Eötvös Loránd University

Faculty of Informatics

Department of Software Technology

Log analyzer for real-time DSP scheduling framework

Supervisor:

Zoltán Gera

tanársegéd, MSc Computer Science

Author:

Hossameldin Abdin

Computer Science BSc

Budapest, 2021

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:

Student's Name: Abdin Hossameldin

Student's Neptun code: BELNRM

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Zoltán Gera*

Supervisor's Home Institution: *ELTE IK*

Address of Supervisor's Home Institution: *1117 Bp. Pázmány Péter sétány 1/C*

Supervisor's Position and Degree: *tanársegéd, MSc Computer Science*

Thesis Title: Log analyzer for real-time DSP scheduling framework

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

1 Introduction:

Logging various information regarding different aspects of projects is vital to measure the sanity and the behavior of a system. Unfortunately in many cases especially in a case of a huge amount of information to log, the advantage turns into an issue that takes time and effort from the developer(s) to be able to check the sanity of a created system or a program and from that point the logging becomes a burden which takes from the efficiency of the program without giving back the wanted/requested quality of results. Also the debugging in a real-time system is not possible, the log analyzing is the only option in such a system. From the issue described above, the idea of a log analyzer was born. The log analyzer will support the DSP (Digital Signal Processing) framework PipeRT which is developed at ELTE University.

2 General information about PipeRT:

PipeRT is a hybrid scheduling and data flow framework for DSP applications, which offers high performance and easy to use framework. (for more info: https://github.com/gerazo/pipert/blob/master/README.md)

3 The responsibility of the log analyzer:

The log analyzer should be a separate API which can communicate with the framework to have a low delay live sanity checking for the system, it should be able to represent the pipeline of the framework visually and it should spot the bottleneck in the system if any. The analyzer should generate statistics that can reflect the status of the system as a whole.

4 Goal:

Offering the developers of DSP applications who uses the PipeRT framework a detailed yet understandable representation of their development's pipeline. The analyzer is supporting the measurement oriented approach of the development.

Contents

Chapter 1

Introduction

1.1 Motivation

Logging is not an easy addition to any system but becomes useful only with a tool that knows how to extract valuable data from a huge stream and from these data can bring an overview, and statistics that describe the behavior and analyze it. The log analyzer became essential not only to support such a system and shows its flows, but also to visualize a picture to force the developer(s) to see what he/she never expected.

For all the reasons mentioned and more, building a log analyzer to show the bottlenecks and help the developer(s) of DSP (Digital signal processing) application who are using the PipeRT framework was a project eager to be born.

1.2 Thesis Structure

This thesis is composed of 4 main chapters, a bibliography, a list of figures, and a list of tables.

Chapter 2 is going to introduce the user documentation, including how to install and run the analyzer.

Chapter 3 contains the developer documentation with detailed Structure of the implementation, and its capabilities to be extended.

Chapter 4 is the conclusion, and the summary of the project can be found there, with ideas to be added to the project.

Chapter 2

User Documentation

This chapter contains a brief description of the project, a guide on how to install and run the analyzer, and the way to use.

2.1 Project Description

This project is a log analyzer working alongside the PipeRT framework with its profiler. The project aims to analyze the continuous stream of data coming from the PipeRT's profiler, in a server-client relationship.

The analyzer was build using python in the backends and Javascript in the frontend, it provides various checkers to investigate the sanity of the pipeline created by user using PipeRT, graphs associated with the measurements, and visualization of the pipeline and how the channels are structured.

The analyzer in itself was built to be an analyzing framework where extensibility was the key and will be in the design decisions, so as a result, it is relatively easy to add new features and already prepared to be extended by new checkers and measurements.

2.2 Installation Guide

PipeRT is currently supporting Linux only, but soon, it will support Windows as well. That said, the steps to install the analyzer are the same in both operations systems.

The `log_analyzer` folder inside the PipeRT project folder is where all of the Installation steps will take place.

Python 3.9 should be installed on the operating system. All the requirements can be installed by typing the following command in the terminal or the command prompt.

```
$ pip install requirements.txt
```

Code 2.1: Install requirements

It is also recommended to create a python virtual environment before installing the requirements, in order to have a separate environment for the analyzer. The commands to create and run the environment are:

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Code 2.2: Create virtual environment

2.2.1 Running

In order to run the analyzer make sure to be inside the `log_analyzer` folder and type the following command:

```
$ python start.py
```

Code 2.3: Start application

This will start the application, so once you type `127.0.0.1:5000` in the browser (Firefox recommended). You will be able to see the following:

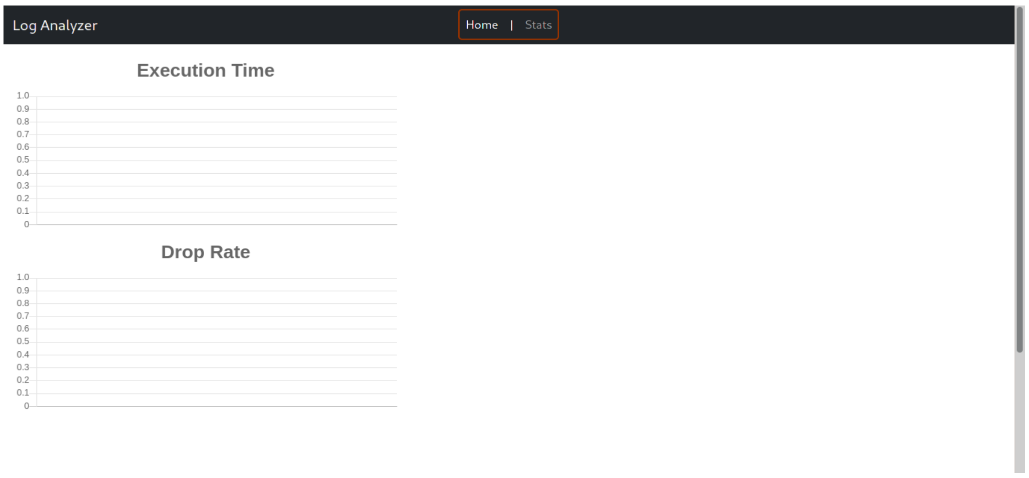


Figure 2.1: Start of the application

2.3 Client-Server Configuration

To establish a connection between the profiler (Client) and the log analyzer (server), there should be modifications in both sides.

2.3.1 Server Side

The profiler is the utility for monitoring the DSP pipeline and sending logs, it has 3 arguments, first is `destination_uri`, which describes the destination and the used protocol, udp and file are the protocol options in the profiler currently. The Second argument is `aggregation_time_msec` which is the time in milliseconds to wait before gathering monitoring data again, so it determines how often aggregated log data is sent to the log processor, if not given that means not to collect periodically. The third argument is `buffer_size`, it controls the size of buffer which is filled to be sent at once, the default value depends on the protocol chosen.

To establish a connection with the analyzer, the udp protocol is the one to choose, the IP and socket are based on the user preference.

Adding the profiler to the scheduler is the last step to configure the server-side, and the following example shows how to add the profiler.

```
  pipert::Scheduler sch(0, pipert::Profiler("udp:127.0.0.1:8000"));
```

Code 2.4: Adding profiler

2.3.2 Client Side

On the client-side, the same port number that has been provided to the profiler should be added in the **config.json** inside the `log_analyzer` folder. same for the IP as well.

```
{
    "port": 8000,
    "ip": "127.0.0.1"
}
```

Code 2.5: connection configuration

An important note: The log analyzer will start analyzing and draw the visualization, at the end of the first packet cycle **??**, so, the web page will be the same as 2.1 until the completion of the cycle.

2.4 Analyzer Configuration

The log analyzer is made to check the sanity and analyze various applications and systems, so having a configuration, was essential.

The configuration is a JSON (JavaScript Object Notation) file, the choice of the format to be JSON was due to its lightweight, well-known among developers, and simplicity. The config.json configuration file consists of 3 main parts, general, measurements' and checkers' configurations. Let's enumerate these configurations.

```
  {
    "PORT": "8000",
    "IP": "127.0.0.1",
    "PACKET_CYCLE_THRESHOLD": 1000,
    "pipeline_measurements": [
      {
        "name": "pipeline_measurement_1",
        "enabled": true,
        "key": "Pipeline Measurement 1"
      }
    ],
    "channel_measurements": [
      {
        "name": "channel_measurent_1",
        "enabled": true,
        "key": "Channel Measurement 1"
      }
    ],
    "checkers": [
      {
        "name": "checker_1",
        "enabled": true,
        "measure_key": "Channel Measurement 1",
        "parameters": {
          "THRESHOLD": 20
        }
      }
    ]
  }
```

Code 2.6: Sample configuration

2.4.1 General Configurations

Consist of 3 configurations, **PORT**, **IP**, and `PACKET_CYCLE_THRESHOLD`. The first two were discussed in 2.3.2.

The `PACKET_CYCLE_THRESHOLD` is an integer value that describes how many packets should the analyzer receive before running the checkers once and these packets' events will be stored in the channels and when the cycle finishes (when the number of packets received is equal to `PACKET_CYCLE_THRESHOLD`), these events will be deleted from the channels and a new packets cycle starts.

