBot used.

Following this, we will also discuss the different test environments we used, as we first manipulated the robot in simulation before moving to physical tests.

4.1 Computer and cameras

As the driver has been developed and tested in a first time completely independently from the TurtleBot, we believed that it was important to mention the hardware on which we tested our implementation as it would provide a better interpretation of the results obtained. This being said, the driver has been implemented and tested on an ASUS ZenBook with 8gb DDR4 RAM and an Intel® Core™ I5-825OU 1.60GHz CPU, under Ubuntu 20.04.6 LTS (Focal Fossa).

Since the implementation of our package is general enough to be used with any iniVation camera, we also take advantage of this section to introduce the different devices with which we evaluated our driver. We used 3 different models of camera : The DVXplorer[15][34], the DAVIS346[35][34] and the DVXplorer mini[36].

The DVXplorer camera is an event-based only device. It is able to transmit up to 165 millions of events per second at a resolution of 640x480 pixels. Its temporal resolution is about 65-200µs being the minimum time between two timestamps and its dynamic range reaches 110dB.

The DAVIS346 device is able to output frames and events with a resolution of 346x260 pixels. Its event rate is reaching up to 12 millions of events per second with a temporal resolution of 1µs and a dynamic range of 120dB. The lower temporal resolution of the DVXplorer device can be explained here[37], but the reason is mainly that the gain obtained using a timestamp unit of 1µs instead of 65-200µs

was very little. Concerning frames, the device is able to produce up to 40 frames per second with a dynamic range of 55dB.

The DVXplorer mini is very similar to the DVXplorer, as both models share the same spatial and temporal resolutions, respectively of 640x480 and 200µs. The main differences lie first in the event rate, as the DVXplorer mini is able to capture up to 450 millions of events per second, and secondly, in their physical characteristics as the mini device is about 65% smaller while being way lighter than the DVXplorer.

4.2 TurtleBot

The robot with which we plan to combine the camera to is a TurtleBot3 by Open robotics and Robotis. It is described as *"a small, affordable, programmable, ROS-based mobile robot for use in education, research, hobby, and product prototyping"*[2]. In other words, this makes it one of the go-to model when it comes to teaching robotics. The reason we are relying on this robot is because it is the model used in different courses of the university, and in the context of these courses, having a neuromorphic vision interface might open the gates to new opportunities.

There are two models of TurtleBot3, the "burger" one and the "waffle" one. The main differences between them concern their physical properties (size, velocity, weight, battery ...)[38]. We will work with the burger model as this is the one we have at our disposal. It has the following physical dimensions 138x178x192mm (L,W,H) and weighs 1kg. It is expressing 2 degrees of freedom, as it is able to move forward/backward at a maximum linear velocity of 0.22m/s and to rotate at a maximum angular velocity of 2.84 rad/s. It is provided with a 360° lidar that can be used for navigation and is programmed through a Raspberry Pi 4 model B rev 1.5. The TurtleBot can be observed in Fig.4.1.

One significant advantage of the TurtleBot is its built-in ROS nodes. Indeed, Robotis provides a set of open source ROS packages that can be used to manipulate the robot. Combined with its simplicity, this makes the TurtleBot a very convenient testing robot. The packages are available in different ROS1 and ROS2 distributions, including Galactic, and can be accessed here[39]. Not only these packages provide sufficient ROS support to control and update the state of the robot, they also provide algorithms for different tasks, such as SLAM or navigation, and even the modeling mechanisms needed to represent the robot in simulation.

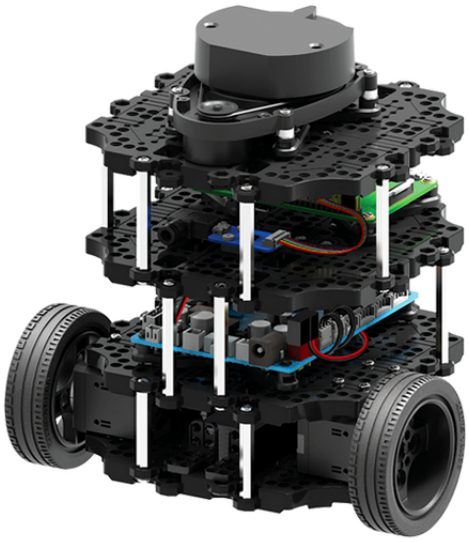

Figure 4.1: TurtleBot3 burger [38]

4.3 Simulation environment

Before testing the communication between the cameras and the robot physically, we first worked in a simulation environment. Indeed, it is of good practice in robotic projects to first deploy the whole pipeline in simulation before evaluating it directly on the real robot. This makes us free of the physical setup and constraints as everything would be materialized in software, but most importantly, it keeps us safe from damaging the material during the development period.

In order to simulate the behavior of the robot, we are going to rely on Gazebo[40], which is one of the most popular simulation frameworks in robotics. It is an open source software which allows the creation and simulation of robotic systems in a virtual environment. It provides necessary support to create a scene containing different objects and to simulate a robot able to interact with them while reproducing their physical properties.

In Gazebo, a robot is defined by its description contained in a URDF file (Unified Robot Description Format) containing a set of links interconnected by joints. A link corresponds to a specific individual part of the robot, in order to break its model into several smaller pieces. Each link is described by its physical properties (shape, mass, center of mass, position...). A joint represents the relation between 2 links. It indicates the initial distance between them as well as the motion restriction induced by their connection.

In addition to this, Gazebo is directly compatible with ROS. Indeed, it offers a complete ROS interface allowing to directly interact with the simulated environment, through the use of dedicated nodes. Using the package from Robotis, it becomes very easy to instanciate a Turtlebot and to control it in Gazebo. Relying on their

launchfile, we obtained the ROS architecture observable in Fig.4.2 to manipulate the robot in simulation.

The corresponding ROS nodes work in the following way : turtlebot3_imu and turtlebot3_laserscan collect the information respectively from the imu and the lidar of the robot and publish them to the corresponding topics. The node turtlebot3_diff_drive is reading the velocity data from the cmd_vel topic and is in charge of translating it into motor controls updating the velocity of the wheels appropriately. Joint nodes are used to keep consistency between the different joints of the robot. It is writing with the diff drive node on the tf topic. Typically, a robot usually needs several reference frames which are moving over time (sensors, hands, ..), and tf is there to keep a track of them[41]. The node Gazebo and the topic performance_metrics are used to evaluate the performance of some sensors.

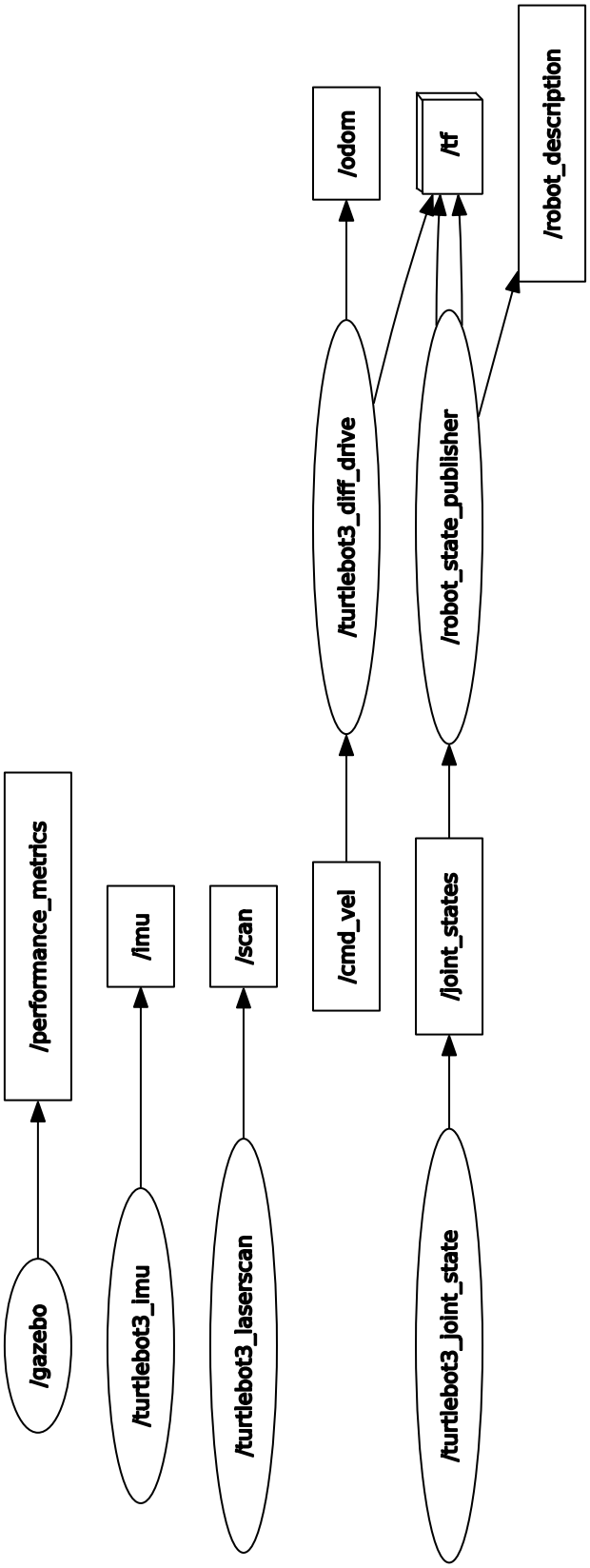

Figure 4.2: TurtleBot ROS architecture within Gazebo

4.4 Physical environment

After checking and testing everything in Gazebo, we wanted to establish a communication between the computer and one of our TurtleBot in order to observe the results with the physical TurtleBot. The connection between the Ubuntu 20.04 server running on the Raspberry Pi and the computer was established through SSH. However, we had a few things to setup before starting the SSH connection.

