Roman Varnakov (2116364)

Developing a communication layer between Python and JavaScript

Bachelor's thesis

Bachelor of IT and Engineering

Information Technology

2025

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author	Roman Varnakov
Thesis title	Developing a communication layer between Python and JavaScript
Commissioned by	LightningChart Oy
Year	2025
Pages	43 pages
Supervisor	Ville Kauppi

ABSTRACT

The study focused on the development of a two-way communication system for Python and JavaScript programming languages in situations where JavaScript runtime responses to requests sent by Python runtime. The objective of the study was to create a socket-based communication module that facilitates interaction between a Python client and a JavaScript client, leveraging Flask as the server framework and SocketIO for communication.

The study leveraged various methods to create the system and evaluate it. As a structure, class-based approach was introduced. For serialization, were used JSON, MessagePack and Numpy. Evaluation methods involved simulated request-response scenarios, latency measurements under different payload sizes with a manual logging method. Wireshark was used to capture network traffic and analyze it.

As a result, a complete prototype was developed and tested. The system demonstrated reliable performance, even if there were certain improvement considerations, efficient data exchange with a combination of JSON and zstd, and stability under load, confirming its suitability for two-way communication between Python and JavaScript environments.

Keywords: API, networking, visualization, communication

CONTENTS

1 INTRODUCTION

The modern, technologically advanced era brings a range of IT tools, offering numerous approaches to developing innovative technological solutions. As the variety of these solutions continues to rapidly grow, the demand for efficient, scalable, and consistent solutions is also increasing. This leads to the fact that a great focus must be placed on development processes, communication methods, and integration techniques.

In this thesis, one is explored: enabling real-time, bidirectional communication between Python and JavaScript (JS) programming languages, when Python should leverage a JS library for data visualization. In order to address this, a network socket-based approach was implemented and tested.

It is essential to establish the study on a coherent theoretical background, but it is not less important to address the practical aspect correspondingly. Therefore, the study aims to explain the development of a functional prototype and provide comprehensive analysis of its performance. The study also describes the process of ensuring system consistency and explains the features of efficient optimization.

1.1 Background and motivation

During the initial phase of familiarization with the commissioning company workflow, infrastructure, products, and operations, it was eventually decided to proceed with the further development of the Python LightningChart visualization library (LCPY or LC). The topic was selected, and the objectives of the study were defined.

However, progress was stalled at the early stages for approximately a week due to an insufficient use of functions instead of classes. Python Team Lead provided a foundational object-oriented base structure for the prototype. For this I am very grateful, as it significantly accelerated the development process and saved time.

In effect, this thesis is focused on the functionality development, rather than abstract code structure.

1.2 Objective

The initial aim of the study, in general, was to consider improvement for the existing one-way communication which is only capable of sending requests. Therefore, a two-way communication layer, which can also receive responses, was developed.

The study objective can be structured as follows:

- Develop a communication layer
    - Collect useful information
    - Plan the prototype
    - Create the prototype
- Test the solution
    - Ensure functionality
    - Find bugs and debug
    - Benchmark the prototype
- Analyze the results
    - Highlight the pros and cons
    - Discuss results

In sum, the study aims to highlight one of the possible communication solutions for a specific real-word scenario.

2 THEORETICAL BACKGROUND

This section provides the conceptual basis for the development of the prototype. The communication between programming languages, networking protocols, and the use of the WebSocket protocol will be explained. Furthermore, the concept of LCPY library will be introduced, as initially it is a commercial product and one of the key elements of the prototype functionality and purpose. Among all possible

scenarios of establishing communication between programming languages and processes, many definitives could be collected, while the system is limited to one network instance, localhost. Additionally, system locality is the key reason why security concerns were neglected in general. Even if it would be possible to leverage other than over-the-internet connection techniques, a networking data transfer was selected to ensure minimum set-up requirements.

This section aims to present a comprehensive description of the communication mechanisms and technologies that were used the study. Instead of comparing all possible communication methods, only the one that was selected is described in detail. It is essential to mention that the study is concentrated on a complete prototype. Many different development scenarios are possible, starting from choosing the in-process or inter-process system structure and ending with data compression mechanisms, the primary focus of this thesis will be on aspects of prototype development, specifically networking, serialization, and server-client communication.

2.1 Python and JavaScript programming languages

Python and JavaScript were primarily utilized during the prototype development. These programming languages were selected due to their ability to support a wide range of functionalities, ease of integration, and extensive library support. Both are among the most widely used tools for programming (Statista 2019).

Python programming language

Python is an object-oriented, interpreted, and interactive programming language (Python 2013). It is further described in Python docs as high-level programming language with built-in data structures and good readability (Python 2025). Python Software Foundation is the organization responsible for supporting Python.

Further, Python plays a significant role in the prototype as a backend. It is responsible for allowing server initialization, maintenance during library utilization and communication logic over runtime. Python does not only send and receive

requests but also serves an important purpose in data manipulation. This includes tasks such as data serialization and compression for efficient processing and communication.

JavaScript programming language

JavaScript, or JS, is a lightweight interpreted, or just-in-time compiled (JIT) high-level programming language (MDN 2025). In the MDN web docs, JIT is described as a compilation process which allows code to be converted to machine code at runtime (MDN 2024). This is usually used in modern web browsers, optimizing the JavaScript code performance. MDN is an open-source, collaborative project owned and developed by Mozilla Corporation.

Eventually, JS side is responsible for receiving data from Python and sending data back if requested. Once data is received, it is processed inside code and with LightningChart JS library. As a result, a chart with data points should appear and title name returned to Python, according to the request.

2.2 Communication between programming languages

Programming languages have their own strengths and specified use cases. In general, every programming language can be described as low-level or high-level, with either backend or frontend orientation. Python is good for scripting, data processing and backend development. JavaScript is a high-level programming language, and it is primarily used for interactive web frontends. A combination of the capabilities of different languages would lead to a complex system which might offer a variance of possibilities in future development (Daily Dev 2024).

In this study, communication between JavaScript and Python was achieved with Flask which is a web application framework for Python. A chart can be visualized in the web browser with JS client according to requests from Python client, networking-based communication, as in current LCPY library. The process of communication is described in section 4.1.

2.2.1 Overview of inter-language communication

As it proved to be difficult to enable a seamless connection between Python and JavaScript, inter-language communication became a crucial aspect in the study to consider. Originally, in this study the concept of inter-language communication was derived from the concept of programming language interoperability which is the multiple programming languages' ability of communication inside the same software system (Nguyen 2022). In fact, communication will be performed on the basis of local network data exchange. This is due to the objective of developing a solution that relies solely on a single Python library (module). Because the solution is intended to use only one LCPY library, other potential methods that require additional modules or tools are not applicable and, therefore, not considered in this study.

In the context of this study, inter-language communication can be described as the process of transferring and recognizing data between programming languages. That is, when Python runtime sends a request, JS should catch and process this request. Efficient communication requires consideration for data serialization, transport mechanisms and operation over data to make it recognizable for any programming language. These aspects are addressed below.

Data serialization

Serialization plays an important role in high-speed data transmission.
Before data is sent from one point to another, it is mandatory to convert it to a readable format. This can be achieved by common formats such as JSON, XML and MessagePack. For the prototype in its current stage, only JSON and msgpack were used.

However, the initial purpose of serialization is to convert data to a byte string to make sure the compression mechanism functions properly. While serialization is

required for a compression process to complete, communication can be established by using raw JSON format only.

JSON

JSON is a lightweight, human readable and machine efficient data-interchange format. JSON stands for JavaScript Object Notation and is considered data-interchangeable due to language independence. (JSON 2023). As can be seen from the Figure 1 example, this format is both human and machine readable. As for the packet capture, the message is also in JSON format (Figure 1).

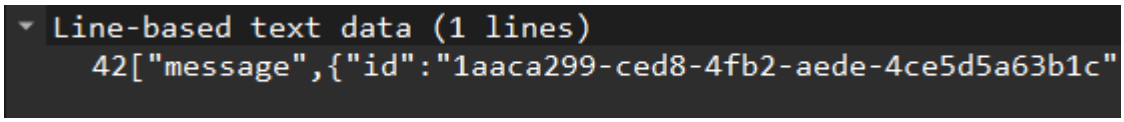

Figure 1. Line-based data with WebSocket code identifier, Wireshark packet capture screenshot

During the development of JSON-based communication, the chosen technologies demonstrated excellent performance, particularly in terms of compression efficiency when used with the Zstandard (zstd) compression library. It is also important to note that all data formatting in this system is initially based on the JSON format. However, JSON is not the only data format that allows for data serialization, which is demonstrated by the use of json.dumps(data).encode("utf-8").

Moreover, during the development process, it was found that compression can be applied to the data only if serialization is used. In other words, before compression, it is required to convert data to the byte string format. Notably, once JSON serialisation with binary encoding is applied, the message is marked with ‘b’ at the start which means binary (Figure 2). In the network, on the other hand, it has an actual binary format. (Figure 3.)