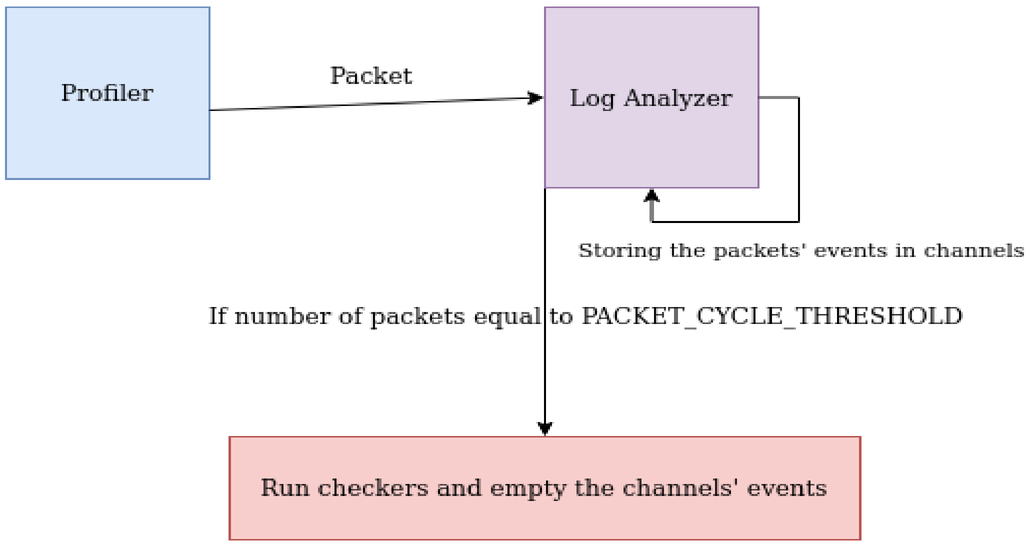

Profiler
Packet
Log Analyzer
Storing the packets' events in channels
If number of packets equal to PACKET_CYCLE_THRESHOLD
Run checkers and empty the channels' events

Figure 2.2: Packets cycle

2.4.2 Measurements' Configurations

These configurations are split into two categories, the pipeline's measurement, and the channel's measurements. Despite their difference from the measuring point of view as discussed in **??**, they do share the same texture of their configuration. (2.6)

Both of the two main measurements are a list of dictionaries, where each dictionary represents a measurement, and each measurement has three key-value pairs, **name** to be able to dynamically import them as in **??**, **enabled** to turn the measurement on or off, and **key** this will be the shown text for the measurement and will also help to connect the measurement to a specific checker if needed (see **??**).

2.4.3 Checkers' Configurations

As it is shown from 2.6, the checkers' configuration is a list of dictionaries where each dictionary is a checker's configuration. Each configuration contains the checker's name in the **name** field, whether it should work or not in the **enabled** field, **measure_key** to specify the measurement for that checker if needed, and a **parameters** field where the parameter of the checker can be created or changed.

The **name** field should be the same name as the file of the checker without the extension (.py). And that is how the dynamic importing module in the project will be able to import the checker with its configuration **??**.

The **enabled** field is a boolean field to turn on or off the checker.

The **measure_key** field is a string to connect the checker with its corresponding measurement.

The **parameters** field is a dictionary with as many keys as the checker's parameters. These values can be changed based on the DSP application using the analyzer, and it is the user's responsibility to assign the appropriate values.

2.5 Measurements

The backbone of the log analyzer is the measurements, and that is the reason for implementing various of them, given these varieties of measurements, it is expected to be able to help to check the sanity of the pipeline and spotting the bottlenecks for the user. They are also essential for most of the checkers (more about it in 2.6).

They consist of two categories, **channel's** and **pipeline's** measurements. All the measurements run once per packet cycle and the graphs resulted should be drawn every ten cycles, as can be seen from figure 2.3. The following pages contain a detailed description of the categories and their measurements.

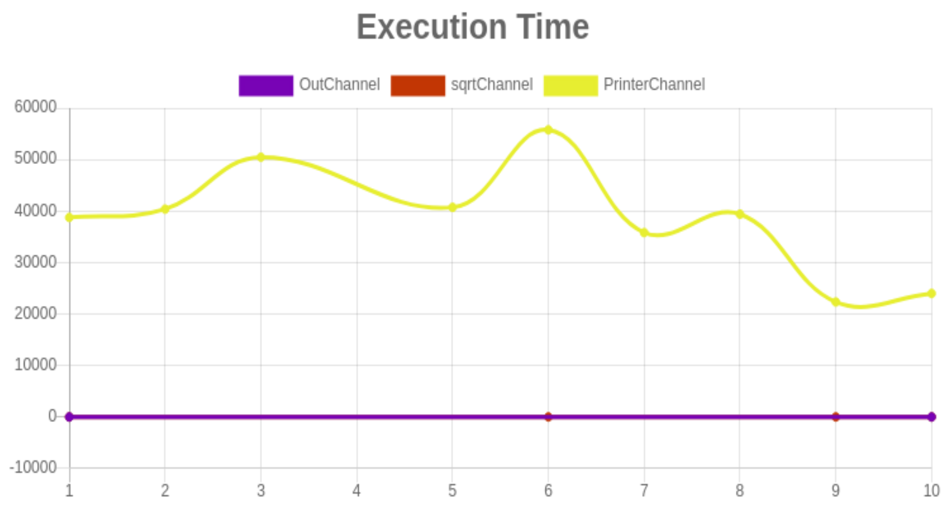


Figure 2.3: Measurement's graph

2.5.1 Channel's Measurements

The measurements in this section focus on the channels and their attributes. They are seven implemented measurements, and the details of them will be explained below.

Drop Rate Measurement

Calculating the number of dropped events over the number of executed events. In case of no executed events, the result will be **1**, if no dropped events, it will return **-1**, and if both events are missing **-1** will be the output of the calculation.

Drop Ratio Measurement

Calculating the number of dropped events over the number of read events. In case of no read events, the result will be **1**, if no dropped events, it will return **-1**, and if both events are missing **-1** will be the output of the calculation.

Execution Time Measurement

This measurement is responsible of calculating the average of the execution time by dividing the sum of the execution time events' average over the number of the execution time. In case the is no execution time events, it will output **-1**.

Fill Time Measurement

Its responsibility is calculating the average of the fill time by dividing the sum of the fill time events' average over the number of the fill time. In case the is no fill time events, it will output **0**.

Time To Buffer Average Measurement

2.6 Checkers

Checkers results flags and measurements of different aspects of the channel and pipeline. All the enabled checkers run for all the channels in the pipeline and in case the check passes, it will appear in the webpage as in 2.4, on the other hand, figure 2.5 shows the representation of a failed check in the current packets cycle.

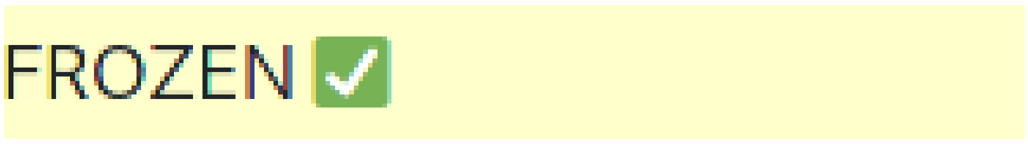


Figure 2.4: Passed checker

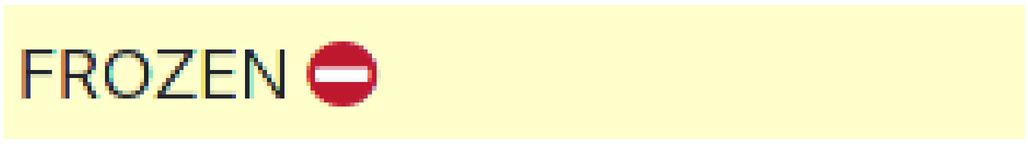


Figure 2.5: Failed checker

The checkers are stateless, so no information from the previous cycles is saved. Each checker has a name, can have a configuration, and a description of what does it check. We are going to enumerate these information in the following pages.

2.6.1 Frozen Checker

The checker name on the webpage is `FROZEN`, it checks if a channel received one or more events in the last packets cycle, if it did not receive any, the check fails for this cycle.

2.6.2 Drop Rate Checker

The name on the webpage is `HIGH_DROP_RATE`, it Calculates the drop rate of a channel (see **??**), and passes if and only if the rate is bigger than the configured threshold value. The configuration consists of one key-value pair, `DROP_RATE_THRESHOLD` is the key, and the already configured value is **0.5**.

2.6.3 Drop Ratio Checker

The name on the front-end is `HIGH_DROP_RATIO`, it is responsible for calculating the drop ratio (see **??**), and if the calculated value exceeds the threshold configured then the check fails. The `DROP_RATIO_THRESHOLD` is the only configuration and **0.75** is the default value.

2.6.4 Execution Time Checker

`HIGH_EXECUTION_TIME` is the name shown on the page, it checks whether the execution time of a channel did surpass the configured threshold. The calculation of the execution time can be found at **??**. **0.75** is the default value for the only configuration `EXECUTION_TIME_THRESHOLD`.