Indeed, in order to communicate with the computer and share its different ROS topics, the TurtleBot relies on multi-cast communication. However, multi-cast is completely disabled in the university network, making it impossible for the TurtleBot to communicate. In practice, we used a third party computer connected to the university network as an access point to obtain a dedicated network on which we connected the TurtleBot and our computer. This way, the robot had access to multi-cast communication.

Once the connection was established, we had to run a bringup launchfile from the TurtleBot ROS package directly on the Raspberry Pi. This launchfile initializes all the different nodes and topics used to manipulate the robot. The complete ROS pipeline is available in Fig.4.3.

The architecture obtained is very similar to what we have seen in Gazebo, with the most important difference being that the data from the Turtlebot are retrieved by one single main node turtlebot3_node instead of several.

Finally, to establish the communication between ROS nodes runnning on the TurtleBot and on the computer, the last step was to synchronize their respective ROS domain id.

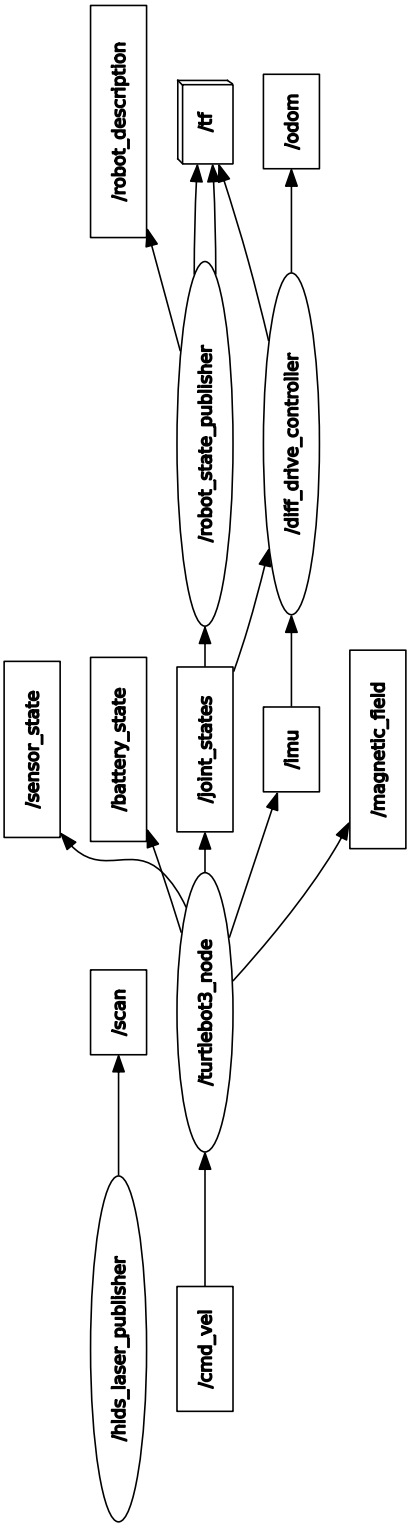

Figure 4.3: TurtleBot ROS architecture

Chapter 5

Experiments

In this section, we describe the different experiments performed in order to assess the quality of our driver. In a first time, we will evaluate the ROS package on its own by running it on a computer and observing the speed at which events are being retrieved and sent from one node to another.

In a second time, we will connect our driver to the ROS architecture of the TurtleBot and will control it based on the events captured by the camera.

Finally, in order to completely integrate the camera as a dedicated sensor, we will plug it into the TurtleBot and run our ROS package directly on the Raspberry Pi.

5.1 Latency analysis

5.1.1 Motivations

The first step in our performance assessment methodology is to evaluate the latency that can be found within our ROS package. Indeed, as retrieving data from the camera in a ROS node is what we try to achieve, the implementation becomes completely unusable if the data received by a second node is outdated considering the high data rate and temporal resolution of those cameras. We initially analyzed the latency in a qualitative way, by not only visualizing the events in the event_visualizer node, but also in the camera node. This way, we could visually observe the latency from one node to another.

From a qualitative perspective, the results obtained were pretty convincing, as we could not differentiate the frames displayed in the capture node from the frames displayed in the visualizer. However, this analysis alone was far from being sufficient as it did not provide any metric concerning the latency and could not be used to evaluate in details the different parameters that are influencing it.

To tackle this problem, we conducted a quantitative analysis providing temporal values describing the latency. To do so, we identified different causes that could

explain the time needed between the capture of an event batch in the camera node and its visualization in the second node, and we studied their impact. The first one is obviously the processing latency. Since there is quite a lot of computation time needed to generate and publish the different ROS messages in the camera node, we have to make sure that our package is able to forward those messages in a reasonable amount of time. The second one is related to the network latency. As ROS relies on message passing mechanisms between nodes, the travelling time of the different messages might heavily influence the overall latency.

5.1.2 Results

For this analysis, we studied the evolution of the processing time of the callback function within the camera node, and the evolution of the travelling time between the camera node and the event_visualizer node. For the sake of interpretability, we did not display the generated frame in a dedicated window in the visualizer as it was producing a huge latency spike at its creation that made the variations in the latency graphs almost impossible to observe. The results described in this section have been generated with the Davis camera.

As the quantity of data processed directly depends on the number of events generated, we benchmarked our driver over 3 scenarios with different amount of events generated. The first one is a completely motionless scenario of 1 minute used to evaluate the lowest amount of latency we can expect running the driver. The size of each batch retrieved during the record is displayed in Fig.5.1. On the following graph, you can observe the different batches on the x axis and their corresponding size on the y axis.

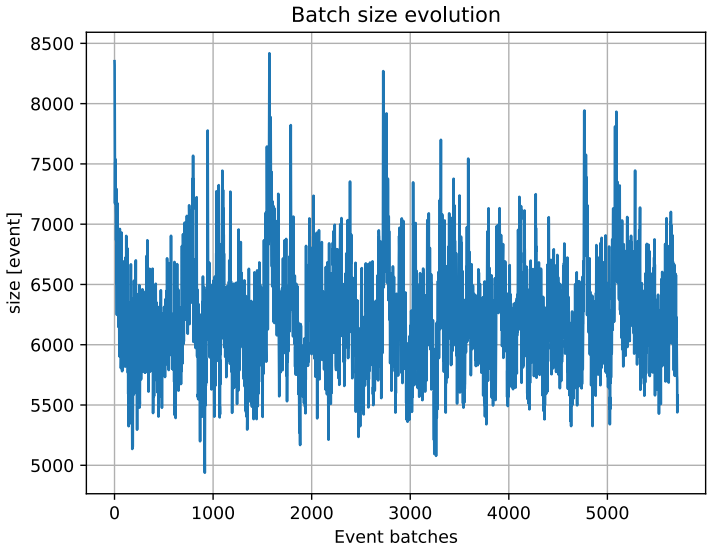

Figure 5.1: Evolution of the batch size over 1 minute in a motionless scenario

In this experiment, since we tried to generate as little events as possible, the average batch size is pretty low as it is of 6209.58 events. The latency related to the processing of these events can be observed in Fig.5.2 while the network latency can be observed in Fig.5.3.