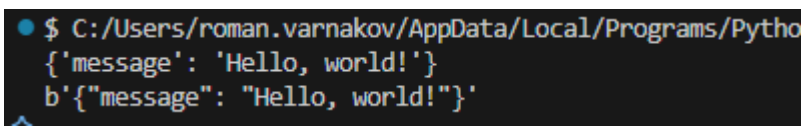

Figure 2. Message before JSON encoding and after, VScode console

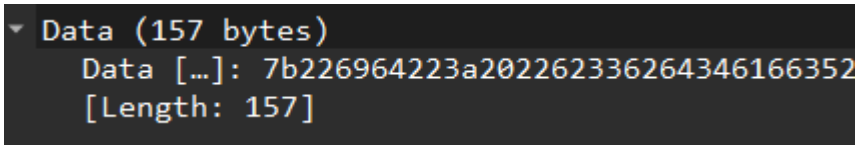

Figure 3. JSON byte string data, Wireshark packet capture

Notably, Wireshark was able to translate the data inside packet to human readable format, whereas on the JavaScript side, a decoding algorithm must be implemented to parse the data correctly, whether using JSON or MessagePack. The decoding mechanism inside Wireshark can recognize utf-8 data as presented in Figure 4.

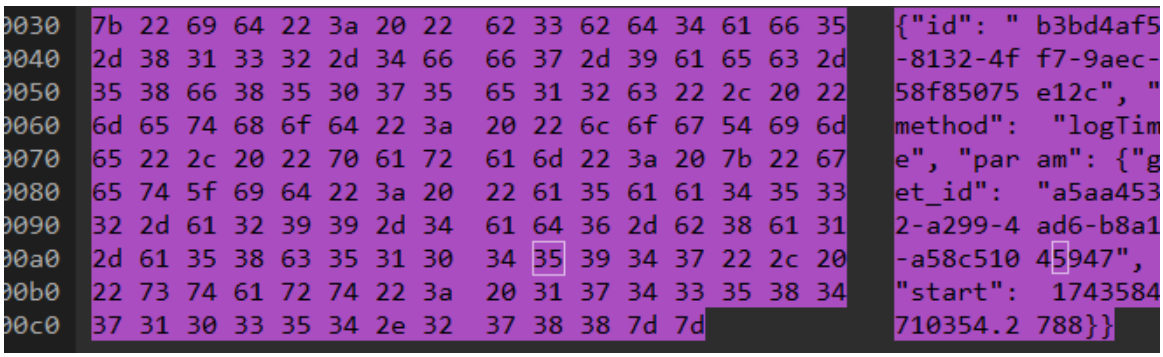

Figure 4. Byte string data (to the left) and JSON decoded format (to the right). Wireshark packet capture

On the other hand, it was not possible to use JSON serialization method to convert NumPy arrays which are multi-dimensional, high-performance arrays used extensively for numerical computations in Python (NumPy 2025). Since NumPy arrays are not directly JSON serializable, he use of JSON which would allow efficient data exchange prevented. MessagePack proved to be more suitable for this purpose.

MessagePack

MessagePack (or msgpack) is a format similar to JSON, but it is smaller and faster (Msgpack 2017). It shows high efficiency with a structured data serialization in the prototype, it was used to serialize JSON formatted messages. The serialization of JSON message with msgpack is illustrated in Figure 5. Figure 6 shows an example of packet capture data encoded in UTF-8 alongside its decoded representation. As can be seen in Figure 7, Wireshark is not able to translate MessagePack back to human readable JSON format.

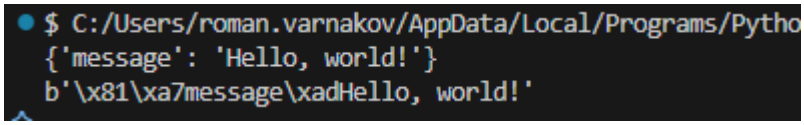

Figure 5. Message before MessagePack encoding and after. VScode console

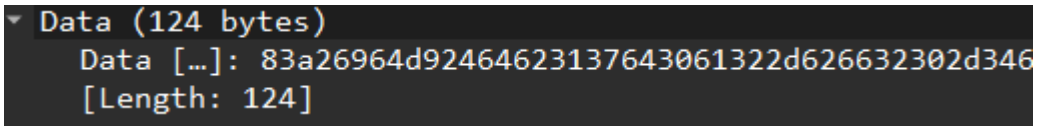

Figure 6. MessagePack byte string data. Wireshark packet capture

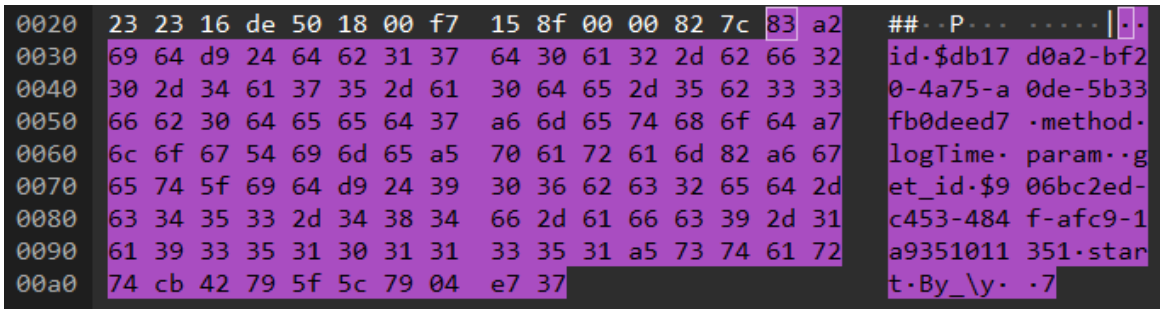

Figure 7. Byte string data to the left and partially recognized data to the right. Wireshark packet capture

Msgpack has a wide range of methods, including NumPy packing which was used for serializing a NumPy array in one of the prototype variations. In general, NumPy array is used for achieving better performance results. Once a normal array of numbers is converted to NumPy array with defined data structure, overall data processing performance is expected to improve. Efficient array can be then serialized with msgpack, or other serialization libraries or methods supporting NumPy-type data serialization.

Protocol Buffers

Protocol Buffers are an extensible mechanism for structured data serialization. They are language and platform-neutral, developed by Google and aimed to support customized serialization for formatted data technology. (Google 2025). According to observations made in this study, this library eventually gives a flexible serialization process configuration in exchange of complexity.

However, the potential of this mechanism remains unproven, to provide a modular data serialization for all possible data requests. Usually, speed delays are caused by a vast quantity of data points, so it might be possible to reach high performance by achieving a synergy with effective NumPy arrays and compression tools.

FlatBuffers

FlatBuffer is another serialization library format by Google based on customized serialization library format. FlatBuffers library provides access to serialized data without parsing or unpacking. It is memory efficient, and backwards or forwards compatibility with a small footprint (Google 2025). FlatBuffers and Protocol Buffers are similar in essence.

Summing up this section, it must be said that there exist many possible formats. It is important to remember that format choice is only a small part in the development process and its effect varies, depending on the combination in which it is used other compression mechanisms.

Finally, even if Protocol Buffers and FlatBuffers were not applied in this study, they seem efficient. However, they might even improve the performance of the prototype solution, but this might also raise maintenance costs.

2.2.2 Communication challenges between Python and JavaScript

This section describes both general and communication challenges associated with the prototype. In the prototyping phase, were used technologies that support two-way communication, such as Flask-SocketIO. These are further discussed in the 3 and 4 sections.

According to observations made in the study, common challenges in multi-language communication include:

1. Data inconsistenc
2. Data type incompatibility
3. Selecting Asynchronous or Synchronous Processing
4. Security concerns

Ensuring data consistency has always been a sensitive matter in software development. It is relatively reliable and secure to ensure data quality in one test environment, but the task becomes difficult when multiple software systems are used and, consequently, the error rate grows. There are many ways of ensuring data consistency, ranging from simple logging messages inside programming languages to external tools analysis such as Wireshark for network traffic (CompTIA 2025).

Data type incompatibility is a challenge that relates to differences between systems or components in how they represent and process data. Since languages are based on different syntax, compatibility issues arise between them. First, communication is established by agreeing on some language-neutral communication mechanism. Second, performance must be considered. Asynchronous or synchronous processes play an important role in communication and the enabling or disabling of modules with asynchrony creates a significant effect on the software system functionality. This was a key consideration during development. For example, blocking a response request may halt system processes, while request threading allows the processing of requests sequentially rather than combining them into a single call. The study addresses most aspects of the communication plan but omits the aspect of

security. If communication is carried out within a local network and is limited to loopback traffic on a single device, external threats such as packet injection or man-in-the-middle attacks are considered unlikely. The main challenges were faced in relation to communication between Python and JS. In short, data processing and communication performance are critical challenges.