2.6.5 Read Time Checker

Very Similar to the **Execution Time Checker**, but checking for the reading time. The configuration consists of one key-value pair, which is the `CHANNEL_READ_THRESHOLD` and the default value is **20**. The name on the page is `HIGH_READ_TIME`. (see **??** for the measurement details)

2.6.6 Fill Time Checker

Having the same methodology for checking as the **Execution Time Checker** and the **Read Time Checker**. Its configuration contains only one key-value pair to set the threshold of the filling time of a channel. `CHANNEL_FILL_THRESHOLD` is the configuration's name, and the default value is **20**. The checker page's name is `HIGH_FILL_TIME`.

2.6.7 Time To Buffer Average Checker

2.7 Graphical User Interface

The analyzer interface is a webpage containing two pages, **Home** and **Measurements**. In this section, I will describe the components of each page and the features provided.

The navigation bar (figure 2.6) is shared between the two pages and controls navigating between the two of them. To load the **Home** page, you can click on either the *Home* or *LogAnalyzer*. In case of **Measurements**, clicking on *Measurements* will load it.

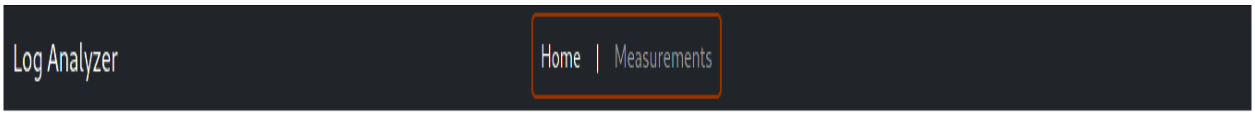


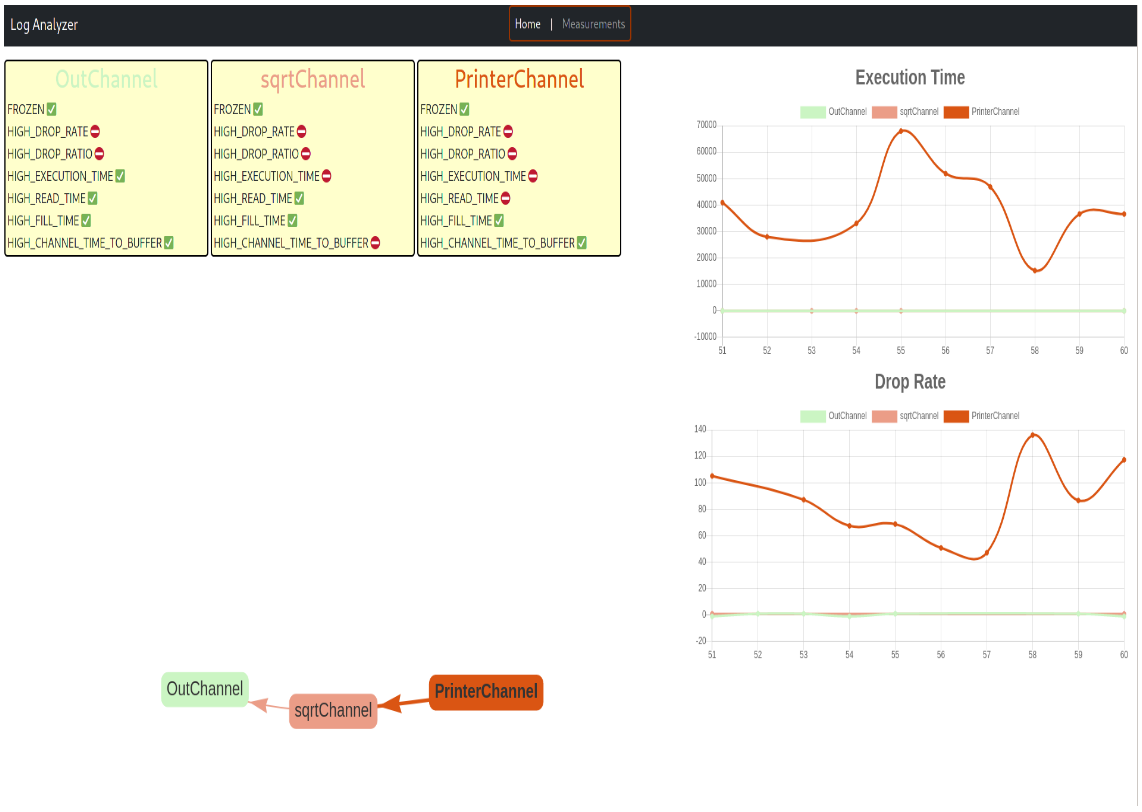
Figure 2.6: The navigation bar

2.7.1 Home Page

The home page should give a brief description of the pipeline's status through its components. It consists of checkers for channels (2.8), measurements section (2.9), and pipeline visualization (2.10).

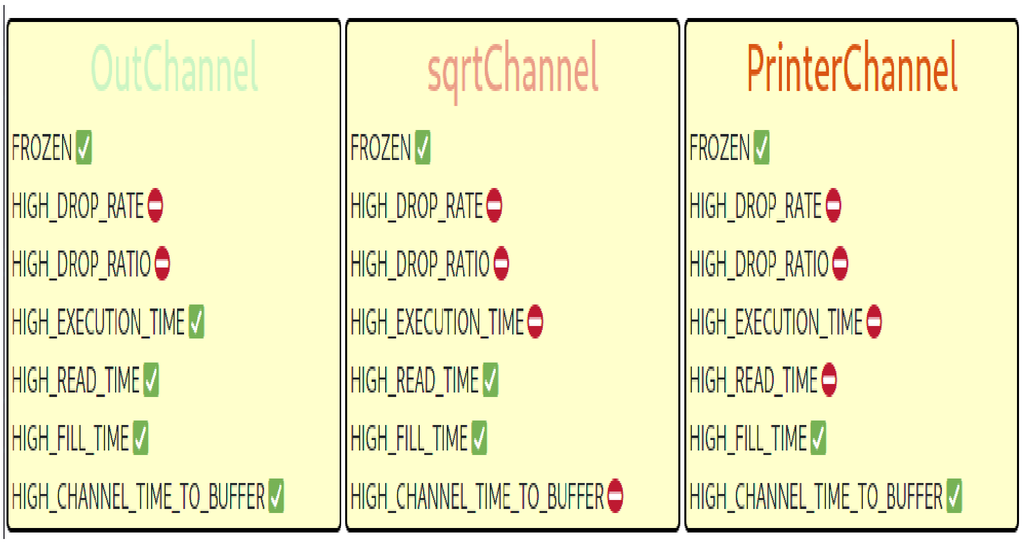
As mentioned before, before completing the first packet cycle, the home page will look as in figure 2.1. Once the profiler is sending regularly, you can expect to have the home page similar to figure 2.7.

Each channel has a corresponding color which is used through the different components in order to ease the distinguishment and to have a visual identity belong to each of them.



OutChannel	sqrtChannel	PrinterChannel
FROZEN ✓	FROZEN ✓	FROZEN ✓
HIGH_DROP_RATE ⛔	HIGH_DROP_RATE ⛔	HIGH_DROP_RATE ⛔
HIGH_DROP_RATIO ⛔	HIGH_DROP_RATIO ⛔	HIGH_DROP_RATIO ⛔
HIGH_EXECUTION_TIME ✓	HIGH_EXECUTION_TIME ⛔	HIGH_EXECUTION_TIME ⛔
HIGH_READ_TIME ✓	HIGH_READ_TIME ✓	HIGH_READ_TIME ⛔
HIGH_FILL_TIME ✓	HIGH_FILL_TIME ✓	HIGH_FILL_TIME ✓
HIGH_CHANNEL_TIME_TO_BUFFER ✓	HIGH_CHANNEL_TIME_TO_BUFFER ⛔	HIGH_CHANNEL_TIME_TO_BUFFER ✓

Figure 2.7: Home page in running state



OutChannel	sqrtChannel	PrinterChannel
FROZEN ✓	FROZEN ✓	FROZEN ✓
HIGH_DROP_RATE ⛔	HIGH_DROP_RATE ⛔	HIGH_DROP_RATE ⛔
HIGH_DROP_RATIO ⛔	HIGH_DROP_RATIO ⛔	HIGH_DROP_RATIO ⛔
HIGH_EXECUTION_TIME ✓	HIGH_EXECUTION_TIME ⛔	HIGH_EXECUTION_TIME ⛔
HIGH_READ_TIME ✓	HIGH_READ_TIME ✓	HIGH_READ_TIME ⛔
HIGH_FILL_TIME ✓	HIGH_FILL_TIME ✓	HIGH_FILL_TIME ✓
HIGH_CHANNEL_TIME_TO_BUFFER ✓	HIGH_CHANNEL_TIME_TO_BUFFER ⛔	HIGH_CHANNEL_TIME_TO_BUFFER ✓