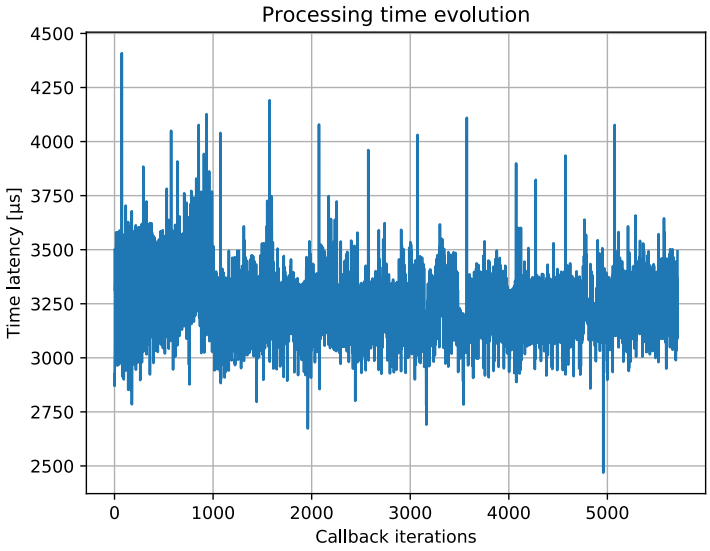

Figure 5.2: Evolution of the processing latency over 1 minute in a motionless scenario

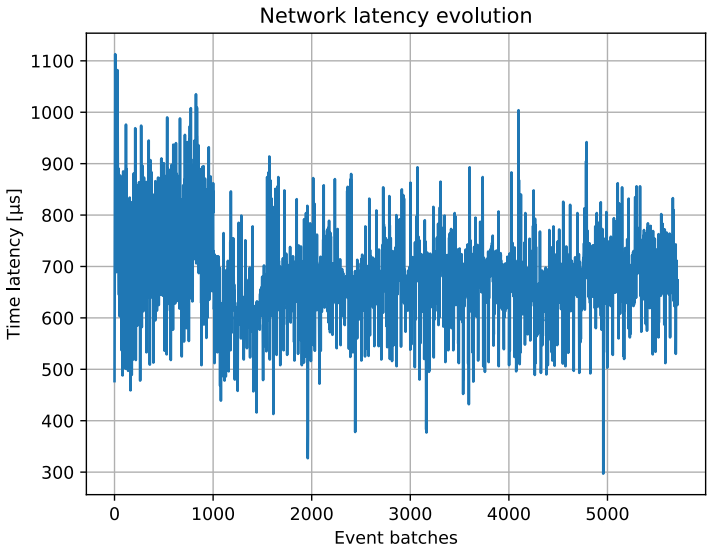

Figure 5.3: Evolution of the network latency over 1 minute in a motionless scenario

On the former, the callback function iterations are available on the x axis, and their respective execution time is available on the y axis. On the latter, you can observe all the batches generated on the x axis, and their corresponding travelling time to go from one node to the other on the y axis.

As we can see, the results obtained in this situation are pretty good as the network latency is on average under 1ms, while the processing latency remains most of the time between 3 and 3.5ms, which is what should be expected in a static situation.

The second scenario is a more generic one of a 2 minutes duration. This time, the events captured represent the movement of a person in the background. In addition to this and to observe the impact of a sudden burst of events, we decided to shake the camera after the first half of the record. The different batches as well as their corresponding size can be observed in Fig.5.4.

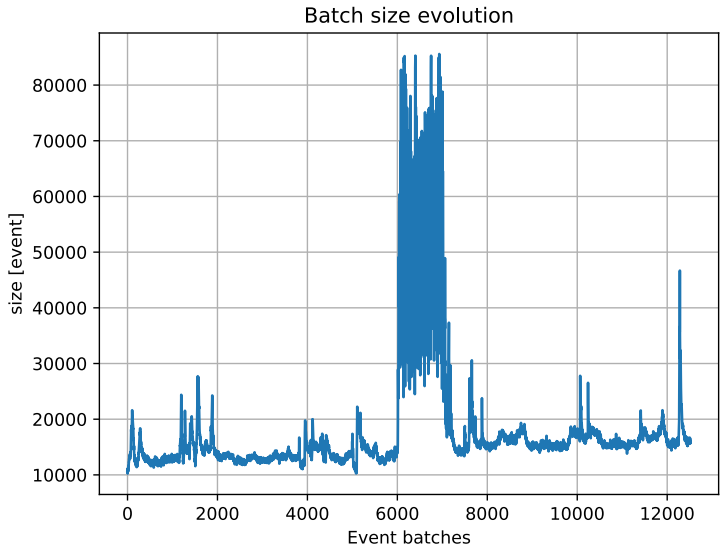

Figure 5.4: Evolution of the batch size over 2 minutes in a regular scenario

In this case, we can see that the number of events generated has drastically increased as every single batch now contains more than 10 000 events, while the biggest batch retrieved in the previous situation contains 8418 events. The spikes generated after 6000 batches correspond to the burst of events generated by shaking the camera. In this situation, batches contain up to 85586 events. The latency graphs corresponding to this experiment are available in Fig.5.5 and Fig.5.6.

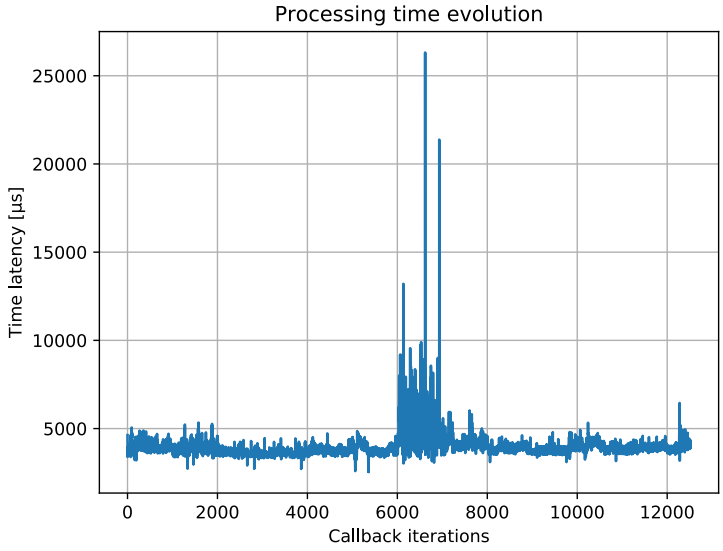

Figure 5.5: Evolution of the processing latency over 2 minutes in a regular scenario

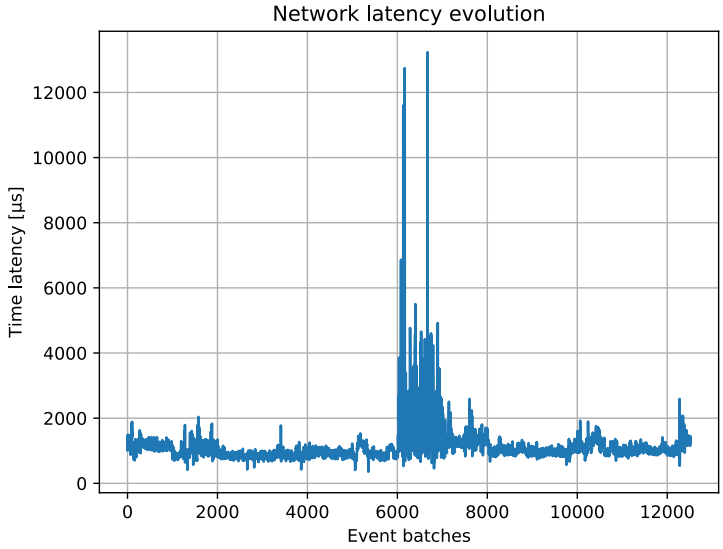

Figure 5.6: Evolution of the network latency over 2 minutes in a regular scenario

Looking at those graphs, we can directly see the impact of the batch size on both the processing time and the travelling time as they share the same kind of shape. However, it has to be noted that the processing latency is more than twice the value of the network latency during the whole experiment as the former mainly remains around 5ms while the later stays around 1ms. For both of them, the latency observed

is rather small in a first time, and increases after 6000 batches in reaction to the burst of events. Nevertheless, the amount of latency generated at this moment still remains acceptable, and most importantly, the data communicated by the driver are still up-to-date.

Finally, our last experiment consists in a worst case scenario in which we decided to shake the camera for a complete minute in order to have a huge amount of events generated during a longer period of time. The different graphs corresponding to this experiment are available in Fig.5.7, Fig5.8 and Fig.5.9.