2.3 Networking and client-server communication

This section outlines the general aspects of network communication, concluding with a description of the client-server architecture used in the prototype. Definitions of the terms client and server are provided from a networking perspective and later adapted to the specific structure of the prototype.

In networking, a client is typically the process that initiates communication, while a server is the process that waits to be contacted and responds to incoming requests (Kurose & Ross 2013). This classical model supports the communication flow in many systems, including the prototype's structure (described in section 4.1).

However, the prototype is constrained due to its local architecture where all components run on the same machine. Although it follows a client-server communication model, the roles are not strictly separated. The Python process starts a server which then launches the JavaScript process. The server process incoming requests and responses, while each client carries out its specific tasks: the Python client manages data, and the JavaScript client manages visualization.

The server is initiated through instructions defined in the LCPY library which specifies the server's starting procedure and the functions called, data formats and compression mechanisms to be used. Once active, the server facilitates communication between clients by processing requests in both directions. This local communication relies on networking technologies such as Flask and Flask-SocketIO which are described in the following sections. Loopback communication, which underpins the system's security assumptions, is also explained.

2.3.1 Protocol

In networking, a protocol is a predefined set of rules that governs the transmission of data between connected devices, regardless of their type or function (CompTIA 2019). For example, a phone can send information to a website on a PC, and the PC can transfer data to a television. As long as devices comply with the same protocol, communication is possible. In the prototype, the main protocols are TCP and HTTP. Several other protocols are also discussed in this section.

IPv4

IPv4 (Internet Protocol version 4) is a network-layer protocol responsible for addressing and routing packets across networks, whether over the internet or a local area network. Unlike TCP, which is a transport-layer protocol, IPv4 does not need to verify data delivery and is, therefore, considered connectionless (ClouDNS 2025).

HTTP

HTTP (Hypertext Transfer Protocol) is used to transmit hypermedia documents, such as HTML (Hyper Text Markup Language). It operates on a traditional client-server model in which the client initiates a connection and requests resources from the server (MDN 2025). In the prototype, HTTP defines how the Python client makes requests for the data that the JavaScript component visualizes.

HTTPS

HTTPS (Hypertext Transfer Protocol Secure) is the encrypted version of HTTP. It uses TLS (Transport Layer Security), previously known as SSL, to secure communication between client and server by encrypting data in transit (MDN 2025).

TCP

TCP (Transmission Control Protocol) is a transport-layer protocol that ensures reliable, connection-oriented data exchange between applications. Unlike the connectionless UDP protocol, TCP confirms that messages are delivered in sequence and without loss (Kurose & Ross 2013). In this study, TCP is primarily used for loopback communication. It was determined that the use of UDP would not significantly improve performance of data transmission for large datasets due to the associated risk of data loss.

WebSocket

WebSocket is a real-time communication protocol that enables full-duplex, bidirectional communication between a client and server over a single, persistent TCP connection. It allows continuous interaction for as long as needed, making it suitable for real-time data exchange between the Python backend and the JavaScript frontend (Ably 2024). In the prototype, WebSocket is implemented using a Flask server in combination with SocketIO to ensure efficient communication.

Flask-SocketIO

Flask-SocketIO is a technology existing under Flask's maintenance. It is described as an extension to Flask applications which gives access to low latency bi-directional communication between the clients and the server. SocketIO helps to enable multi-language communication it is supported as a third-party library for C++, Java and Swift, including JS and Python programming languages, allowing clients to establish a permanent connection to the server (Miguel Grinberg 2018).

2.3.2 Loopback and network hierarchy

Loopback refers to a communication method in which the client and server operate on the same device. This allows data to be sent and received internally without leaving the host system (Lenovo 2021). This was extensively analyzed using Wireshark in this study. All communications between Python and JavaScript occurs within loopback traffic.

Figure 8 illustrates protocol-level traffic analysis. Loopback traffic is segmented into internet protocol versions, with communication primarily based on TCP. WebSocket accounted for 96.5% of the total data volume transmitted (Figure 8.)

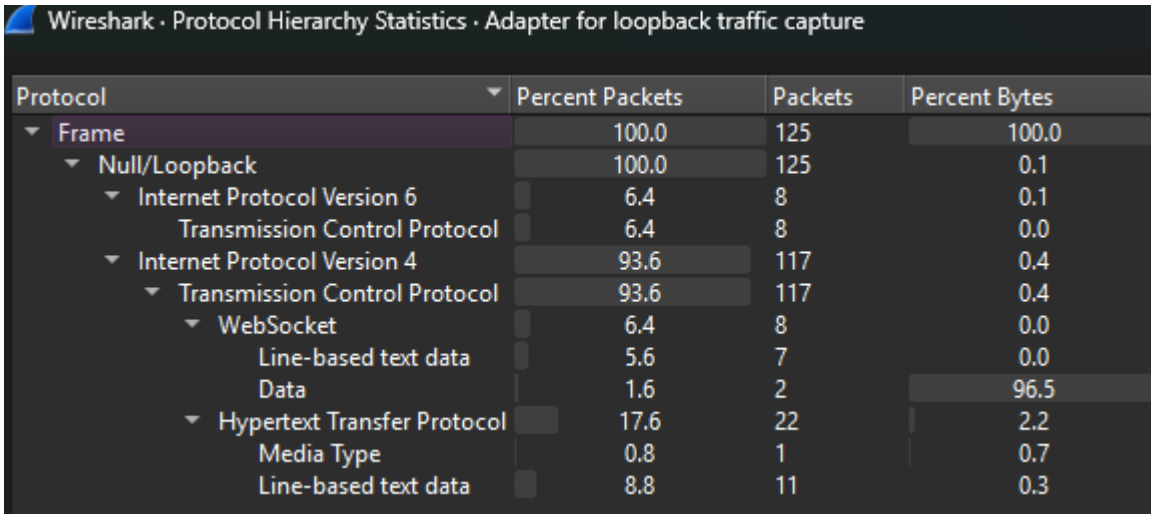

Figure 8. 1.000.000 data points statistics. Statistics tab → Protocol hiearchy. Wireshark packet capture

2.4 LightningChart library

LightningChart is a high-performance, GPU-accelerated charting framework, originally developed for .NET in 2009. It was the first charting library to process graphics entirely on the GPU. As of the beta release in summer 2024, the library also supports Python, in addition to several other programming languages (LightningChart 2025). However, in the Python version of the library, a slight performance decrease occurs at the application level. This is primarily due to the overhead introduced by data serialization and transport between components.

Out of more than 25 available chart types, Chart XY and Line Chart were selected and adapted for use in the prototype's communication architecture. Noticeably, LightningChart library has nearly 300 interfaces, almost 150 classes, over 20 unique enumerators, more than 120 Type of Aliases, 52 Variables and 60 Functions. This is another reason for developing a communication layer for a JS library version, as creating a Python-only version of the library would have required unreasonable effort.

3 METHODS

This section introduces the methods used in the development of the prototype. The approach involves both development and testing tools. The section begins with an overview of the development environment, including hardware, software, and programming languages.

3.1 General methods

This subsection describes the general methods employed in the development and testing of the prototype. It covers the software development process, including the techniques and tools used for coding, debugging, and version control. Testing approaches, including test cases, test automation, and validation methods, are discussed in detail in the following section.

3.1.1 Hardware

Development PC

The development environment was set up on a Windows 11 Pro system (version 24H2), equipped with 32 GB of RAM, providing ample memory for handling large datasets and running multiple applications simultaneously. The AMD Ryzen 5 2600, a six-core processor with a base clock speed of 3.40 GHz, offered sufficient processing power for the demands of software development and testing

tasks. For graphical processing, the system is equipped with an NVIDIA GeForce GTX 1050 Ti, which, while not the latest model, provided reliable performance for visualizations and debugging during development.

Network Configuration

The development machine was connected to the network via an Ethernet cable, using a Realtek PCIe GbE Family Controller. This setup ensures a stable and high-speed wired connection, with a link speed of 1000/1000 Mbps for both receiving and transmitting data. During testing, the system achieved a tested speed of 200/125 Mbps for receiving and transmitting data, respectively. These speeds were sufficient for real-time communication and data exchange between the client and server.

3.1.2 Software

Web Browser

The web browser used for testing and running the prototype was Google Chrome, specifically version 135.0.7049.42 (64-bit), running on a Windows 11 (64-bit) platform. The browser's user agent string indicates compatibility with the latest web standards, ensuring optimal rendering and functionality of the prototype’s front-end components. The user agent, Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/135.0.7049.42 Safari/537.36, confirms that Chrome was able to run the required JavaScript and display dynamic visualizations without issues.

Code Editor

For development, Visual Studio Code (VS Code) was the primary code editor, running version 1.98.2 on the Windows 11 (64-bit) platform. VS Code provides a versatile environment for writing and debugging code, offering features like IntelliSense, integrated Git support, and a rich ecosystem of extensions that

helped streamline development. Its lightweight nature and powerful features made it an ideal choice for both Python and JavaScript development during the project.

