School of Computing
Teesside University
Middlesbrough TS1 3BA

Assessing WebSocket Protocol Performance for Real-Time Cryptocurrency Algorithmic Trading with Compiled, Intermediate and Interpreted Programming Languages in Cloud Environment

An academic research paper for possible submission to
Computers Journal (ISSN 2073-431X)

Submitted in partial requirements for the degree of MSc in Computing

Date: 24.8.2020
Martin Papík (18276455/1)
Supervisor: Ing. Bohuš Získal, Ph.D.

Wordcount: 8884

computers

MDPI

Article

Assessing WebSocket Protocol Performance for Real-Time Cryptocurrency Algorithmic Trading with Compiled, Intermediate and Interpreted Programming Languages in Cloud Environment

Martin Papík

School of Computing, Teesside University, Middlesbrough TS1 3BA, United Kingdom; V8276455@tees.ac.uk

Received: date; Accepted: date; Published: date

Abstract: The most of cryptocurrency exchanges provide their market data via WebSocket API. Therefore, trading systems are recommended to utilize WebSocket protocol to be able to connect to the exchange and receive the data. Various programming languages and their respective libraries which contain the referential RFC 6455 implementation of WebSocket protocol can be used for development of communication interface within the trading system. Our study focuses on their evaluation in order to determine their performance differences which are determinative for communication speed that is an important criterion for profitable trading system. Six connectors were developed in compiled, intermediate and interpreted programming languages and their respective WebSocket libraries and deployed in the cloud. The WebSocket layer performance of each connector represented by the event latency metric was tested with three cryptocurrency exchanges producing high workloads on their WebSocket API data streams. In this experiment WebSocket protocol implementation in C++ and Go, both of which belong to the group of compiled languages, have been evaluated as the best performing implementations. Node.js runtime which produces in-memory bytecode from JavaScript source which is considered as a representative of interpreted languages, placed second right behind C++, and that proves the high performance of Google's V8 engine. Python and PHP were measured approximately to have the same performance, slightly worse compared to Go. Java took the last place from the sample of tested languages. This observation implies compiled languages and JavaScript in combination with Node.js runtime together with their WebSocket libraries should be preferred for building communication interfaces within cryptocurrency trading systems.

Keywords: WebSocket protocol; programming language; programming language library; performance test; cryptocurrency exchange; algorithmic trading system; cloud

1. Introduction

In the last decade cryptocurrencies have experienced broad market acceptance and fast development [1]. The innovative and revolutionary concept of digital currency enabled the evolution of cryptocurrency exchanges, where buying and selling of cryptocurrencies is primarily concentrated. These exchanges are purely digital and do not have shutdown periods. This provides serious profit opportunities for wide spectrum of financial traders who take advantage of algorithmic trading [2]. Moreover, the attention of traders is drawn even more by the extreme volatility of the cryptocurrency market. The days when Bitcoin touched $20.000 and then went back to $7.000 – $8.000 are not so far away [3]. Furthermore, this is also confirmed by the overall growth of the cryptocurrency market capitalization from $18 billion in January 2017 to $599 billion in January 2018 [4].

Cryptocurrency trading and related technologies is still highly emerging market. The research in this field have seen considerable progress and notable upturn in interest and activity – more than 85% of related research papers have appeared since 2018 [5].

Technological advancements in the form of supercomputers, multi-core processors, GPUs, FPGAs, high performance networks and fiber optics together with services like colocation of trading servers, raw data feeds or direct exchange API access are contributing to reduction of competitive advantage differences across traders [6–8]. However, speed in the sense of being faster than other traders, is still one of key factors for cryptocurrency traders, as it is for traditional stock traders, because it creates profit opportunities by enabling a prompt response to market activity [9].

There are several sophisticated commercial and open-source cryptocurrency trading infrastructure systems, platforms and libraries aimed for analyzing, generating, routing and executing orders (3commas, Apex Trader, AutoView, Autonio, BitUniverse, BlackBird, BTC Robot, Cap.Club, Capfolio, Catalyst, CCXT, Coinigy, Coinrule, CryptoHopper, CryptoSignal, CryptoTrader, Ctubio, Freqtrade, Gekko, GoLang Crypto Trading Bot, GunBot, HaasOnline, HodlBot, Kryll, Leonardo, Live Trader, Pionex, ProfitTrailer, Quadency, Shrimpy, Signal, StockSharp, TradeSanta, ZenBot, Zignaly etc.) and there are also many others like real-time, turtle or arbitrage systems developed for utilizing very specific algorithms that execute suitable trading strategies or run under specific market conditions [5]. Each of these systems has been developed in different programming language varying mainly between C++, C#, Go, JavaScript, PHP and Python. Based on several researches where authors discuss determination of use and measure performance and quality aspects of widely used programming languages, including those which were used for development of mentioned trading systems, it is highly probable that results of trade execution in the real world would vary for each of them [10–13].

Each of the mentioned systems implements software component which is intended for communication with cryptocurrency exchange in order to receive market data that is mostly provided via API utilizing WebSocket protocol. Despite there are some studies which focus on WebSocket protocol performance measurements [14–21], there are very few studies which focus on measuring WebSocket performance in context of programming languages [22, 23].