Figure 2.8: Checkers in the home page

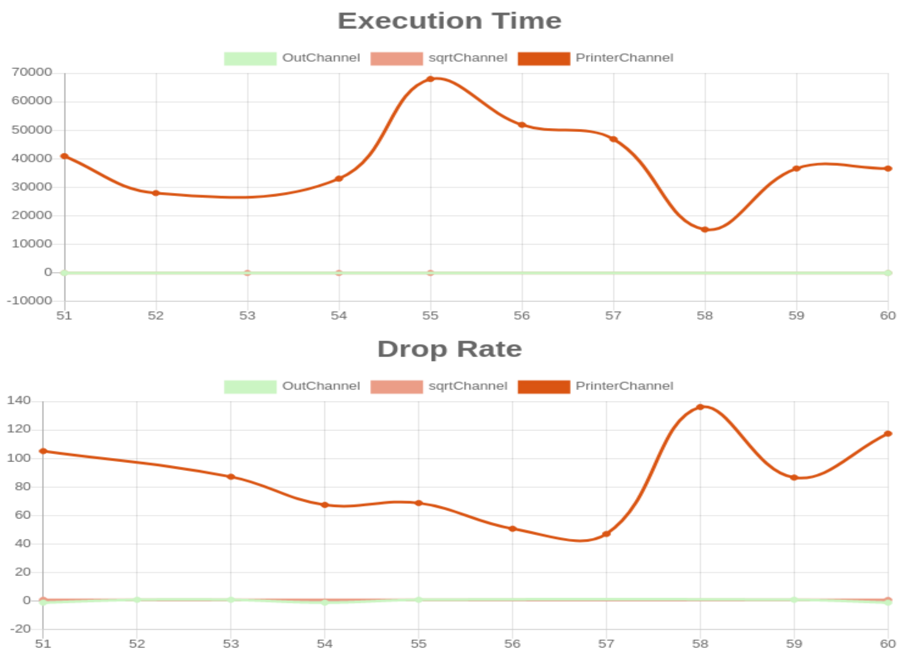


Figure 2.9: Measurements in the home page

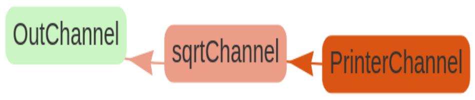


Figure 2.10: Pipeline's visualization in the home page

Channels' Checkers

They are located in the top left of the home page (2.7), each channel is represented as a box where the name of the channel is at the top of it and the checkers and their status come after.

Measurements

They can be found in the top right of the home page (2.7). They are two measurements on the page, **execution Time** and **drop rate**, these two measurements were chosen because of their capabilities of reflecting important aspects of the channels.

As shown in figure 2.3, the name of the measurement is at the top, and the channels' names and colors following come after, and then the graph itself. Each channel corresponds to the color on the left of its name.

To be able to see the exact axis of a certain point of the graph, you can hover with the mouse, and the result will be as in figure 2.11. The name of the channel and its color will also be included.

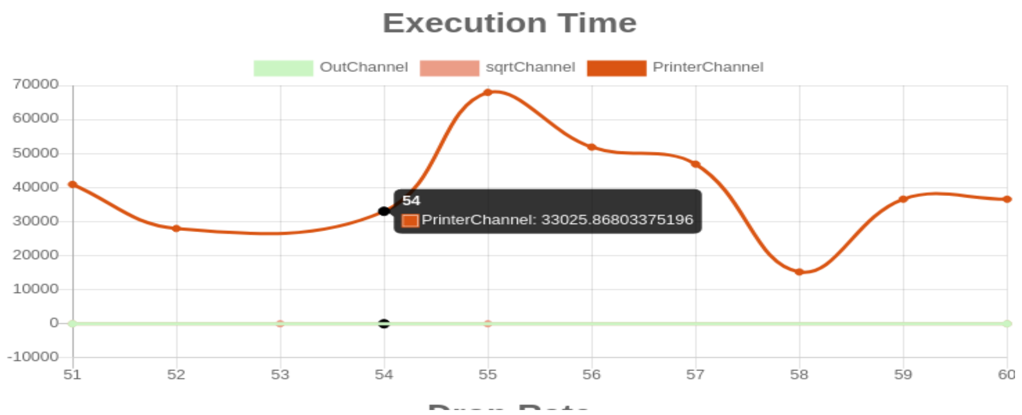


Figure 2.11: Hovering a point in a graph

In many cases, some channels can be unseen because of other channels' high y-axis value. If you want to hide the channel line graph, you can click on the channel's name or color, and there will be a horizontal line on the clicked channel's name to indicate that its measurements will not be shown in the graph. Figure 2.13 shows the effect of this action.

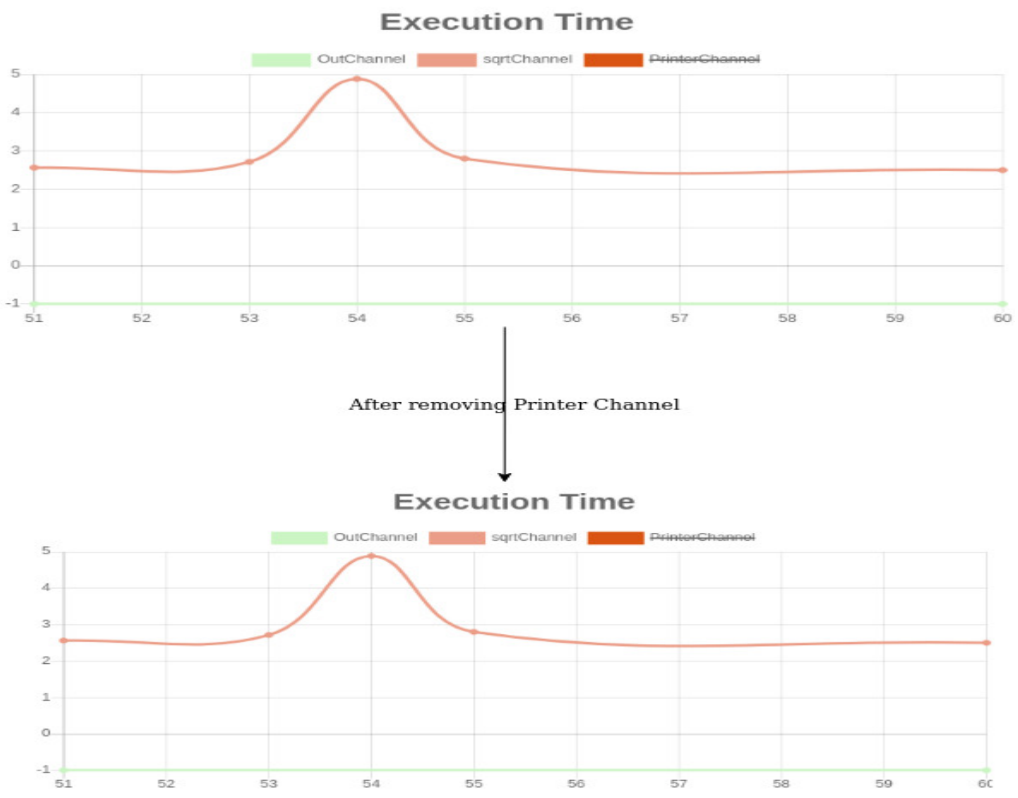


Figure 2.12: Graph after removing a channel

Pipeline Visualization

It is located at the bottom of the home page. Each box contains the name of a channel, and the box's color is the channel's color, and the arrows between these boxes represent the connection between the channels in the actual pipeline created by the user as in figure 2.10.

You can change the position of the pipeline visualization by clicking anywhere in the bottom half of the page beside the channels' boxes themselves and dragging the visualization around. It is also possible to change the shape of the visualization or rearrange it by clicking any of the channels' boxes and dragging.

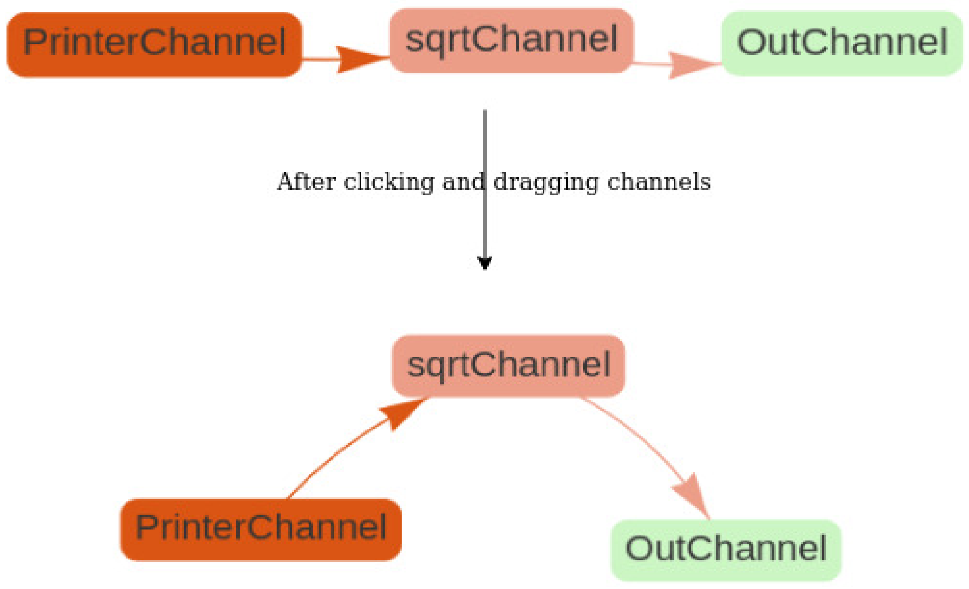


Figure 2.13: Rearranging the pipeline visualization

2.7.2 Measurements Page

The page contains all the measurements explained in 2.5, it is divided into **channels'** and **pipeline's** measurements as in figure 2.14 and 2.15. The channels' measurements come before the pipeline's measurements.