Development Languages

The prototype was primarily developed using Python 3.11.9 for backend processes and JavaScript (version 1.5) for the front-end logic and interaction. In addition to these, other technologies such as HTML for the structure of the web pages and JSON for data interchange were employed. For data compression, zstd was used to optimize performance. This combination of languages and tools enabled the seamless development of both the server-side and client-side components of the system.

Visualization Library

To handle the data visualization, the LightningChart JS library (version 7.0.3) was utilized. This library's core methods, such as ChartXY() for creating two-dimensional charts and addLineSeries() for adding data series to the chart, were key components in rendering dynamic and interactive visualizations. LightningChart provided high-performance rendering for real-time data, which was essential for the visualization of server-client communication in the prototype.

3.2 Testing methods

Logging library and Wireshark packet tools were used as testing tools. These tools provide a valuable information and be used for an analysis and of the collected data. In addition, print() and console.log() methods were used as means of information retrieval during the development phase. Even if they can be considered secondary debugging tools, they are still important instruments.

Wireshark

Wireshark is a network packet analyzer. A packet refers to a discrete data unit. It is widely used in networking (CompTIA 2024). In this study, Wireshark was used to collect key results related to network traffic and communication patterns.

Logging

Logging is the process of recording information during program execution such as the origin of errors (Fahim 2025). During the prototype development, it was actively used to collect information from Flask server.

Examples from Flask-SocketIO and the built-in Python logging library:

1. socketio = SocketIO(logger=True, engineio_logger=True)
2. logging.basicConfig(level=logging.DEBUG)

In the first example, the socketio variable contains configuration options that enable logging for both SocketIO and Engine.IO components. By setting logger=True and engineio_logger=True, detailed logs of the server's activities such as incoming and outgoing messages, as well as connection events, are captured. This facilitates effective debugging and real-time monitoring of server-client communication. In the second example, the logging configuration sets the logging level to DEBUG, ensuring that detailed log output is generated, which can help identify issues or performance bottlenecks during communication.

print()

print() is a built-in Python function used to display the current value or type of an object, such as a variable. In this thesis, it is considered as a lightweight logging tool and it frequently used during development to monitor output through the console. Simply placing a variable inside parentheses will display its value when printed.

console.log()

Like print() in Python, console.log() is a logging tool in JavaScript. It was helpful during the development process by printing useful information in the browser console. Writing a variable inside parentheses will print its value.

4 PROTOTYPE

This part is completed by applying findings acquired during hands-on prototyping, methodology and information reflection. Main aim is to describe a solution for a two-way communication between Python and JS. In this section the solution structure and general idea are established and the study objective is completed.

4.1 Prototype structure

In this subsection, theoretical and planning aspects of the prototype are highlighted. It explains how the selected technologies were applied and outlines the expected results. Communication sketch is shown under Figure 11.

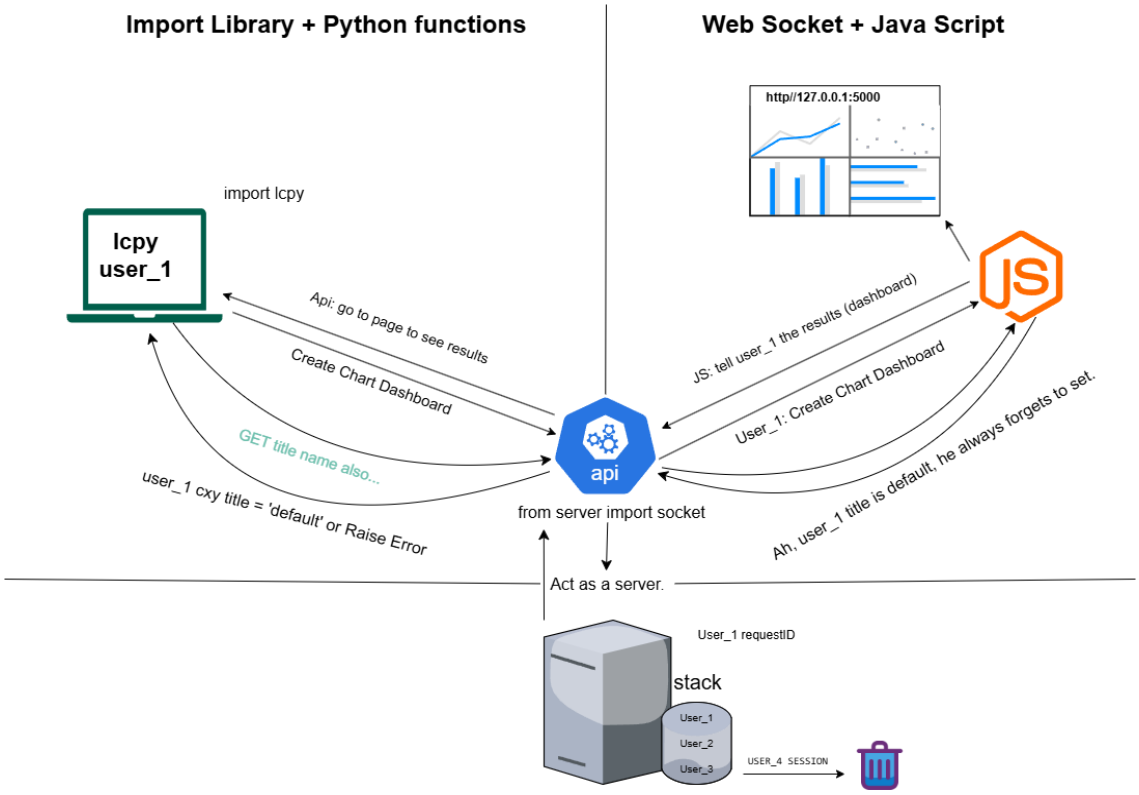

Figure 9. Prototype communication schema. Draw.io

The Python runtime sends requests to an API, which transmits them to the JavaScript client and vice versa. The JavaScript client responds if a conditional GET parameter is specified in the request header. These methods are inspired by REST API practices, but naming conventions are used informally to clarify function behaviour.

Consider the case of a single chart. The goal is not only to generate a chart upon request, but also to maintain the chart state after a page refresh or when opening multiple identical pages. In any case, only one chart should be displayed as requested (Figure 12.)

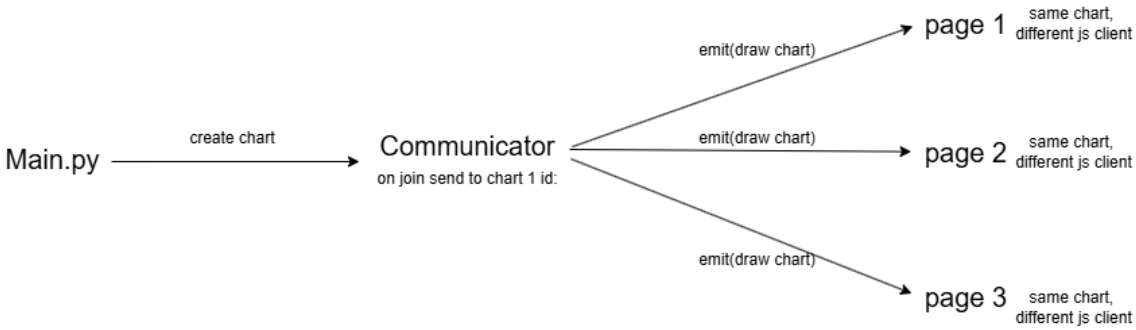

Figure 10. Expected communication behaviour. Draw.io

After defining the prototype plan and communication principles, a working solution was developed. It is important to note that the communication sketch was prepared to outline core concepts before the prototype implementation. In the result, the API and server components were merged for simplicity. However, functionality remained unchanged.

4.2 Solution

In this core section of the study, the most significant components of the prototype are highlighted. This part also presents the key functionalities of the software system from both the Python and JavaScript perspectives. It includes an overview of the Flask server, the LCPY client library, as well as the front-end components developed using JavaScript and HTML.

4.2.1 Python

This subsection explains the roles of the Python server and client. The client functions as a library that defines classes and request structures. The server, serving as the backend, provides the foundation for managing connections. In the prototype, the server uses a SocketIO connection implemented with the Flask framework.