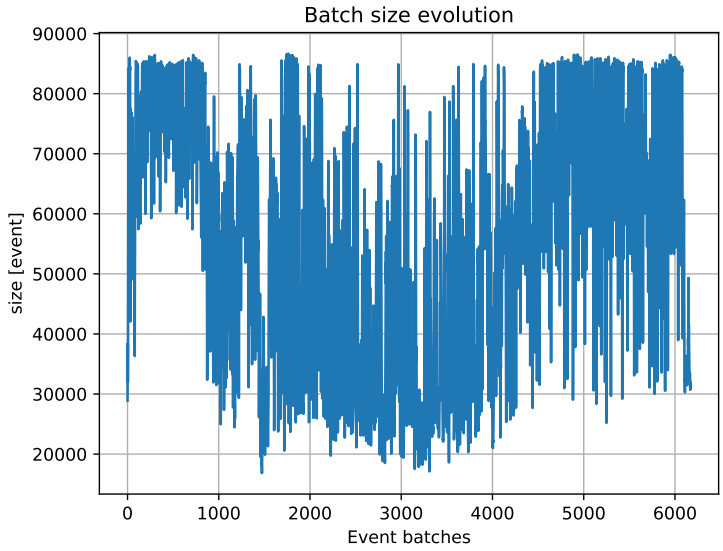

Figure 5.7: Evolution of the batch size over 1 minute in a worst case scenario

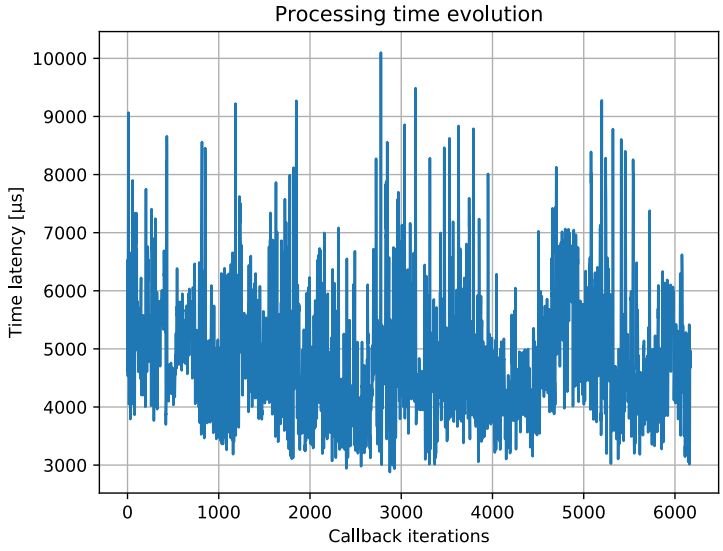

Figure 5.8: Evolution of the processing latency over 1 minute in a worst case scenario

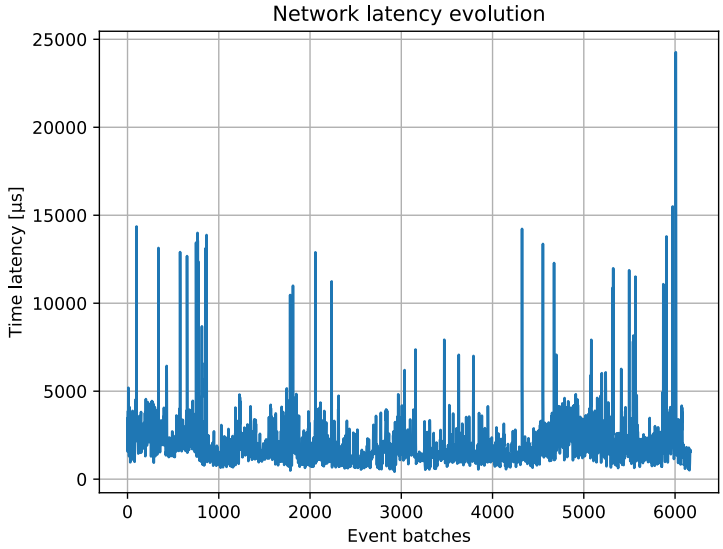

Figure 5.9: Evolution of the network latency over 1 minute in a worst case scenario

In this situation, the batch size is in average way higher than what we have experienced before. The network latency remains most of the time under 5ms with a few spikes over 10ms. Concerning the processing latency, it is on average way higher than what we have seen so far, but still remains low enough as only one iteration exceeds the 10ms threshold. Overall, even in a worst case scenario, our driver is still

able to communicate the events in a sufficiently short amount of time. This worst case scenario also makes us consider the 2 spikes over 20ms in Fig.5.4 as outliers generated by internal ROS, camera or computer factors as we never met such values again even in this situation.

We can notice that over our 3 scenarios, the processing latency is almost all the time higher than the network latency. The first reason for this is that the processing done by the callback function manages all the data streams and not only the event one, while the network latency was computed as the time elapsed between the moment when the EventBatch ROS message was send on the corresponding topic in the camera node and the moment where it was read by the visualizer. The second reason is that since we are running both nodes on the same device, the connection between them is done through the internal network of the computer. We assume the network latency component to be heavier in a multi-device communication scenario.

A summary of the results obtained with these 3 scenarios is available in Table5.1.

	Min size	Max size	Average size	Average process latency	Average network latency
Motionless	4938	8418	6209.58	3225.11µs	686.22µs
Regular	10250	85586	18702.78	3995.26µs	1120.26µs
Worst case	16839	86678	57018.24	4857.45µs	1960.1µs

Table 5.1: Latency analysis results

We also performed the same analysis with the DVXplorer camera considering the differences between the 2 devices. The corresponding results are available in appendix B. The main distinctions are that the DVXplorer is less sensitive to noise compared to the Davis camera thus capturing less events in regular or low variation scenarios, but since its event rate is way higher, it is able to capture almost twice the amount of events captured by the Davis device. In addition to this, despite the bigger amount of events generated in the worst case scenario or during the burst phase of the regular one, the DVXplorer device led to less latency, both from a processing or a networking point of view. We asssumed that this difference was mainly caused by the frame stream of the Davis device. As the Davis camera has to manage regular frames as well, we assumed that it was faster to read data from the DVXplorer which only produces one of those streams. The driver also does not have to manage the frame to ROS message conversion.

5.2 Event-based motion control

5.2.1 Motivations

Now that we have studied the latency of our package and showed that the data could be communicated in a reasonable amount of time, we wanted to use it in

combination with our TurtleBot. Our objective was to establish a communication between the camera plugged in a computer and the robot, first in simulation then with the physical robot, and to integrate the data streams from the camera within the decision process of the TurtleBot. Doing so, we believe to show that first of all, our driver can be manipulated in a plug and play manner from a ROS perspective by integrating it in an already existing ROS2 pipeline, and secondly, to show that it could be reliable when being paired with our TurtleBot in a real time scenario. As a mean to illustrate these results, we implemented a way to control the robot through the events captured by the camera.

5.2.2 Task definition

The driving mechanism we have designed is rather simple, but heavily takes advantage of the event-based nature of the output. To control the TurtleBot, we divided the resolution of the camera into 9 different bins in the same way as in Fig.5.10.

1	2	3
4	5	6
7	8	9

Figure 5.10: Event bins division

Whenever a batch is received, we compute the number of events generated in each bin. Afterward, a movement decision is triggered depending on which bin contains the biggest amount of events. As the TurtleBot is able to move forward/backward and to rotate left or right, our goal was to make it move forward/backward if events were triggered on the top/bottom of the sensor, and to make it turn left or right if they were triggered on the sides. More specifically, we used the following movement orders :

1. Move forward and turn left
2. Move forward

3. Move forward and turn right
4. Turn left
5. Stay in place
6. Turn right
7. Move backward and turn left
8. Move backward
9. Move backward and turn right

We implemented this control interface through another node called "control" reading data from the events topic and publishing to the cmd_vel topic. The velocity is being defined as a Twist message from the geometry_msgs library[28] which contains 2 vectors of 3 elements : the linear and angular velocities over each dimension. Being limited by the 2 degrees of freedom of the TurtleBot, we will only modify the linear velocity over its x axis, and its angular velocity around its z axis.

5.2.3 Simulation results

The ROS pipeline obtained with this experiment can be observed in Fig.5.11 with each node running on the computer. In Gazebo, we managed to successfuly drive the TurtleBot, but we initially faced 2 issues.

The first problem encountered was related to the amount of "unwanted" events generated by the camera. Indeed, event-based cameras are often prone to noise, either due to electrical perturbations within the device, or because of tiny brightness variations that we did not want to consider caused by light's instability for example. The combination of both of these reasons were creating a bias toward the bin in the top left corner.

In order to fix this problem, we tried to reduce the number of unwanted events by running the experiment in a more stable light environment, but we also added a threshold concerning the number of events needed in a bin to trigger an action. In the case were the number of events in each bin was too low, we triggered the action corresponding to the 5th bin to stop the movement of the robot. The threshold value was chosen empirically, based on the noise level observed during the motionless scenario performed in the previous section.

The second problem was caused by a lack of processing power. While running our package and Gazebo at the same time, we noticed very small processing difficulties introducing a tiny amount amount of latency both in the events retrieved and in the update of the state of the robot. We assumed the problem was being caused by a CPU overload, as it had to manage both the event processing and the simulation environment, with Gazebo requiring more than 80% of the CPU work time. In the end, we still managed to control the robot very accurately despite the latency, and

we managed to completely fix the latency problem by running the driver without the visualizer. However, we expect the results to be latency free with the physical TurtleBot under the assumption of a sufficiently good network as the ROS nodes would be spread over the computer and the Raspberry Pi, while not having to manage the simulation mechanisms.