The graphs represent 10 packet cycles of their measurements. Each channel has the same color through all the various graphs to easily distinguish between each other same as in the **Home** page.

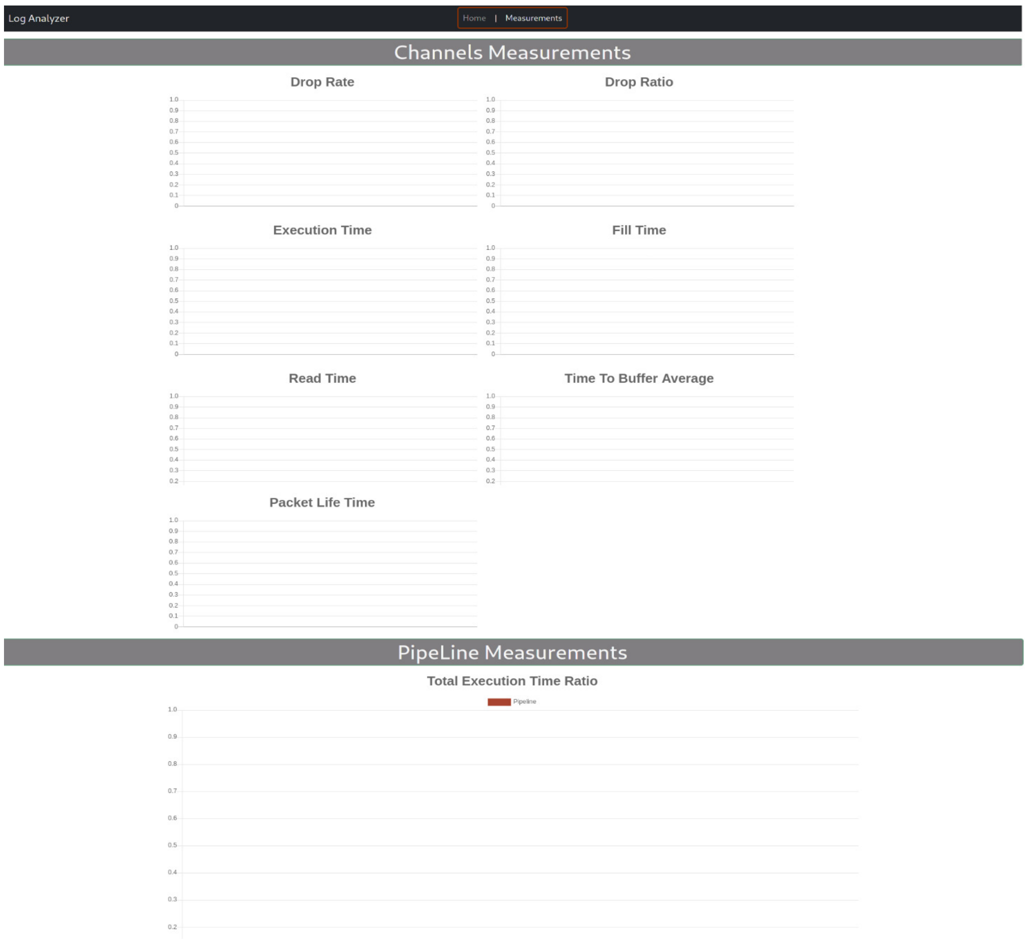


Figure 2.14: Measurements page in empty state

All the features that were in the measurements section of the *Home* page (2.7.1) are extended to this page as well, including removing one or more channels from the graph and seeing the exact coordinate of a point.

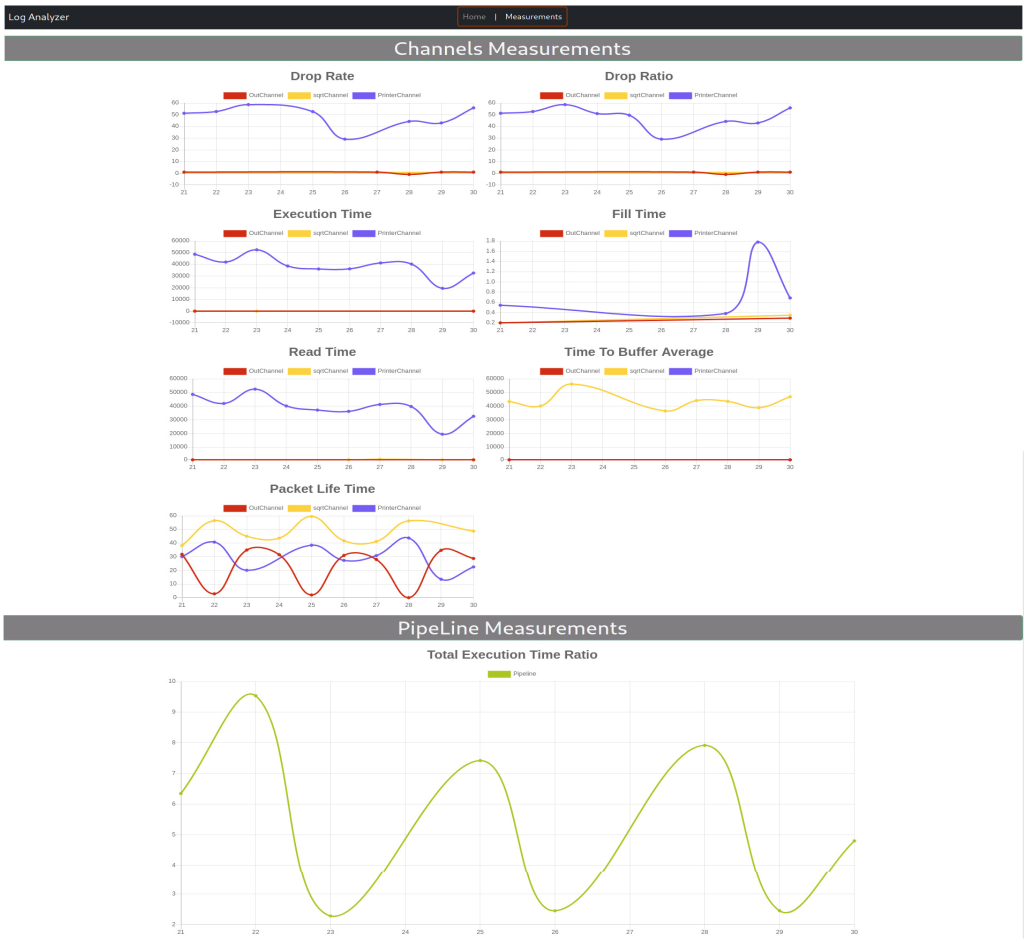


Figure 2.15: Measurements page in running state

Chapter 3

Developer Documentation

In this chapter, we are going to dive deep into the details of creating the log analyzer, the architecture, and why the different parts of it were necessary, and how they all fit together.

To build and run the project as a developer, it is the same steps as in 2.2. The user is expected to be able to add to the analyzer to match their different need and that is why the roles of the developer and the user are very close to each other in the project.

As mentioned before, the log analyzer and the profiler are very tight, and to understand the analyzer's design and its components, an overview of the profiler is needed. The next section covers this overview.