Server

The server is defined within the Server class. It includes various variables such as the Flask app instance and an app route (Figure 13). Flask app definition and route specification. The specified route uses a forward slash (/), which triggers the index function when accessed (Figure 14.)

```
24  class Server:
25      def __init__(self):
26          self.room_id = str(uuid.uuid4())
27          self.storage = []
28          self.server_is_open = False
29          self.port = None
30          self.socketio = None
31          self.received_data_storage = {}
32          self.app = Flask(
33              __name__,
34              static_folder=os.path.join("static"),
35              template_folder=os.path.join("templates"),
36          )
37          self.app.config["SECRET_KEY"] = "secret!"
38          self.socketio = SocketIO(self.app, async_mode="gevent", ping_timeout=100)
39          self.server_thread = None
40          self.app.route("/")(self.index)
```

Figure 11. Server class. Python server file

```
44      def index(self):
45          room = request.args.get("room")
46          return render_template("index.html", room=room)
```

Figure 12. Server index function. Python server file

Once the index function is triggered, it returns the index.html file as the template. This file acts as the default HTML document and includes references to the required JavaScript libraries and the main script file.
After defining all abstract variables, the start function can be constructed (Figure 15). If the server is not already running, it will be initiated as a new thread.

```
46      def start(self):
47          if not self.server_is_open:
48              self.port = self.get_free_port()
49              self.server_thread = threading.Thread(
50                  target=lambda: self.socketio.run(
51                      self.app, host="127.0.0.1",
52                      port=self.port,
53                      debug=True,
54                      use_reloader=False
55                  ),
56              )
57              self.server_thread.start()
58              self.server_is_open = True
```

Figure 13. Server start function. Python server file.

As the server's start function is triggered, it opens a web browser and defines two socket events: join and get (Figure 16). These functions are executed when the JavaScript client connects and sends corresponding get requests. Each socket event is identified by a name and linked to a respective function join and get_data.

```
60              url = f"http://localhost:{self.port}/?room={self.room_id}"
61              webbrowser.open(url)
62              self.socketio.on_event('join', self.join)
63              self.socketio.on_event('get', self.get_data)
```

Figure 14. Server start function. Python server file

The join function is responsible for connecting the client to the server. As the name suggests, it assigns an appropriate session ID to the client and adds it to a room (Figure 17). Once the client has joined, it becomes possible to iterate

through the message storage, where all requests associated with that specific client are stored. This is done with threading, that way asynchrony is enabled and larger items are processed later.

```
67      def join(self, data):
68          sid = data["sid"]
69          join_room(data["room"])
70          
71          def thread_function():
72              for message in self.storage:
73                  self.socketio.emit("message", message, to=sid)
74
75          threading.Thread(target=thread_function).start()
76
```

Figure 15. Server join function. Python server file

The get_data function handles receiving responses from the JavaScript client and processes them in the correct order. It includes strict data consistency checks and operates in a blocking manner. This means that when Python awaits a response, execution is paused until the expected reply is received (Figure 18).

```
123     def get_data(self, data):
124         if self.received_data_storage.get(data['get_id']) is not None:
125             print('returning')
126             return
127         self.received_data_storage[data['get_id']] = data['data']
128         if self.get_data_event:
129             self.get_data_event.set()
130
131     def wait_get(self, get_id):
132         if not self.server_is_open:
133             raise Exception("Attempt to GET data from the server that has not
134
135         self.get_data_event = threading.Event()
136         self.get_data_event.wait()
137         self.get_data_event.clear()
138
139         if get_id not in self.received_data_storage:
140             raise Exception("Data not received, or received incorrectly")
141
142         return self.received_data_storage[get_id]
```

Figure 16. Server get functions. Python server file

However, sending requests is the primary mechanism for delivering data to the JavaScript client. The function responsible for handling any request is emit_message (Figure 19). Each request is also stored in the message storage to ensure that data is saved and can be processed when required.

```
77      def emit_message(self, data: dict):
78          data = self.zstd_msgpack_prepare(data, debug=True)
79          self.debug_storage(data)
80
81          self.storage.append(data)
82
83          if self.server_is_open: # this happens when connected to the server
84              self.socketio.emit("message", data, to=self.room_id)
85
```

Figure 17. Server emit function. Python server file

As an extension to existing communication, function with zstd compression was included into the Server class (Figure 20). It is required to make data more compact before sending it over the loopback traffic. Tests show up to 50% speedup due to compression feature included.

```
95      def zstd_json_prepare(self, data, debug=False):
96          start_time = timeit.default_timer()
97          dump = json.dumps(data).encode("utf-8")
98          compressed = zstd.compress(dump)
99          end_time = timeit.default_timer()
100         if debug:
101             print(f"Prepare time (JSON): {(end_time - start_time) * 1000:.2f}
102         return compressed
103
```

Figure 18. Server compression function. Python server file

The overall Server class structure ensures efficient real-time communication between the backend and frontend. It includes the Flask app instance and defines routes, with the index function serving the index.html template to the client. The start function initializes the server and opens a web browser, while also setting up the WebSocket events, join and get, which handle client connections and requests. Only wait_get method is blocking event during runtime.

Client

```python
from .server import Server
import uuid

class Chart:
    def __init__(self):
        self.id = str(uuid.uuid4())
        self.server = Server()

    def open(self):
        self.server.start()

    def close(self):
        self.server.stop()

```

Figure 19. Client abstract Chart class with general functions applicable to any chart. Python client file

```python
class ChartXY(Chart):
    def __init__(self): # get info does not work in init (we only starting the chart)
        super().__init__()
        self.server.emit_message(
            {"id": self.id, "method": "createChartXY"}
        )

    def set_title(self, title: str):
        self.server.emit_message(
            {"id": self.id, "method": "setTitle", "param": {"title": title}}
        )

    def get_title(self):
        get_title_id = str(uuid.uuid4())
        self.server.emit_message(
            {"id": self.id, "method": "getTitle", "param": {"get_id": get_title_id}}
        )
        title = self.server.wait_get(get_title_id)  # call general GET function
        return title

    def log_time(self, start: float):
        get_log_id = str(uuid.uuid4())
        self.server.emit_message(
            {"id": self.id, "method": "logTime", "param": {"get_id": get_log_id, "start": start}}
        )
        log = self.server.wait_get(get_log_id)
        return log
```

Figure 20. Client ChartXY class with request structured functions. Python client file

```python
class LineSeries:

    def __init__(self, server, chart_id):
        self.server = server
        self.chart_id = chart_id
        self.id = str(uuid.uuid4())
        self.server.emit_message(
            {"id": self.id, "method": "addLineSeries", "param": {"chart_id": self.chart_id}}
        )

    def add_Point(self, x: float, y: float):
        self.server.emit_message(
            {"id": self.id, "method": "addPoint", "param": {"x": x, "y": y}}
        )

    def add_Points(self, data_points: list):
        self.server.emit_message(
            {"id": self.id, "method": "addPoints", "param": {"data_points": data_points}}
        )
```

Figure 21. Client LineSeries class. Python client file

4.2.2 JavaScript

In this subsection JS client and HTML file are overviewed. Client is meant to be a receiving script, which has uncompressing and decoding functions reference to the actual LightningChart JS library. JS client is the main script file handling requests received from Python.

JS

```javascript
<reference path="lcjs.iife.d.ts" />

const { lightningChart } = lcjs;

const lc = lightningChart({
    license: license,
    licenseInformation: {
        appTitle: "LightningChart JS Trial",
        company: "LightningChart Ltd."
    }
});
```

Figure 22. JS lc import. JS file

```
const createChartXY = (id, param = {}) => {
    objects[id] = lc.ChartXY();
    get_id = param.get_id
};

const setTitle = (id, param = {}) => {
    const chart = objects[id];
    chart.setTitle(param.title);
};

const getTitle = (id, param = {}) => {
    const chart = objects[id];
    const data = chart.getTitle();
    get_id = param.get_id
    console.log("get_id:", get_id);
    sendData({get_id, data});
};

function sendData(data) {
    socket.emit('get', data);
}
```

Figure 23. Example functions to handle the request instructions. JS file

Conceptually, it is achieved by creating a list of methods inside the JS file (Figure 26). Once the data is received, it contains a unique id, method information, and respective parameters to the method. As long as data is structured well, it is possible to process data, map methods, and produce results.

```
// Map functions to method names
const methods = {
    "createChartXY": createChartXY,
    "setTitle": setTitle,
    "getTitle": getTitle,
    "addLineSeries": addLineSeries,
    "addPoint" : addPoint,
    "addPoints" : addPoints,
    'logTime': logTime,
};
```

Figure 24. Methods list with relative naming. JS file

```
93  function zstd_json(data) {
94      const compressed = new Uint8Array(data);
95      const decompressed = fzstd.decompress(compressed);
96      const decoded = JSON.parse(new TextDecoder().decode(decompressed));
97      return decoded;
98  }
```

Figure 25. Compression function. JS file

```
124 function connectToRoom(room) {
125
126     socket.on('connect', function() {
127         console.log(`Connected with ID: ${socket.id}`);
128         socket.emit('join', {room, "sid": socket.id});
129     });
130
131     socket.on('disconnect', function() {
132         console.log(`Disconnected`);
133         window.close();
134     });
135
136     socket.on('message', async (received_data) => {
137         data = zstd_msgpack(received_data);
138
139         if (Array.isArray(data)) {
140             for (const element of data) {
141                 console.log("Processing:", element);
142                 const id = element.id;
143                 const method = methods[element.method];
144                 if (method) {
145                     method(id, element.param || {});
146                 }
147             }
148         }
```

Figure 26. JS client function to handle the connection. JS file

It is also planned to upgrade this mechanism, to remove unnecessary list of methods. Very alike to Python, JS should be able to match key words as methods, as long as methods defined in the data are supported.