5.2.4 Physical results

Trying to control the physical robot, we obtained the ROS pipeline available in Fig.5.12. As our application do not rely on the lidar, we decided to remove it during our tests. It has to be noted that the camera, control and event_visualizers nodes from our package have been launched on the computer, while the rest of the nodes which are the ones managing the TurtleBot are running on the Raspberry Pi.

As expected from our simulation tests, we managed to control the robot without any latency. Even if getting rid of Gazebo and moving to the physical robot should have increased the message travelling time as the nodes are now communicating in Wi-Fi instead of communicating within the same device, the velocity messages were still received in time and the robot was able to move in a latency free manner. We thus managed to have a reliable communication between the camera and the TurtleBot.

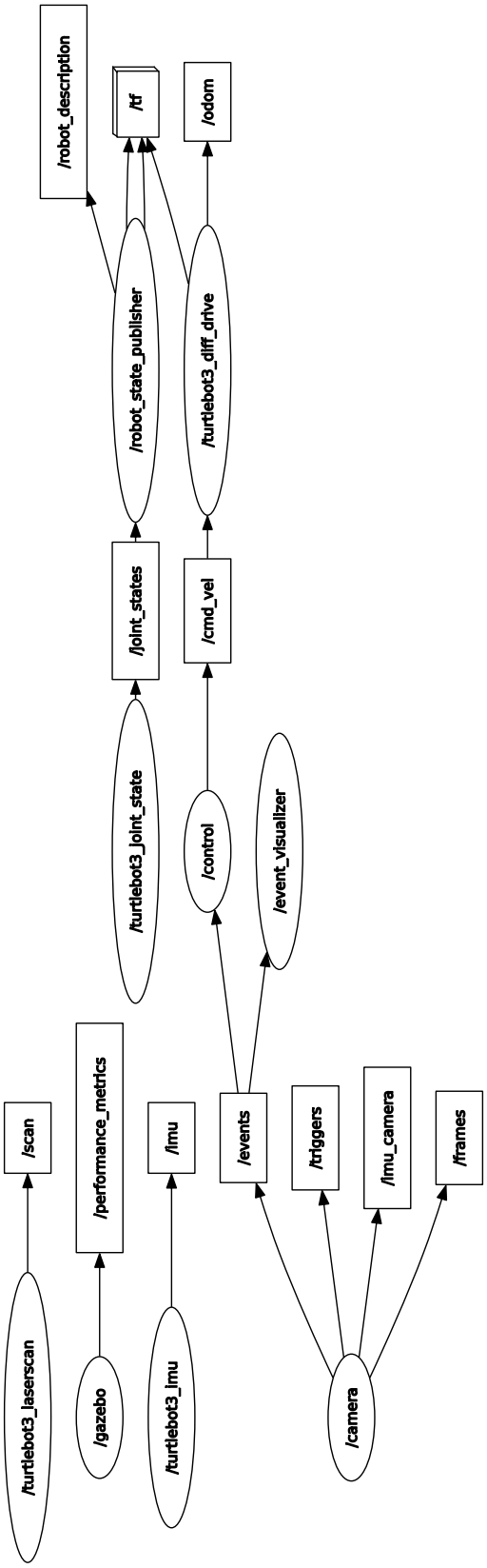

Figure 5.11: Event-based motion control ROS architecture in Gazebo

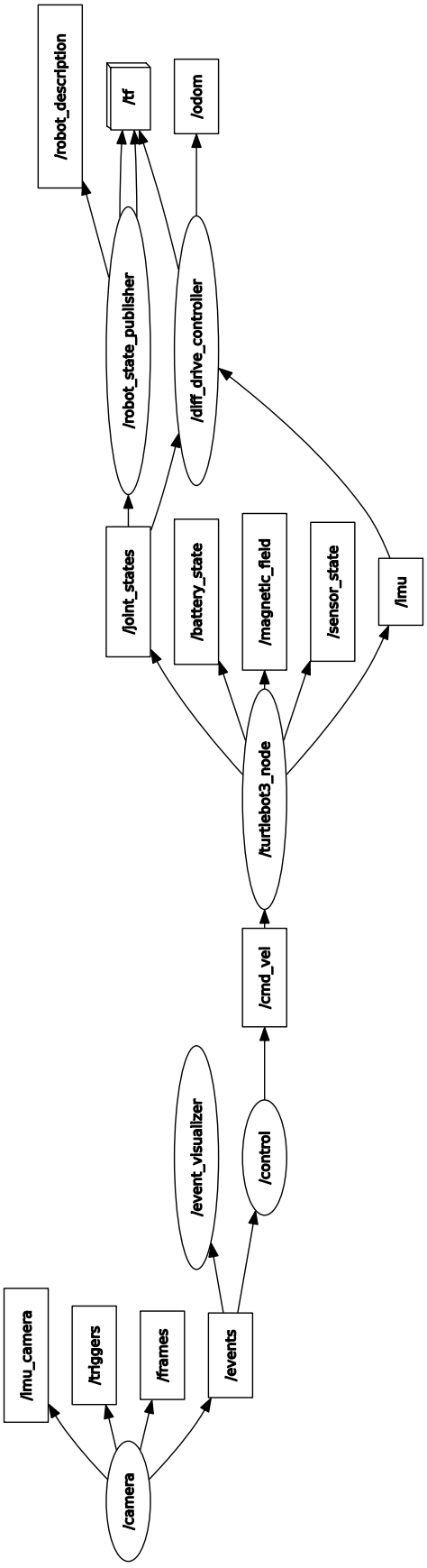

Figure 5.12: Event-based motion control ROS architecture

5.3 Integration with TurtleBot

5.3.1 Motivations

As a final test and in order to fully integrate the camera, we tried to plug it directly in the robot and to run the different nodes on the Raspbery Pi. This way, the robot would be able to use the camera as one of its sensors. However, taking into account the limited power of the Raspberry Pi, we pushed as much processing as possible away from the robot. To do so, we launched the camera node on the TurtleBot, while receiving the data and visualizing frames and events on a computer.

5.3.2 Results

In practice, the TurtleBot manages to retrieve the different streams from the camera and to publish their content on the corresponding topics successfully. We are also able to read those topics and to recover the data on the computer side.

We did not have any problems with frames and were able to visualize them in real time, but we faced some serious latency issues when dealing with events making them not really usable in practice. Similarly as in our latency analysis, we assumed that the latency observed could be caused either by the communication between the TurtleBot and the computer, or due to the relatively low processing power of the Raspberry Pi.

We first tried to run a simple C++ script retrieving and visualizing the events on the Raspberry Pi to have an idea of what we could expect without ROS. Even in this situation, the results obtained were not very satisfying as we could observe a bit of latency and a lack of fluidity. We assumed that if we moved the visualizer out of the robot, we could probably gain in fluidity, depending on the overhead caused by the ROS interface.

We evaluated the processing time required by the driver on the TurtleBot to illustrate the impact induced by the hardware differences between the computer and the Raspberry Pi. The results of this analysis are available in Fig.5.13. For this analysis, we tried to increase the number of events generated over time. Initially, we were capturing the movement of a person in the background. After 15 seconds, that person got closer to the camera to generate more events. After 30 seconds, we decided to shake the camera to generate a burst of events.

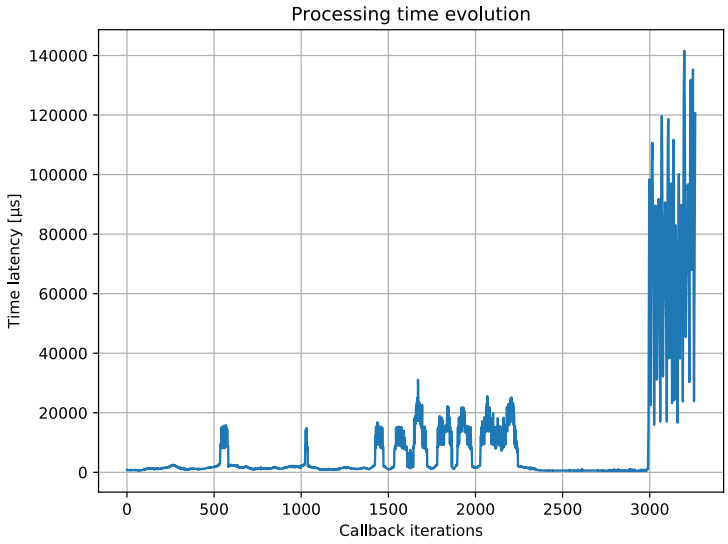

Figure 5.13: Evolution of the processing latency on the Rasbperry Pi

The first thing we can observe is that the processing time on the Raspberry Pi is way higher than on the computer. Indeed, the Raspberry Pi dealing with a low amount of events remains slower than the computer dealing with a worst-case scenario in Fig.5.8. In the case of a large number of events, the latency produced by the Raspberry Pi reaches more than 0.1s, completely delaying the next batches. The events sent afterward are then directly outdated. In addition, as those batches are not very likely to be lonely, in the sense that an important movement will be captured by several high data batches, the latency will be accumulated reaching several seconds in practice. Considering the time needed to process such batches, the driver struggles to recover from that latency.