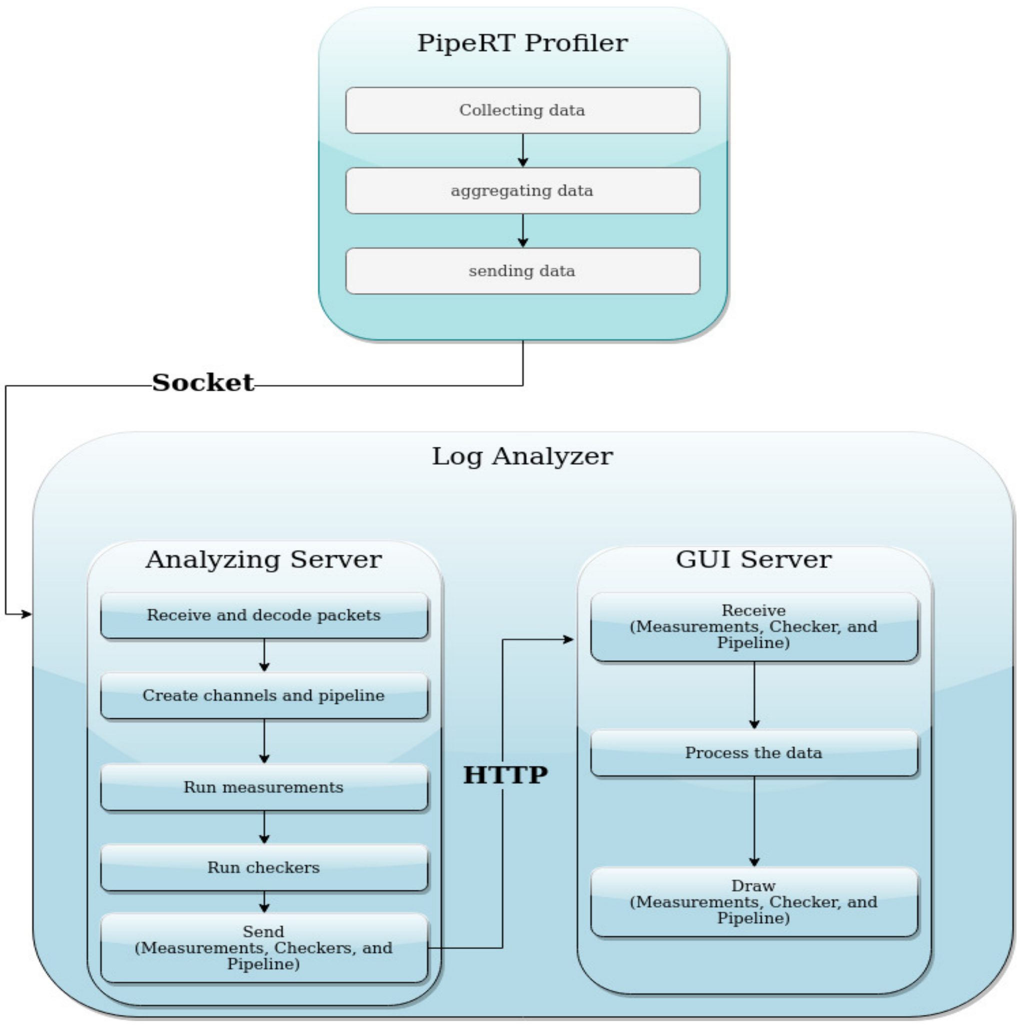

PipeRT Profiler
Collecting data
aggregating data
sending data
Socket
Log Analyzer
Analyzing Server
Receive and decode packets
Create channels and pipeline
Run measurements
Run checkers
Send (Measurements, Checkers, and Pipeline)
HTTP
GUI Server
Receive (Measurements, Checker, and Pipeline)
Process the data
Draw (Measurements, Checker, and Pipeline)

Figure 3.1: General workflow

3.1 PipeRT Profiler

The profiler is the utility in PipeRT responsible for profiling and monitoring functionality. It is a sub-system to collect, aggregate, serialize, and send data to the log analyzer. The different configurations of the profiler can be found at 2.3.1.

The collection and aggregation of the profiler happen over different types of events, there exist several predefined events, however, the user can also add their events if needed. These different events are the analyzer guideline for the channels' measurements and to understand the pipeline and measure it.

3.1.1 Events

The events are actions that happen to a packet in a channel. There are six predefined events, **Pushed**, **Retrieved**, **Execution Time**, **Fill Time**, **Read Time**, and **Dropped Packet**.

Since the profiler is supposed to aggregate these events, the *log aggregator*, a helper utility in the profiler, is going to group these events based on the event type, and these aggregated data will be sent to the log analyzer. Each aggregated event will have a log count, and that is the number of logged events, and the time aggregation started, minimum time, maximum time, and the average time that event took to complete.

Pushed Event

Time in packet's timeline when the packet is filled with data and pushed into the channel buffer.

Retrieved Event

Time in packet's timeline when the packet is retrieved for processing.

Execution Time Event

Time spent by a scheduler thread servicing a channel callback.

Fill Time Event

Time spent between acquiring the packet and pushing it into the buffer

Read Time Event

Time spent between retrieving a packet and releasing it by the retriever

Dropped Packet Event

Packet was dropped because the buffer was full when the packet arrived.

3.1.2 Sending Data

The profiler has a *Sender Logger* auxiliary utility, which is responsible for serializing and sending the data aggregated by the profiler. The sending part is the bridge of communication between the **PipeRT Profiler** and the log analyzer so it was very important to consider the different possibilities before choosing the appropriate one.

Sending through sockets was the choice in sending and it is normal to use with such application, but the serialization was not as clear of a choice, there are various ways to serialize the data, we are going to discuss four of them, **binary sending**, **strings**, **XML** (Extensible Markup Language), and **protobuf** [1]. Each one of them has its pros and cons.

Turn the data into a string is costing building the string and also any changes will mean changes in the receiving side as well. The useful outtake of this option is the straightforward implementation.

Using XML as a format to communicate is a big burden on the performance and can lead to high memory usage, that said XML is a very popular format and a go-to option based on the project.

Protobuf or protocol buffers from *Google* is a work out of the box experience but will cost extra maintenance in the long run and can be slow.

The final decision was using **binary sending**, despite that choosing this will lead to always make sure that the **profiler** and **analyzer** are up to date with each other's sending and receiving changes, the performance will be better compared to the other possibilities, and this performance difference is an important key to minimize the delay hence having a better live analyzing which is the main goal for the **log analyzer**.

3.1.3 Packet serialization

The profiler sends packets of bytes, all the packets have the same structure, starts with *DGRP* as declaration of a new packet, null-terminated string of the receiver channel, *SEND*, a null-terminated string of the sender channel, arbitrary number of aggregated data starts with *LOGA*, a null-terminated string of event's type, four bytes integer for *log count*, four bytes integer for *time passed in microseconds*, eight bytes double for *minimum value*, eight bytes double for *maximum value*, and eight bytes double for *average value*.

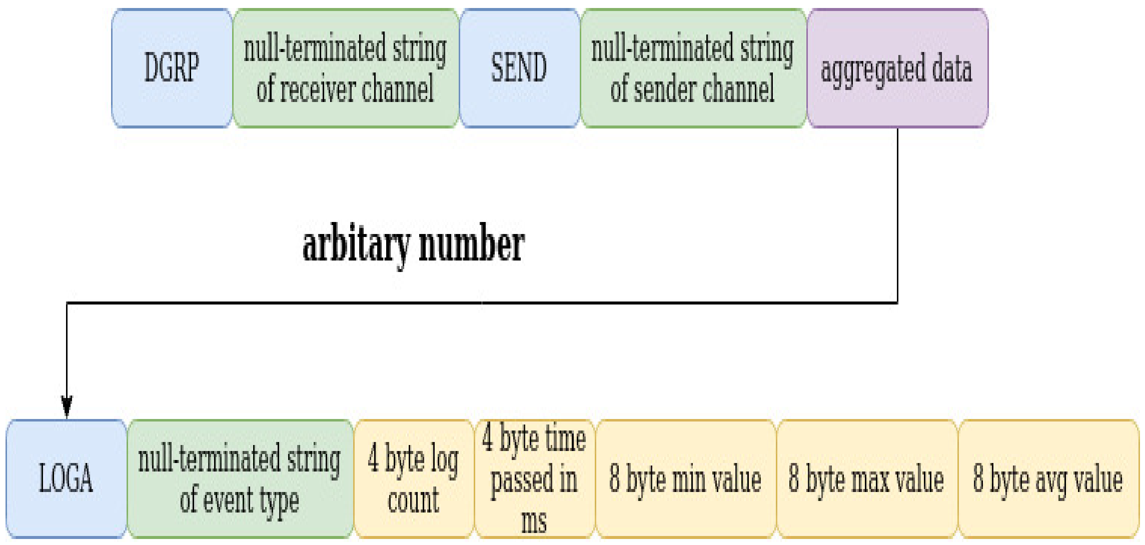


Figure 3.2: Packet's structure

sending the packets is the last step in the profiler's workflow (3.1) and from there, the log analyzer will start receiving and analyzing as we see in the following pages.

The log analyzer is separated into two main parts, as can be seen from 3.1. In the next sections, details about these parts, their different components, and the connection between them will be our topic.

3.2 The Analyzing Server

The first and the most vital part of the two, it is importance comes from its responsibility to hold the logic of the analyzing, reading configuration, sending the data to the **GUI server**. The coding of the **analyzing server** was done in *python3* and the language choice was because it is easy to use and fast to develop in which is suitable since it is expected from the user of the analyzer to extend it to match their need.

The *Front Controller Pattern* is the most used in the design of the **analyzing server**, and the *Front Controller Pattern* is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler which is very suitable to the various checkers and measurements.

AnalyzerServer is the heart class of the **analyzing server** so it is the main controller and responsible of starting and providing data for other controllers.

The first task for the **analyzing server** is to receive the packets from the profiler and decode it.

3.2.1 Receiving and Decoding Packets

Configuring the ip and port is crucial for successful receiving, the *socket* library in python is used to create the server and listen to the upcoming requests. The implementation of the server is inside the **AnalyzerServer** class and upon receiving the packets the class is starting running the other controllers.

The **PacketsManager** is the controller responsible for decoding the packets and distinguish them by giving each of them different id number, it has two main methods, **add** to decode and add new packet and **get_latest_packet** which is used as an input for another controller. The *sigleton* desgin pattern was used to implement the **PacketsManager** as to have single instance of it everywhere in the code.

```python
def add(self, packet):
  packet = PacketDecoder(packet).decode_packet()
  packet.set_id(self.__packets_count)
  packet.set_id_for_events()
  self.__latest_packet = packet
  self.__packets_count += 1

def get_latest_packet(self):
  return self.__latest_packet

def get_packet_count(self):
  return self.__packets_count
```

Code 3.1: puplic methods of PacketsManager

The packet is represented as **Packet** class, and it contains all the information that is sent in the packet from the profiler (3.1.3), and also has an id which is set by the **PacketsManager**. The packet object has a method to set id for the events in it, and this method is used by its controller as can be seen in 3.1.

The event is also represented as **Event** class and contains the data for the type, log count, passed time, minimum value, maximum value, and average value. The packet can have more than one event as mentioned in 3.1.1, and that is represented as a list of events objects in the packet object.