HTML

```
21  <body>
22      <script src="{{ url_for('static', filename='script.js') }}"></script
23      <script defer>
24          connectToRoom("{{ room }}");
25      </script>
26  </body>
```

Figure 27. Script reference for the web page. HTML file

5 TESTING

Before starting this section, it is essential to keep in mind that the prototype is based on well-known and therefore, well-tested technologies. The author assumes that the system, which is developed on a TCP-based connection-oriented protocol provided by Flask-SocketIO results in reliable communication.

The aim of this section is to apply analyzation methods and benchmark solution performance, additionally, but secondary, ensuring data consistency on the run. Even though, analysis is also aiming to satisfy curiosity needs by highlighting things, hidden behind the actual functionality. At this point, testing and analysis are blurring into each other. In fact, observing the solution behavior using several analyzation methods resulted in creating comprehensive prototype testing. Otherwise, the whole software system functionality is dictated by several components based on information technologies fundamental principles. In simpler words, prototype relies on network and code functionality. As any software would.

Therefore, the main testing scope is performance of a constantly developing prototype. While code in development, it was decided to concentrate on two main aspects justified in the communication between Python and JS challenges' part. Therefore, these are data processing and communication performance.

5.1 Testing objectives

Testing objectives included measuring latency and performance, identifying potential bottlenecks and ensuring in reliability of communication between Flask and JavaScript.

Overview of applied tests schema:

1. Python client – logging, print(), benchmarking methods
2. JavaScript client – console logging
3. Network – Wireshark packet analysis tool

5.2 Test results

One of the most fascinating findings discovered during network packet traffic examination is a void between package transmission (Figure 30). This gap is dependent from item size and can not be logged or seen directly. The study cannot answer to the nature of this question.

Time	Protocol	Data Length (in bytes)	Info
3.421613	TCP	65539	52299 → 52304 [ACK] Seq=3667410 Ack=2(
3.421624	TCP	65539	52299 → 52304 [ACK] Seq=3732905 Ack=2(
3.421640	TCP	65539	52299 → 52304 [ACK] Seq=3798400 Ack=2(
3.421691	TCP	44	52304 → 52299 [ACK] Seq=2012 Ack=3863
3.421701	TCP	65539	52299 → 52304 [ACK] Seq=3863895 Ack=2(
3.421721	TCP	44	[TCP ZeroWindow] 52304 → 52299 [ACK]
3.423803	TCP	44	[TCP Window Update] 52304 → 52299 [AC
3.423853	TCP	65539	52299 → 52304 [ACK] Seq=3929390 Ack=2(
3.423874	TCP	65539	52299 → 52304 [ACK] Seq=3994885 Ack=2(
3.423890	HTTP	64351	HTTP/1.1 200 OK (text/plain)
3.423969	TCP	44	52304 → 52299 [ACK] Seq=2012 Ack=4124(
3.427056	TCP	44	[TCP Window Update] 52304 → 52299 [AC
9.145760	WebSocket	51	WebSocket Text [FIN] [MASKED]
9.145811	TCP	44	52299 → 52307 [ACK] Seq=138 Ack=592 W
9.146260	WebSocket	47	WebSocket Text [FIN]
9.146291	TCP	44	52307 → 52299 [ACK] Seq=592 Ack=141 W
27.072575	WebSocket	47	WebSocket Text [FIN]
27.072630	TCP	44	52307 → 52299 [ACK] Seq=592 Ack=144 W

Figure 28. Red square highlights time gap between packets. Wireshark Packet capture

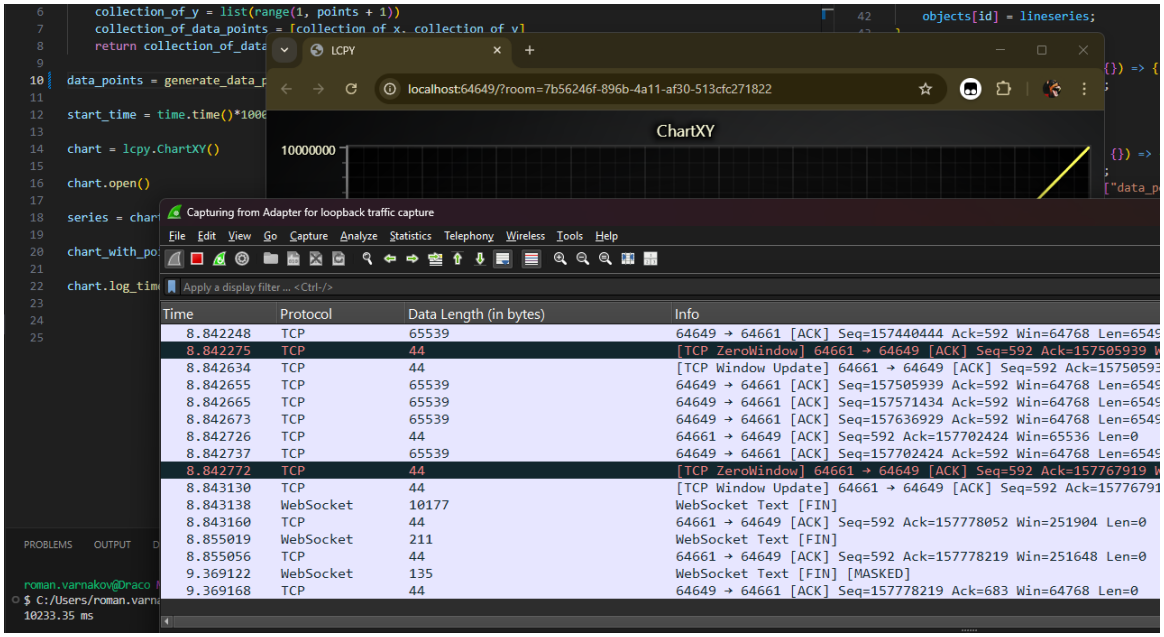

Figure 29. Around 10.2 (+- 0.5s) seconds for 10.000.000 points over loopback traffic from Python to JS. Python, JS, Wireshark applications

From initializing chart until visualization 10.2 seconds elapsed (Figure 31). From first packet sent. Not seen in the picture, but in the beginning of the capture TCP handshake. In the Wireshark capture [FIN] signals about complete data transfer between two languages.

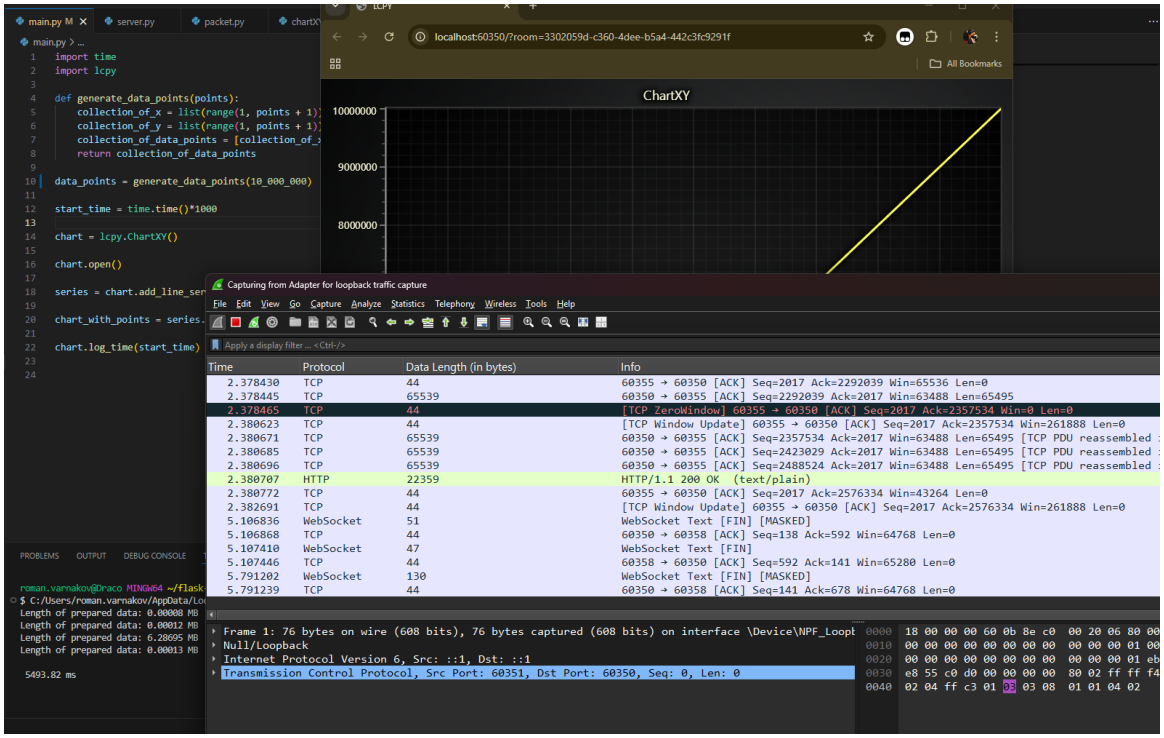

Figure 30. Around 5.5 (+- 0.5s) seconds elapsed for 10.000.000 points transfer over loopback traffic from Python to JS. Python, JS, Wireshark applications

Same functions and transmission/visualization methods used as in previous example, only zstd lossless compression library is leveraged (Figure 32). As the result almost 50% speed up is achieved compared to Figure 31 results. Testing also shows that it is the fastest way to transmit data through loopback traffic with any combination of JSON, MessagePack, NumPy and set of compression libraries.