Concerning the network latency, we initially thought that the Wi-Fi setup managing the data transfer could be too weak to manage the event stream, as the connection itself was not responding very well during the communication, maybe due to a network overload (going from less then 1ms to several hundreds). As a first improvement, we completely changed the setup to establish a wired connection between the robot and the computer with an Ethernet cable. Doing so, we considerably reduced the latency and improved the fluidity. We still performed a network latency analysis to evaluate the impact of the communication between the robot and the computer. Because of time synchronization issues between the 2 devices, we computed an approximation of this latency. For the sake of interpretability, we also removed the first batches of the graph as they were generating a huge spike due to the creation of the window displaying the events. The results of this analysis can be observed in Fig.5.14. We followed a similar scenario to what we did for the processing latency analysis. The first 1000 batches represent the movement of a person in the background. Between 1000 and 4500 batches, the person got closer to the camera to generate more events.

After 5000 batches, we shook the camera to generate a burst of events.

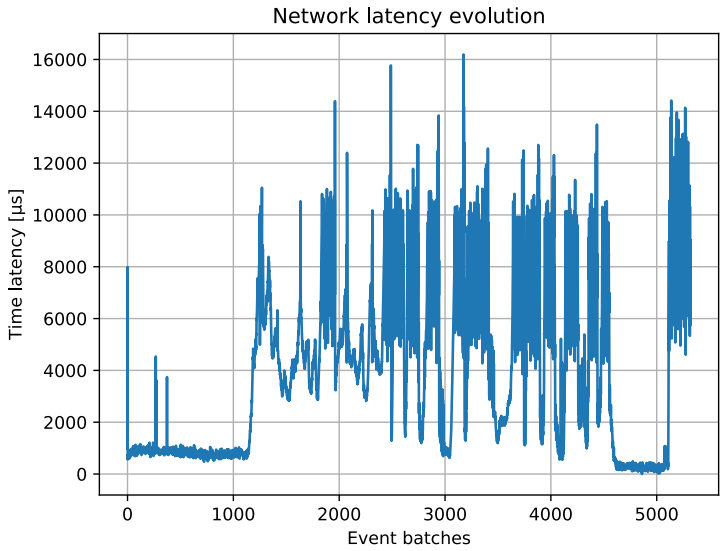

Figure 5.14: Evolution of the approximated network latency between the TurtleBot and a computer

This analysis provided 2 interesting results. The first one is that as expected, the network latency observed in this situation is higher than what we have observed in Fig.5.9 thus illustrating the impact of the multi-device communication setup. The second one is that even if we can see that this latency depends on the size of the batches sent, it always remains in the order of several milliseconds, and does not explode when dealing with huge batches. We still tried to reduce it by splitting those batches into several ROS messages, however this did not bring any improvement. We arrived to the conclusion that the latency appearing when trying to integrate the camera with the TurtleBot was caused by a combination of the high data rate of the cameras combined with the lower computing power of the Raspberry Pi. We finally managed to obtain satisfying results by only considering a subset of the events captured by the camera. For each batch, we only considered the 5000 first events. This considerably reduced the processing latency, as it made the generation of the ROS messages way faster while still keeping an accurate representation of the world. Doing so, we were able to capture and transmit up-to-date information from the TurtleBot while controlling it with a teleop node. The corresponding ROS pipeline can be observed in Fig.5.15. We believe that under the assumption of a low speed scenario, which is the case with a TurtleBot, that number could be increased to 10 000 or even 20 000 events.

In the end, even if we think that most of the latency is caused by the amount of data and the Raspberry Pi, we identified different things that could be improved in our driver regarding that latency. This will be discussed in the next chapter.

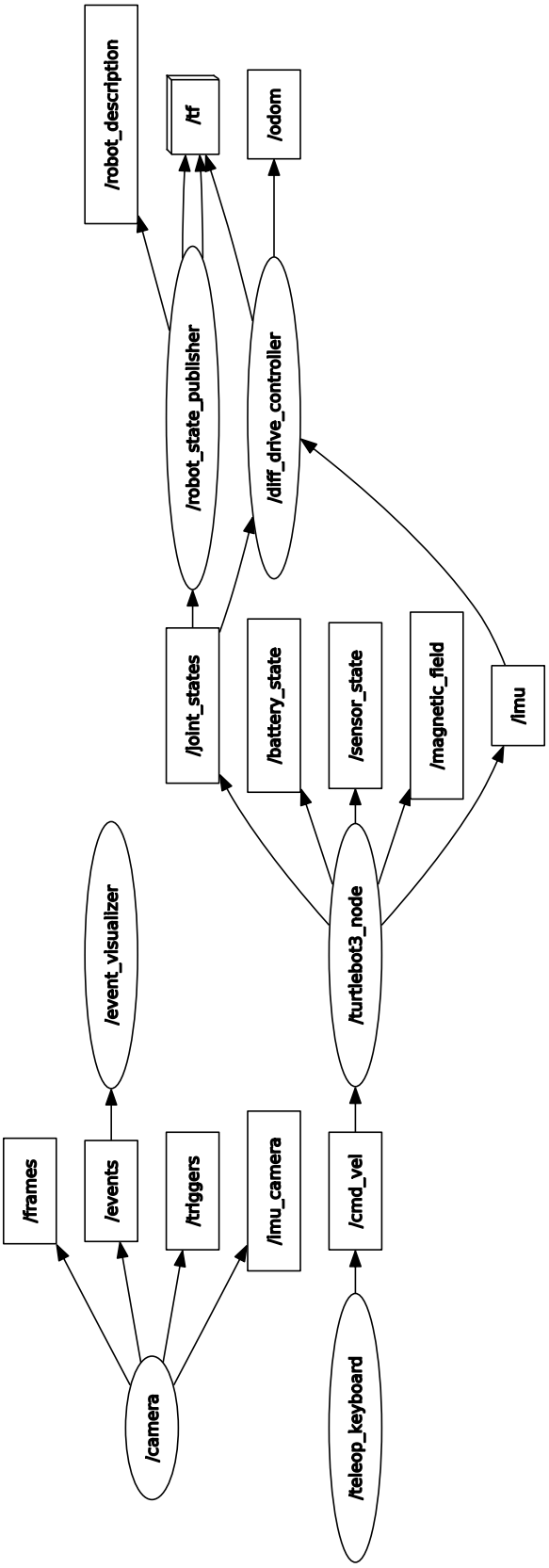

Figure 5.15: Camera integration with teleop ROS architecture

Chapter 6

Conclusion

6.1 Contributions

During this master's thesis, our goal was to design a driver acting as a compatibility layer between the event-based cameras from iniVation and a TurtleBot, in order to provide a dynamic vision interface to the robot. We first investigated the event-based sensors in details, starting from their biological inspirations to their architecture and inner principles. Doing so, we hope to have provided sufficient background to understand the challenges surrounding these types of sensors.

Following this, and to address the lack of a ROS2 package for the iniVation cameras, we implemented our own driver relying on the dv-processing library. This driver is composed of 3 nodes : the first one is able to collect the data from the 4 different streams of the camera, and the two others are respectively responsible for the visualization of the frames and the events. We believe that this driver could interest any developer willing to integrate the event-based cameras from iniVation within a ROS2 project.

Finally, we performed different experiments in order to assess the quality of our driver. We started with a latency analysis to evaluate its efficiency. By doing so, we managed to show that our driver was able to retrieve and communicate the data from the camera in a few milliseconds when being launched on a computer. In a second time, we designed as a proof of concept a small application allowing us to control the TurtleBot using the events captured by the cameras to show that our driver could be efficiently integrated in a ROS2 architecture. As a last experiment, we tried to fully integrate the cameras with the TurtleBot by running the driver on the Raspberry Pi. Facing some latency issues, we showed that the bottleneck of the setup was the Raspberry Pi, as it struggled dealing with the high data rate of the cameras. In the end, we still managed to obtain convincing results by reducing the number of events being processed by the Raspberry Pi. We believe that the objectives fixed at the beginning of this thesis have been achieved, as we are now able to succesfully integrate the cameras in a ROS2 application.

6.2 Limitations

We have identified several limitations regarding our implementation. First of all, our ROS2 driver is limited to the cameras from iniVation, as it relies on the dv-processing library. It is thus not possible to use it to manage sensors from any other manufacturer. Also, we have developed our driver using the Galactic Geochelone distribution of ROS2. Even if the core functionalities should not really differ from one ROS2 distribution to another, we do not know if the driver will be working without any modifications in a more recent distribution since we did not try.