The **PacketDecoder** is the utility responsible for decoding and insinuating the packet object. But before going into details about how the decoding takes place, *endianness* should be discussed first.

Endianness is the order or sequence of bytes in computer memory. *Endianness* is primarily expressed as big-endian (BE) or little-endian (LE). A big-endian system stores the most significant byte at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least significant byte at the smallest address.

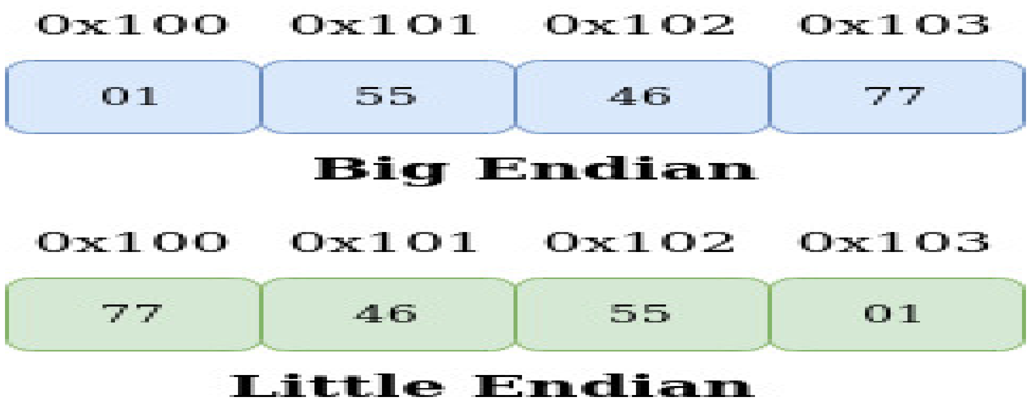


Figure 3.3: BE vs LE

Since the packets comes in bytes (3.1.2), the endianess of the bytes can form an obstacle while decoding, python has some built-in utilities to overcome that and these utilities were used in the decoding such as **decode** method for strings, **from_bytes** method for integers, and **unpack** method in **struct** data structure for double values.

```
def decode_packet(self):
  pos = 0
  if self.__check_for_correct_packet(self.__packet[pos:pos+4]):
    pos += 4
    receiver_channel_name, pos = self.__get_keyword(pos)
    sender_channel_name, pos = self.__get_keyword(pos)
    events = []
    correct_packet = self.__check_for_correct_packet(self.__packet[
       pos:pos+4])
    while not correct_packet and pos < len(self.__packet):
      event, pos = self.__get_event(pos)
      events.append(event)
    return Packet(receiver_channel_name, sender_channel_name,
       events)
  else:
    raise ValueError
```

Code 3.2: decoding of a packet

Code 3.2 is showing the **decode_packet** method in the **PacketDecoder** class, the algorithm is straightforward as it is following the bytes order as in figure 3.2.

```
def __get_keyword(self, pos):
  keyword = b""
  while self.__packet[pos] != b"\x00":
    keyword += self.__packet[pos]
    pos += 1
  return keyword.decode("utf-8"), pos+1

def __get_int_val(self, pos):
  ret = b""
  for i in range(0, 4):
    ret += self.__packet[pos + i]
  pos += 4
  ret = int.from_bytes(ret, byteorder="big")
  return ret, pos
```

```python

def __get_float_val(self, pos):
  ret = b""
  for i in range(0, 8):
    if byteorder == "little":
      ret += self.__packet[pos+(7-i)]
    else:
      ret += self.__packet[pos+i]
  [ret] = struct.unpack('d', ret)
  pos += 8
  return ret, pos
```

Code 3.3: helper methods in decoding

Code 3.3 shows the usage of the python decoding utilities mentioned before and how they can be used for different types.

The next step after decoding is managing the channels and assign the events to the correct channels.

3.2.2 Channels

Every measurement and checker depends on channels, so that makes them the part holding much information about the pipeline and the system performance. Each channel is represented by the **Channel** class, and the controller for these channels is the **ChannelsManager** class.

The **Channel** class is responsible for representing the channel so it contains basic fields for name and events. The class is also storing the different checkers' flags and measurements. Each is only storing events during the packet cycle as mentioned in 2.4.1, the last packet id that has been added to the channel is also stored as it is needed for measurements.

The **Channel** class is not only storing the measurements but also responsible for providing them to send them to the **GUI server**, and different approaches were used to find a suitable logic.

Checkers approach

Checkers are sent at the end of every packet cycle, and as a first approach the measurements were sent in the same way, but this leads to two issues, firstly, the **GUI server** has to update and redraw the graphs too frequently that in a case when the profiler is sending very frequently that browser might stop, secondly, not every measurement is worth of plotting, for example when having ten packet cycles with the same measurement.

Buffering approach

We encountered two obstacles and in this approach, the intendant was to slow down the sending of the measurements so that the browser can draw the graphs without overwhelming it. Instead of sending in every packet cycle, the measurements are going to be stored in lists, and once the **AnalyzerServer** is requesting the measurements the channel is going to send them, and empty the lists and this repeat every specific number of cycles. It is implemented to be ten cycles for now.

Buffering and RDB approach

The buffering approach was successfully able to deal with the first obstacle, but that leaves the problem of having unnecessary points in the graphs, to solve this issue, the *Ramer–Douglas–Peucker algorithm* was introduced in the implementation to reduce the number of plotted points. The *Ramer–Douglas–Peucker algorithm* also known as the **Douglas–Peucker algorithm** and **iterative end-point fit algorithm**, is an algorithm that decimates a curve composed of line segments to a similar curve with fewer points. It was one of the earliest successful algorithms developed for cartographic generalization. [2]

The **Ramer–Douglas–Peucker algorithm** has it is own optimized library in python [3], and it was used to reduce the number of measurements in the graphs.

```python
def reduce_points_n_extract_x_axis(points):
    minimized_points = rdp(points, epsilon=0.5)
    ret_points = [None] * 10
    for point in minimized_points:
        index = int((point[0] % 10) - 1)
        ret_points[index] = point[1]

    return ret_points
```

Code 3.4: using rdp to reduce points

The **Ramer–Douglas–Peucker algorithm** calculations to reduce the points are based on epsilon's value, and based on the choice of it, results can differ from each other on small scale, but for the analyzer use **0.5** is general enough for the use case so it was implemented as can be seen from 3.4.

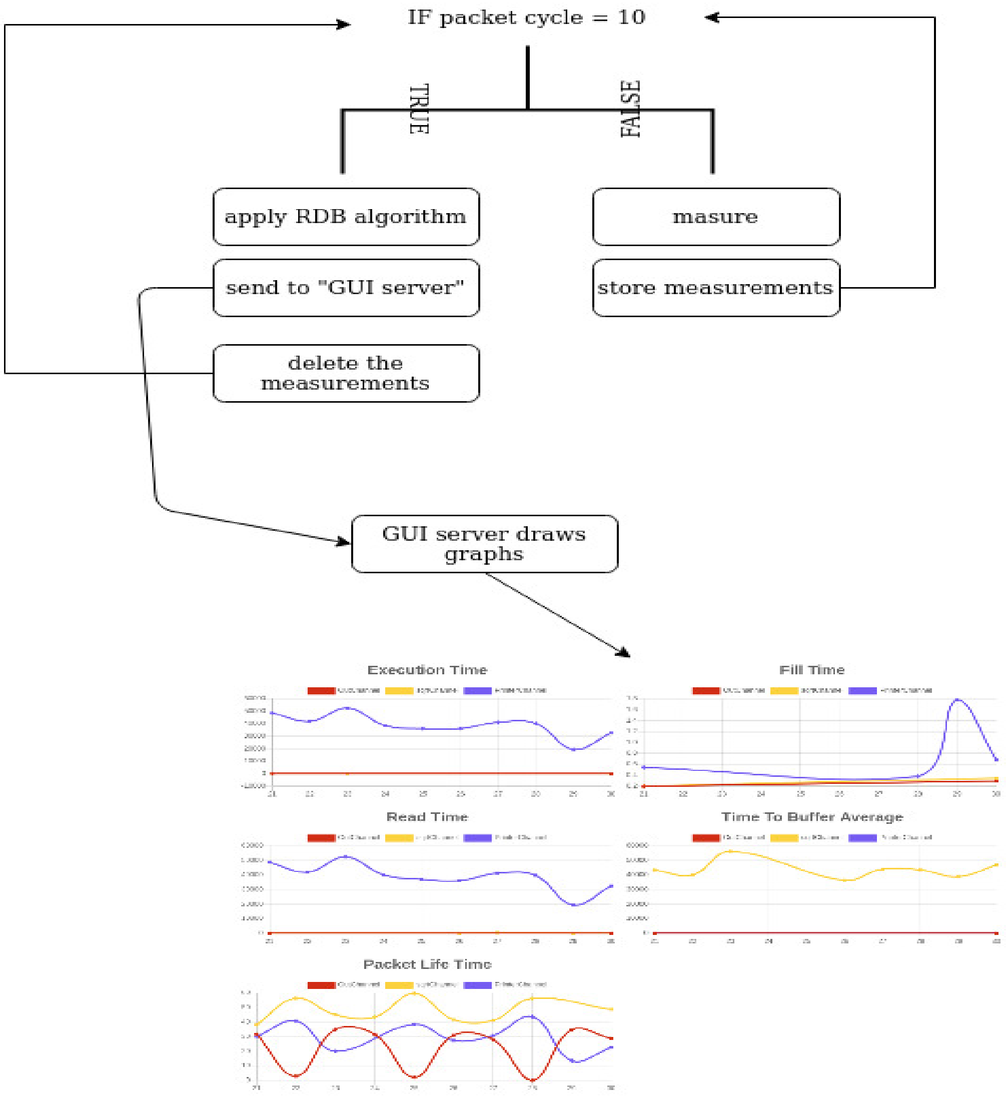


Figure 3.4: Measurements sending mechanism

The **ChannelsManager** is responsible for adding new channels, adding events for existent channels, and creating the channels' map. It also contains a list of the channels in the pipeline and has a getter method to provide this list for the measurements or the checkers. Similar to the **PacketsManager**, the **ChannelsManager** is also implemented using a singleton design pattern.