During the analysis of the results, it was clearly seen that HTTP packets over loopback traffic are persistent. Notably, that both HTTP and WebSocket are using TCP protocol during data transmission over internet traffic. Which is quite expected, this way data consistency and right order of packets is kept.

5.3 Bugs during testing

During prototype testing multiple bugs were discovered. In this section, the most fascinating are discussed. Possible fixes are considered. One of the most relevant but to the study would be a chart duplication. Its essence is well described under Figure 12. Under Figure 33 the actual look of the bug can be seen.

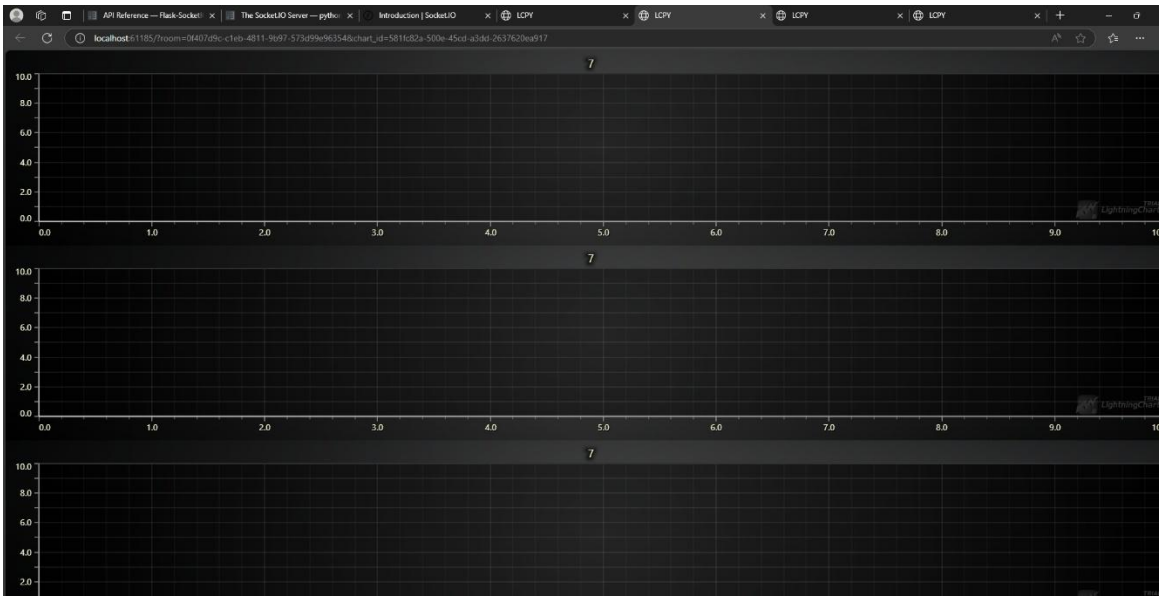

Figure 31. Chart duplication on one page example. Localhost webpage

The example of manual logging function results (Figure 34). At the beginning of 100-time logs from Python to JS, less than 20 are shown.

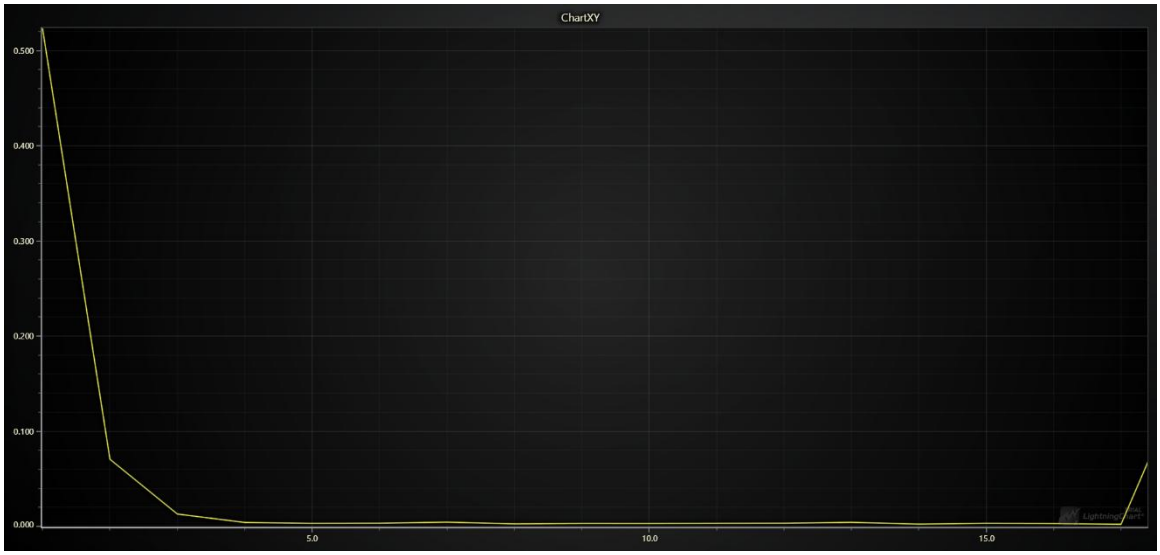

Figure 32. Ping stats. Axis Y in milliseconds, axis X packet sequence number. LightningChart ChartXY chart

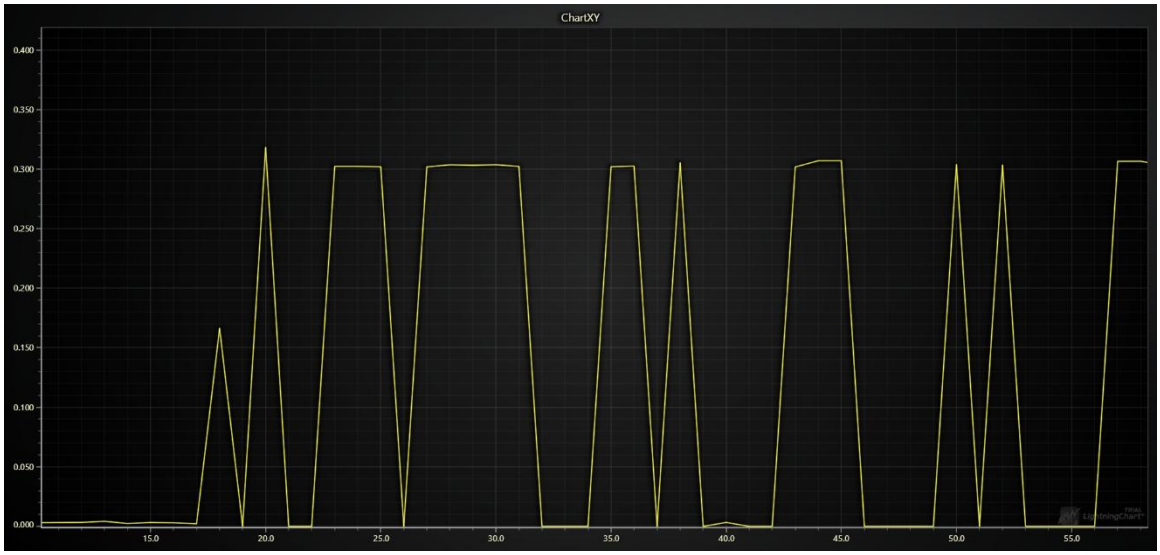

Figure 33. Ping stats. Axis Y in milliseconds, axis X packet sequence number. LightningChart ChartXY chart

According to the results achieved with manual methods for pinging the JS client from Python client. It could be seen that delay of around 0.3 seconds appears (ideally <1ms). From now on, this is an unresolved bug, which may affect

benchmark results. On the other hand, this is an immediate request of 100 (<60 shown) time calculation requests in a row.