The biggest limitation we have encountered is related to the computing power of the Raspberry Pi. Indeed, we did not manage to process 100% of the data retrieved by the camera in a reasonable amount of time and were thus forced to drop a lot of events generated when running the driver on the robot. This results in a trade-off between the quantity of data we want to consider and the latency we can tolerate while using the camera as one of the robot's sensors. We believe that this trade-off should be considered very carefully to avoid providing an incorrect representation of the world to the robot.

6.3 Further improvements

To conclude this thesis, we will mention different points still leaving room for improvement, either in the implementation or regarding topics going beyond the scope of this thesis. To begin with, there are a few quality of life features which do not impact the results obtained that could be implemented, for example a launch-file initializing the different nodes. To continue with the implementation, even if the network latency is not a problem, we believe that we could still reduce it by modifying the architecture of the EventBatch message to reduce its size, especially regarding the timestamps. Doing so, the number of bytes sent from one node to another would be reduced, which would ultimately lower the latency. Finally, we also believe that it should be possible to take advantage of parallel programming mechanisms to enhance the efficiency of the driver, even if our attempts to do so did not bring any improvement. This could be done either by splitting the processing loop in the callback function over the different cores, or by running several instances of the callback function in different threads.

As we have been limited by the capacity of the Raspberry Pi when running the driver on the robot, replacing it for a more powerful one might be a very direct way to enhance the performance. In the case were it would not be possible, we think that we could improve the event selection process in order to provide the most accurate world representation to the robot. This can be done with different event filtering algorithms. In the context of this thesis, as our goal was to implement the interface of the robot, we did not talk about any filtering methodology at all, but this is a crucial step in any event-based related task, as it can enhance the performance by selecting the most meaningful events while removing the others. However, in

the case of our TurtleBot integration, this would add even more processing on the Raspberry Pi, which is already struggling when retrieving all the events. In the end, the biggest improvement would be to enhance the computing power of the TurtleBot.

Finally, to go even further, it could be interesting to experiment our driver with the latest ROS2 distribution, as Galactic has already reached its end-of-life.

Bibliography

[1] Inivation AG, "Inivation." https://inivation.com/. Accessed: 2024-03-20.

[2] Open Source Robotics Foundation, "Turtlebot." https://www.turtlebot.com/turtlebot3/. Accessed: 2024-03-20.

[3] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics, (Kobe, Japan), May 2009.

[4] M. A. Mahowald and C. Mead, "The silicon retina," Sci Am, vol. 264, pp. 76–82, May 1991.

[5] L. Cianci, Colour Theory: Understanding and Working with Colour, ch. Anatomy of the human eye. RMIT university, 2023. https://rmit.pressbooks.pub/colourtheory1/chapter/biology-of-the-human-eye/.

[6] C. Henley, Foundations of Neuroscience, ch. 21. Vision : The Retina. Michigan State University, 2021. https://openbooks.lib.msu.edu/neuroscience/chapter/vision-the-retina/.

[7] E. M. IZHIKEVICH, Dynamical Systems in neuroscience: The geometry of excitability and bursting. MIT Press, 2007.

[8] D. NAVARRO, Architecture et Conception de Rétines Silicium CMOS: Application à la mesure du flot optique. PhD thesis, ACADEMIE DE MONTPELLIER UNIVERSITE MONTPELLIER II, 2003.

[9] Canon Europe, "Image sensors explained." https://www.canon-europe.com/pro/infobank/image-sensors-explained/. Accessed: 2024-04-10.

[10] Edge AI Vision, "Cmos vs ccd: Why cmos sensors are ruling the world of embedded vision." https://www.edge-ai-vision.com/2023/04/cmos-vs-ccd-why-cmos-sensors-are-ruling-the-world-of-embedded-vision/. Accessed: 2024-04-10.

[11] P. Magnan, "Detection of visible photons in ccd and cmos: A comparative view," Nuclear Instruments and Methods in Physics Research Section A: Accelerators,

Spectrometers, Detectors and Associated Equipment, vol. 504, no. 1-3, pp. 199–212, 2003.

[12] D. Litwiller, "Ccd vs. cmos," Photonics spectra, vol. 35, no. 1, pp. 154–158, 2001.

[13] O. Skorka and D. Joseph, "Toward a digital camera to rival the human eye," Journal of Electronic Imaging, vol. 20, no. 3, pp. 033009–033009, 2011.

[14] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. J. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 1, pp. 154–180, 2022.

[15] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128× 128 120 db 15 μs latency asynchronous temporal contrast vision sensor," IEEE Journal of Solid-State Circuits, vol. 43, no. 2, pp. 566–576, 2008.

[16] D. Gehrig, H. Rebecq, G. Gallego, and D. Scaramuzza, Asynchronous, Photometric Feature Tracking Using Events and Frames: 15th European Conference, Munich, Germany, September 8–14, 2018, Proceedings, Part XII, pp. 766–781. 09 2018.

[17] Prophesee, "Prophesee metavision technologies." `https://www.prophesee.ai/`. Accessed: 2024-04-20.

[18] Y.-l. Tian, L. Brown, A. Hampapur, M. Lu, A. Senior, and C.-f. Shu, "Ibm smart surveillance system (s3): event based video surveillance system with an open and extensible framework," Machine Vision and Applications, vol. 19, pp. 315–327, 2008.

[19] J. Howell, T. C. Hammarton, Y. Altmann, and M. Jimenez, "High-speed particle detection and tracking in microfluidic devices using event-based sensing," Lab on a Chip, vol. 20, no. 16, pp. 3024–3035, 2020.

[20] J.-A. Sahel, E. Boulanger-Scemama, C. Pagot, A. Arleo, F. Galluppi, J. N. Martel, S. D. Esposti, A. Delaux, J.-B. de Saint Aubert, C. de Montleau, et al., "Partial recovery of visual function in a blind patient after optogenetic therapy," Nature medicine, vol. 27, no. 7, pp. 1223–1229, 2021.

[21] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," Science Robotics, vol. 7, no. 66, p. eabm6074, 2022.

[22] Open Robotics, "Ros - robot operating system." `https://www.ros.org/`. Accessed: 2024-04-27.

[23] Inivation, "Dv-processing." `https://dv-processing.inivation.com/rel_1_7/index.html`. Accessed: 2024-04-27.

[24] Open Robotics, "Ros noetic ninjemys." http://wiki.ros.org/noetic/. Accessed: 2024-04-28.

[25] Luca Longinotti, "dv-ros." https://gitlab.com/inivation/dv/dv-ros, 2022. Accessed: 2024-04-28.

[26] SBG systems, "Imu - inertial measurement unit." https://www.sbg-systems.com/inertial-measurement-unit-imu-sensor/. Accessed: 2024-04-29.

[27] Open Robotics, "sensor_msgs." http://wiki.ros.org/sensor_msgs. Accessed: 2024-04-29.

[28] Open Robotics, "geometry_msgs." http://wiki.ros.org/geometry_msgs. Accessed: 2024-04-29.

[29] Open Robotics, "cv_bridge." http://wiki.ros.org/cv_bridge. Accessed: 2024-04-29.

[30] Open Robotics, "Ros2 documentation : foxy." https://docs.ros.org/en/foxy/index.html. Accessed: 2024-05-01.

[31] Open Robotics, "Ros2 documentation : galactic." https://docs.ros.org/en/galactic/index.html. Accessed: 2024-05-01.

[32] Petter Nilsson, "Remove deprecated (in c++17 and newer) use of std::allocator<>::rebind." https://github.com/ros2/rclcpp/pull/1678. Accessed: 2024-05-01.

[33] Open Robotics, "Ros2 documentation : humble." https://docs.ros.org/en/humble/index.html. Accessed: 2024-05-01.

[34] iniVation, "Specifications – current models." https://inivation.com/wp-content/uploads/2023/11/2023-11-iniVation-devices-Specifications.pdf. Accessed: 2024-05-03.

[35] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbruck, "A 240 × 180 130 db 3 µs latency global shutter spatiotemporal vision sensor," IEEE Journal of Solid-State Circuits, vol. 49, no. 10, pp. 2333–2341, 2014.

[36] iniVation, "Dvxplorer mini." https://inivation.com/wp-content/uploads/2023/03/DVXplorer-Mini.pdf. Accessed: 2024-05-03.

[37] iniVation, "Understanding the performance of neuromorphic event-based vision sensors." https://inivation.com/wp-content/uploads/2020/05/White-Paper-May-2020.pdf. Accessed: 2024-05-03.

[38] Robotis, "Turtlebot3." https://www.robotis.us/turtlebot-3/. Accessed: 2024-05-05.

[39] Robotis, "Robotis-git : Turtlebot3." https://github.com/ROBOTIS-GIT/turtlebot3. Accessed: 2024-05-05.

[40] Open Robotics, "Gazebo." `https://gazebosim.org/home`. Accessed: 2024-05-07.

[41] Open Robotics, "tf - ros." `http://wiki.ros.org/tf`. Accessed: 2024-05-07.