In some packets, the sender is N/A channel which means that the channel has been fed data from a different source other than the channels, for example, the first channel in the pipeline.

The **add_packet** public method of it is used in the **AnalyzerServer** and it is adding the receiver and the sender channels of the given packet. Adding the receiver is a matter of searching for the new channel name if it is not in the list, create a new channel and add the packet's events to it, if it is already in the list add the events of the packet to the existent channel.

```python
def add_packet(self, packet):
  receiver = packet.get_receiver()
  self.__add_reciever(receiver, packet.get_events(), packet.get_id
      ())

  sender = packet.get_sender()
  pushed_events_count = packet.get_event_count(PACKET_PUSHED)
  self.__add_to_channels_map(receiver, sender, pushed_events_count)

def __add_reciever(self, receiver_channel, events, packet_id):
for channel in self.__channels:
  if channel.get_name() == receiver_channel:
    channel.add_events(events, self.__packets_cycle_threshold)
    channel.set_latest_packet_id(packet_id)
    should_add_reciever = False
    return

c = Channel(receiver_channel, events, packet_id)
self.__channels.append(c)
```

Code 3.5: add receiving channel

Adding the sender channel is taking into consideration the fact that the sender might be a N/A channel, in order to determine if this a real N/A channel or the events in the packet did not need a sender, for example, **Read Time** (3.1.1) or **Retrieved** (3.1.1) events do not require a sender so the packet might have only these events so in this case the sender should not be added to the N/A channels. The only event currently which requires a sender is the **Pushed** event (3.1.1).

Creating the channels' map is important to have the visualization of the pipeline and also, it is a vital key for some measurements for example **time_to_buffer_average** measurement (2.5.1).

The **__add_to_channels_map** method is holding the logic of adding to the channels' map, it is checking if the sender is outside source, channel or there is no sender, and based on that it is adding to the channels' map or not. If the sender is an outside source or a channel, a tuple containing both of them will be added to the **__channels_map** list.

```
def __add_to_channels_map(self, receiver, sender, pushed_event_cnt)
    :
if not pushed_event_cnt:
  return
if sender == "N/A":
  sender = "External_" + receiver
  if sender not in self.__na_channels:
    self.__na_channels.append(sender)
else:
  channel_names = [c.get_name() for c in self.__channels]
  if sender not in channel_names:
    c = Channel(sender, [], -1)
    self.__channels.append(c)
sender_n_receiver = (sender, receiver)
if sender_n_receiver not in self.__channels_map:
  self.__channels_map.append(sender_n_receiver)
  self.__should_update_map = True
```

Code 3.6: add to channels' map

The **ChannelsManager** has two public methods related to the channels' map, **get_channels_map**, and **should_update_map**. The **get_channels_map** method is returning two lists, the unique channels' list and a list of tuples, each tuple represents a connection between two channels where the first element is the sender and the second is the receiver. for example, in figure 2.10, the unique channels' list was equal to [OutChannel, sqrtChannel, PrinterChannel] and the list of connection was [(2,1),(1,0)]. The **should_update_map** method is implementing to send only in case a change happens in the map and this preserves the resources taken to visualize the pipeline.

```python
def get_channels_map(self):
  channels_names = [c.get_name() for c in self.__channels]
  unique_channels = channels_names + self.__na_channels
  channels_dict = {c: i for i, c in enumerate(unique_channels)}
  connections = [(channels_dict[s], channels_dict[r]) for (s, r) in
            self.__channels_map]
  return unique_channels, connections

def should_update_map(self):
  val = self.__should_update_map
  self.__should_update_map = False

  return val
```

Code 3.7: public methods for channels' map

The channel contains the measurements and the checkers' flags so to send these data to the **GUI server**, two public methods were implemented in the **ChannelsManager** and they are responsible for grouping the data into dictionaries so it can be sent through http post request in a json format.

```python
def get_channels_flags(self):
  return [{"name": channel.get_name(),
      "flags": channel.get_flags()}
      for channel in self.__channels]

```

```python
def get_channels_measures(self):
  return [{"name": channel.get_name(),
      "measures": channel.get_all_measures()}
      for channel in self.__channels]
```

Code 3.8: data grouping methods

Chapter 4

Összegzés

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In eu egestas mauris. Quisque nisl elit, varius in erat eu, dictum commodo lorem. Sed commodo libero et sem laoreet consectetur. Fusce ligula arcu, vestibulum et sodales vel, venenatis at velit. Aliquam erat volutpat. Proin condimentum accumsan velit id hendrerit. Cras egestas arcu quis felis placerat, ut sodales velit malesuada. Maecenas et turpis eu turpis placerat euismod. Maecenas a urna viverra, scelerisque nibh ut, malesuada ex.

Aliquam suscipit dignissim tempor. Praesent tortor libero, feugiat et tellus porttitor, malesuada eleifend felis. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nullam eleifend imperdiet lorem, sit amet imperdiet metus pellentesque vitae. Donec nec ligula urna. Aliquam bibendum tempor diam, sed lacinia eros dapibus id. Donec sed vehicula turpis. Aliquam hendrerit sed nulla vitae convallis. Etiam libero quam, pharetra ac est nec, sodales placerat augue. Praesent eu consequat purus.

Appendix A

Szimulációs eredmények

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque facilisis in nibh auctor molestie. Donec porta tortor mauris. Cras in lacus in purus ultricies blandit. Proin dolor erat, pulvinar posuere orci ac, eleifend ultrices libero. Donec elementum et elit a ullamcorper. Nunc tincidunt, lorem et consectetur tincidunt, ante sapien scelerisque neque, eu bibendum felis augue non est. Maecenas nibh arcu, ultrices et libero id, egestas tempus mauris. Etiam iaculis dui nec augue venenatis, fermentum posuere justo congue. Nullam sit amet porttitor sem, at porttitor augue. Proin bibendum justo at ornare efficitur. Donec tempor turpis ligula, vitae viverra felis finibus eu. Curabitur sed libero ac urna condimentum gravida. Donec tincidunt neque sit amet neque luctus auctor vel eget tortor. Integer dignissim, urna ut lobortis volutpat, justo nunc convallis diam, sit amet vulputate erat eros eu velit. Mauris porttitor dictum ante, commodo facilisis ex suscipit sed.

Sed egestas dapibus nisl, vitae fringilla justo. Donec eget condimentum lectus, molestie mattis nunc. Nulla ac faucibus dui. Nullam a congue erat. Ut accumsan sed sapien quis porttitor. Ut pellentesque, est ac posuere pulvinar, tortor mauris fermentum nulla, sit amet fringilla sapien sapien quis velit. Integer accumsan placerat lorem, eu aliquam urna consectetur eget. In ligula orci, dignissim sed consequat ac, porta at metus. Phasellus ipsum tellus, molestie ut lacus tempus, rutrum convallis elit. Suspendisse arcu orci, luctus vitae ultricies quis, bibendum sed elit. Vivamus at sem maximus leo placerat gravida semper vel mi. Etiam hendrerit sed massa ut lacinia. Morbi varius libero odio, sit amet auctor nunc interdum sit amet.

Aenean non mauris accumsan, rutrum nisi non, porttitor enim. Maecenas vel tortor ex. Proin vulputate tellus luctus egestas fermentum. In nec lobortis risus, sit amet tincidunt purus. Nam id turpis venenatis, vehicula nisl sed, ultricies nibh. Suspendisse in libero nec nisi tempor vestibulum. Integer eu dui congue enim venenatis lobortis. Donec sed elementum nunc. Nulla facilisi. Maecenas cursus id lorem et finibus. Sed fermentum molestie erat, nec tempor lorem facilisis cursus. In vel nulla id orci fringilla facilisis. Cras non bibendum odio, ac vestibulum ex. Donec turpis urna, tincidunt ut mi eu, finibus facilisis lorem. Praesent posuere nisl nec dui accumsan, sed interdum odio malesuada.

Bibliography

[1] Google Inc. *Protocol Buffers (a.k.a., protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data.* URL: `https://github.com/protocolbuffers/protobuf`.

[2] Wikipedia. *Ramer–Douglas–Peucker algorithm.* URL: `https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm`.

[3] Fabian Hirschmann. *Ramer–Douglas–Peucker algorithm Library in python.* URL: `https://rdp.readthedocs.io/en/latest/`.

List of Figures

List of Tables

List of Codes