Error with 'too many packets in the payload' (Figure 36). It appears if JS is trying to send back a large amount of get responses at the same time on page refresh. Fixed with adding two lines of code, shown in Figure 37. The fix is aiming to enlarge the number of packets received at the same time.

```
returning
post request handler error
Traceback (most recent call last):
  File "C:\Users\roman.varnakov\AppData\Local\Program
    socket.handle_post_request(environ)
  File "C:\Users\roman.varnakov\AppData\Local\Program
    p = payload.Payload(encoded_payload=body)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\roman.varnakov\AppData\Local\Program
    self.decode(encoded_payload)
  File "C:\Users\roman.varnakov\AppData\Local\Program
    raise ValueError('Too many packets in payload')
ValueError: Too many packets in payload
```

Figure 34. Error Traceback telling payload size reached. VScode console

```
19    from engineio.payload import Payload
20
21    Payload.max_decode_packets = 500*500
```

Figure 35. Payload corrections import statements. Python server file

Bug with favicon.ico, which couldn't be loaded and caused annoying error in console log in web browser. It was fixed with only one line of code. This information might be valuable for other software developers who in the early stages of code production with no set icon yet. The output example can be seen from Figure 38. Fixed by adding <link rel="shortcut icon" href="#"> line of code in the html <head>.

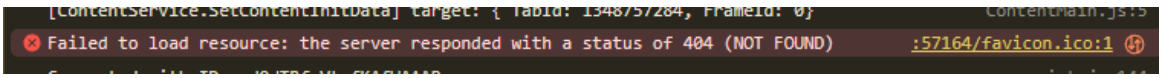

Figure 36. JS attempts to retrieve an icon, error appears. JS console log

Bug with MIME text type, happens when JS file is not found (Figure 39). Fixed accordingly by specifying the file location correctly. Fixed by ensuring that the path to the script is correct.

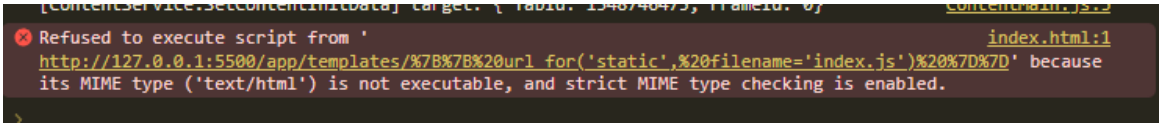

Figure 37. MIME text type inability to execute. JS console log

6 RESULTS AND DISCUSSION

The main goal of the study was to develop two-layer communication layer between Python and JS runtimes. With Flask and SocketIO as the core technologies it was possible to make a communication possible. Generally, the goal was achieved in next phases: collecting information, developing the communication layer, testing and analysis.

Collecting information played a significant role in establishing an understanding of multiple technologies used during the prototype development. Starting from network basics and ending with data retrieval functions in programming languages. As well as applying theoretical findings on practice this step allowed to develop a working prototype.

In the development step, the use of Flask provided a lightweight yet effective web framework, while SocketIO enabled real-time, event-driven communication between the Python client and JavaScript client. A class-based architecture was designed and introduced. Multiple serialization methods, such as JSON, MessagePack, Numpy, in combination with different compression libraries were used to create a better performance.

Testing of the prototype was applied for measuring latency and benchmarking communication performance and analyzing the results. Manual logging methods such as console.log() in JavaScript and print() in Python were used to retrieve

additional information during communication. Above that, Wireshark was used to capture and inspect network traffic in the loopback.

While the system remained operational, the regular delays in response under certain conditions such as multiple responses on reload. Besides, the prototype met the expectations and functionality objectives, demonstrating stable performance and reliable real-time data exchange even under heavier loads. However, testing also revealed a recurring issue: a slight inconsistency in data transmission. Especially, the gap could not be thoroughly explored within the present study due to scope limitations and the manual nature of the logging process. Nevertheless, it points toward a clear area for future research, especially concerning network reliability and real-time synchronization under non-ideal conditions. Relevantly, can be cited: “The hardest thing of all is to find a black cat in a dark room, especially if there is no cat.” - Confucius (Goodreads 2025).

7 CONCLUSION

In conclusion, the study is achieved the initial objectives and allowed to create a two-way communication in the prototype functionality. For this, networking-based approach was used in combination with multiple technologies enhancing the prototype performance. Additionally, prototype was tested and approved.

Despite the achieved results, the room for improvement was identified and highlighted. Future study should focus on investigating such anomalies as gaps and jitters in the communication. These issues may affect the consistency of real-time data transmission and thus warrant deeper technical analysis.

Overall, the developed prototype as the main study objective, serves as a foundation for future research. And demonstrates one of possible ways of establishing the communication between Python and JavaScript. Its modularity and open design also allow for easy adaptation to different use cases and further technological enhancements.

REFERENCES

CompTIA (2019). Network Protocol Definition | Computer Protocol | CompTIA. [Webpage] Available at: https://www.comptia.org/content/guides/what-is-a-network-protocol [Accessed 4 Apr. 2025].

CompTIA (2024). What Is Wireshark and How to Use It. [Webpage] Available at: https://www.comptia.org/content/articles/what-is-wireshark-and-how-to-use-it [Accessed 8 Apr. 2025].

Daily Dev Ltd (2024). Python and JavaScript: Choosing Your First Language. [Blog Post] Available at: https://daily.dev/blog/python-and-javascript-choosing-your-first-language [Accessed 15 Apr. 2025].

Fahim, S.M. (2025). What Are Logs in Programming? [Article] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/what-are-logs-in-programming/ [Accessed 8 Apr. 2025].

flask-socketio.readthedocs.io. (n.d.). Welcome to Flask-SocketIO’s documentation! — Flask-SocketIO documentation. [Documentation] Available at: https://flask-socketio.readthedocs.io/en/latest/ [Accessed 10 Feb. 2025].

flatbuffers.dev. (n.d.). FlatBuffers: FlatBuffers. [Webpage] Available at: https://flatbuffers.dev/ [Accessed 31 Mar. 2025].

Goodreads.com. (2025). A quote by Confucius. [Quote] Available at: https://www.goodreads.com/quotes/140655-the-hardest-thing-of-all-is-to-find-a-black [Accessed 14 Apr. 2025].

JSON.org (2023). JSON. [Webpage] Available at: https://www.json.org/json-en.html [Accessed 31 Mar. 2025].

Kurose, J.F. (2005). Computer Networking: A Top-Down Approach Featuring the Internet, 3/e. [Book] Pearson Education India. Available at: https://www.cs.sjtu.edu.cn/~linghe.kong/CS339/Download/ComputerNetworking.pdf [Accessed 26 Mar. 2025].

Lenovo (2021). Loopback Explained: Testing & Troubleshooting | Lenovo US. [Webpage] Available at: https://www.lenovo.com/us/en/glossary/what-is-loopback/?orgRef=https%253A%252F%252Fwww.google.com%252F&srsltid=AfmBOoo1RI7QoZh8RlgU47rGZILxMV8TyCtVkJO81Kdj4eQNdIpuUacq [Accessed 7 Apr. 2025].

Lightning-Chart (2024). GitHub - Lightning-Chart/javascript-charts-performance-comparison: Public comparison of JavaScript chart libraries performance in visualizing a real-time multichannel ECG chart. [GitHub Repository] Available at: https://github.com/Lightning-Chart/javascript-charts-performance-comparison [Accessed 15 Apr. 2025].

LightningChart. (2024). LightningChart Story - Pioneers in High-Performance Charts. [Webpage] Available at: https://lightningchart.com/our-story/ [Accessed 28 Mar. 2025].

MDN (n.d.). HTTPS - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. [Webpage] Available at: https://developer.mozilla.org/en-US/docs/Glossary/https [Accessed 7 Apr. 2025].

MDN Contributors (2019). HTTP. [Webpage] MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/HTTP [Accessed 7 Apr. 2025].

MDN Contributors (2023). JavaScript. [Webpage] MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/javascript [Accessed 26 Mar. 2025].

MDN Web Docs. (2024). Just-In-Time Compilation (JIT) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. [Webpage] Available at: https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation [Accessed 26 Mar. 2025].

Msgpack.org. (2017). MessagePack: It’s like JSON, but fast and small. [Webpage] Available at: https://msgpack.org/ [Accessed 31 Mar. 2025].

Nguyen, A., Siekkinen, M. and Ovchinnikov, A. (22AD). Programming Language interoperability in cross-platform software development. [Research Paper] Available at: https://aaltodoc.aalto.fi/server/api/core/bitstreams/726bbfd6-613f-4c8d-8a06-9abeda771502/content [Accessed 28 Mar. 2025].

Pramatarov, M. (2021). What is IPv4? Everything you need to know. [Blog Post] ClouDNS Blog. Available at: https://www.cloudns.net/blog/what-is-ipv4-everything-you-need-to-know/ [Accessed 7 Apr. 2025].

Protocol Buffers Documentation. (n.d.). Protocol Buffers. [Documentation] Available at: https://protobuf.dev/ [Accessed 28 Mar. 2025].

Python Software Foundation (n.d.). What Is Python? Executive Summary. [Webpage] Python. Available at: https://www.python.org/doc/essays/blurb/ [Accessed 25 Mar. 2025].

Python.org. (2013). FrontPage - Python Wiki. [Wiki Page] Available at: https://wiki.python.org/moin/ [Accessed 25 Mar. 2025].

Statista (2025). Most Used Languages among Software Developers Globally 2019. [Statistics] Statista. Available at: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ [Accessed 25 Mar. 2025].