Appendix A

Event-based pixel full circuit

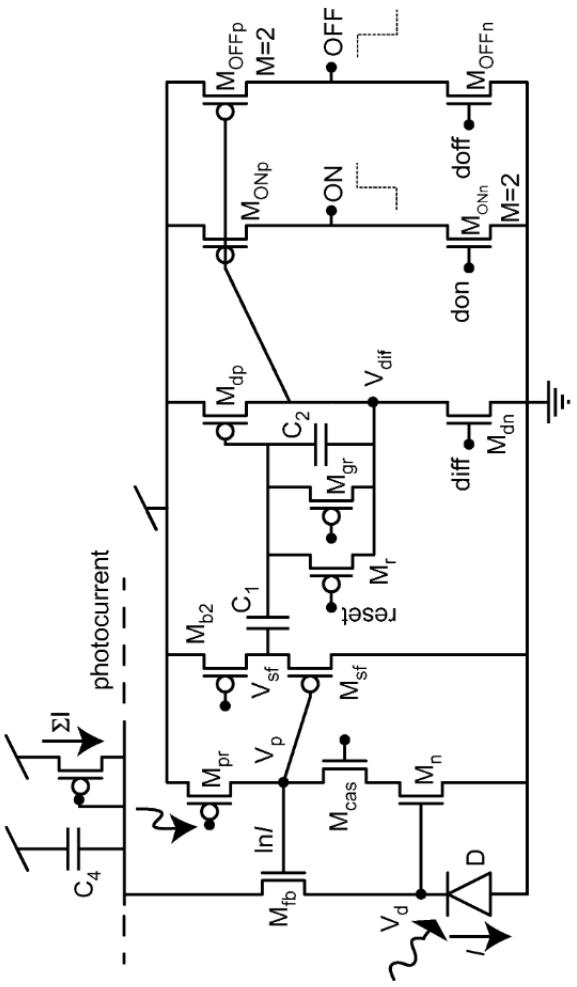

Figure A.1: Transistor-level event-based pixel [15]

Appendix B

Latency analysis with DVXplorer device

	Min size	Max size	Average size	Average process latency	Average network latency
Motionless	1516	8182	2093.98	616µs	532.47µs
Regular	845	165569	13460.56	1089.94µs	776.19µs
Worst case	5984.0	151844.0	67085.62	2200.39µs	1479.08µs

Table B.1: Latency analysis results with the DVXplorer camera

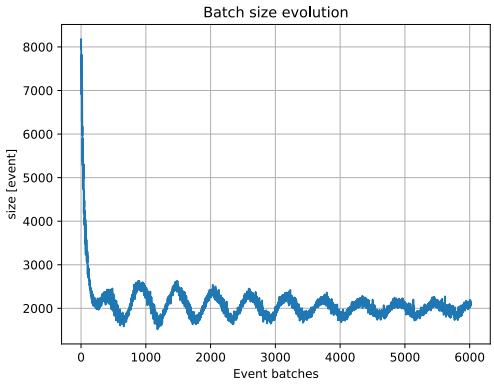

Figure B.1: Evolution of the batch size over 1 minute in a motionless scenario

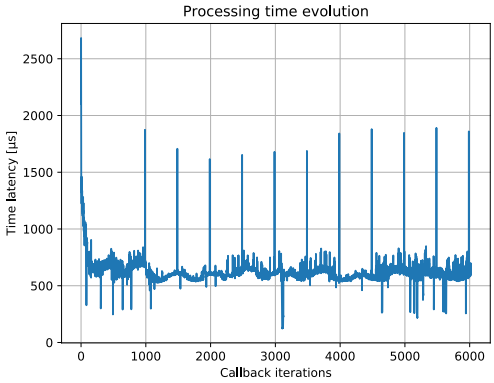

Figure B.2: Evolution of the processing latency over 1 minute in motionless scenario

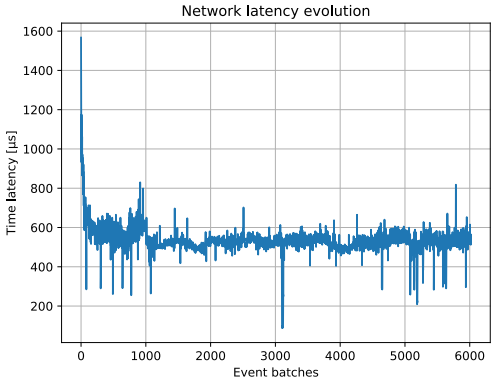

Figure B.3: Evolution of the network latency over 1 minute in a motionless scenario

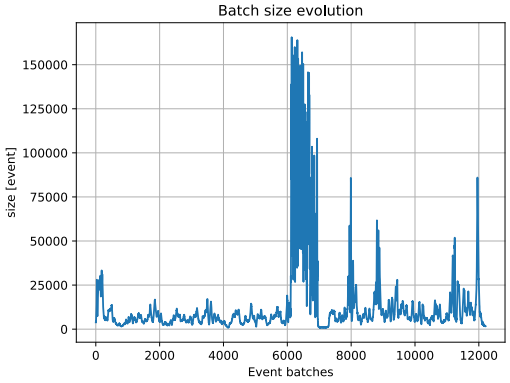

Figure B.4: Evolution of the batch size over 2 minutes in a regular scenario

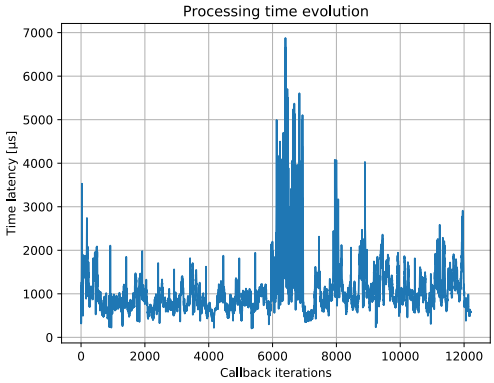

Figure B.5: Evolution of the processing latency over 2 minutes in regular scenario

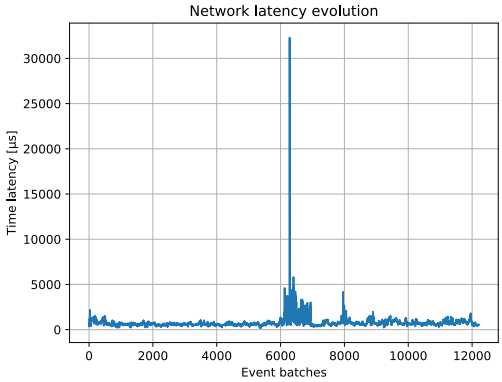

Figure B.6: Evolution of the network latency over 2 minute in a regular scenario

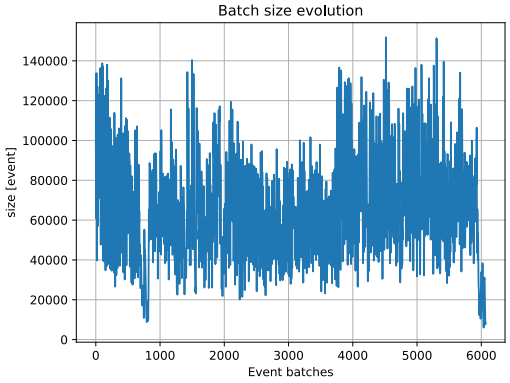

Figure B.7: Evolution of the batch size over 1 minute in a worst case scenario

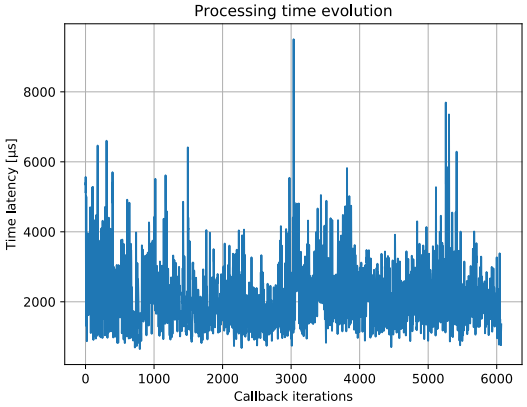

Figure B.8: Evolution of the processing latency over 1 minute in a worst case scenario

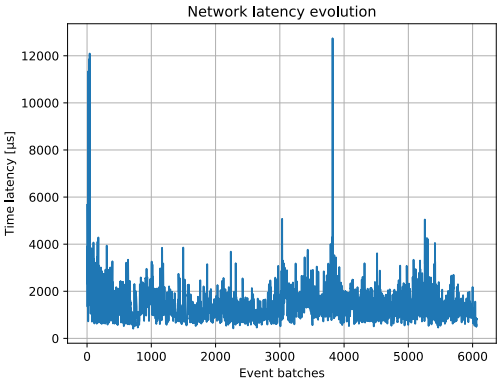

Figure B.9: Evolution of the network latency over 1 minute in a worst case scenario