Imre and Mezei proposed a design of WebSocket benchmark infrastructure created for measuring server-side performance of the WebSocket protocol. The study also validated presented infrastructure design with three measurement scenarios using industrially applied WebSocket implementation. Go programming language with Gorilla package was used on the server side. They also tried several other implementations like Node.js and Socket.io for JavaScript, C++ using WebSocket++ and Erlang with Cowboy framework with the final result that C++ and Go have the highest performance. At client side they used Node.js with ws library [22]. No other programming languages or their respective libraries were used on client side in order to provide their performance measurements.

Wang measured and evaluated performance of five Java WebSocket frameworks (Netty, Undertow, Vert.x, Grizzly and Jetty) from aspects of concurrency, flow, connection type and resource occupancy. The experiment proved that Netty and Underflow perform better in highly concurrent environments, while Grizzly is suitable for large flow conditions. The results also showed that with persistent connection, Netty far outperformed other frameworks and Vert.x and Underflow can handle most requests within relatively shorter time. Besides, Netty and Vert.x occupy less CPU and memory resources in comparison with other frameworks [23].

We have not found any study which focuses solely on establishing communication and receiving events from cryptocurrency exchanges at different workloads and argues the suitability of use of different implementations of WebSocket protocol in trading systems. Performance optimization of a component covering such process may undoubtedly help with reducing the time what is needed for receiving events from exchanges and logically also reducing the overall time needed for trading strategy execution. In the algorithmic trading environment, where each millisecond and even microsecond in communication latency with the exchange is crucial for the trade execution [3, 7, 9, 24–26], the deeper examination of WebSocket protocol performance is expected to lead to significant

findings useful for cryptocurrency trading systems development decisions. These decisions are important for companies, communities or individuals who endeavor to build the most profitable trading systems.

The integral part of our project was the development and subsequent performance testing of WebSocket connector – a software component responsible for subscription to cryptocurrency exchange API and for receiving market events. The WebSocket connector was developed with six programming languages utilizing the libraries with reference implementation of WebSocket protocol for each language. Each connector was placed on separate virtual server within chosen cloud provider. All server configuration parameters in terms of hardware and operating system were identical while servers were placed in the same network segment and datacenter location.

Measurements of WebSocket protocol implementations for each programming language with the performance test is the subject of experimental part of this project. One of the goals of the test was to generate high workloads so that each instance of WebSocket connector developed with particular programming language was receiving as many events from three WebSocket API data streams of selected cryptocurrency exchanges as possible. Further analysis and correlation of data obtained from the performance test helped us to answer the main research question: Does implementation of WebSocket protocol within compiled, intermediate and interpreted programming languages cause significant latency differences in receiving of events which are persistently streamed by cryptocurrency exchanges under various workloads?

We verified performance differences of interpreted, intermediate and compiled programming languages within our research. We confirmed conclusions of previous programming languages research pointing to the fact that languages like C++ or Go, which compile their source code directly into machine code, outperform intermediate and interpreted languages [13, 27] with regards to utilized WebSocket protocol libraries. Another important finding is that Node.js runtime used for execution of JavaScript WebSocket connector took second place just between two compiled languages – C++ and Go. It confirms the incredible performance of V8 which is Google's open source high-performance JavaScript and WebAssembly engine the Node.js runtime is built upon. Python slightly outperformed PHP and took the fourth place. The curiosity is that Java placed last. It was not expected because Java uses JIT (Just-In-Time) compilation and optimization mechanisms. We suspect that Java WebSockets library might not be optimized for workloads which were chosen for this test. This finding will require further investigation.

The selection of particular programming language and its library should be determinative in context of its further use for the composition of module which is responsible for communication with cryptocurrency exchange API within the algorithmic trading system.

The rest of the paper is organized as follows. In **Section 2** we focus on research methodology of WebSocket protocol and the design of system components employed in performance tests including important development decisions for the main component – WebSocket connector. The high-level system design proposed in this chapter is essential for understating of how the connector communicates with exchanges and what type of data is received within the experimental testing phase. The section ends with the description of performance test procedure. The test summary, data cleaning procedure, data evaluation methodology and explanation of results obtained within the performance test are described in **Section 3**. **Section 4** discusses the research results in broader context with regard to previous research and outlines possible future developments.

2. Materials and Methods

In this section, we propose the system architecture used for performance testing of WebSocket protocol implementation in chosen programming languages. Further, we introduce the experiment environment composed of cloud virtual servers, WebSocket connectors and related utilities, and cryptocurrency exchanges. The chapter ends with a description of test methodology.

2.1. System Architecture

The system is primarily designed to cover needs for experimental performance testing of WebSocket protocol in various programming languages. The system is built according to the well-known distributed structure of client-server model. Generally, the system architecture is composed of component types that are commonly used in the IT industry (Linux demons, databases, cloud servers and Internet services) and component types that were developed by authors of this research for the purpose of performance testing and analysis of WebSocket protocol within selected programming languages (WebSocket connector, test manager and data analyzer). Integration of mentioned components is based on the best practices and usual setup of trading systems communication interfaces. Detailed justification of the use of individual components is included in **Chapter 2.2**.

WebSocket connector plays a role of the client, it is placed in virtual server in cloud environment and used to handle events received from cryptocurrency exchanges. Cryptocurrency exchanges are considered servers within the client-server model. They are black-boxes which send events to the client according to the initial instrumentation received from the client. Client and server communicate together over WebSocket protocol in the Internet. The overall system architecture is shown in **Figure 1**.

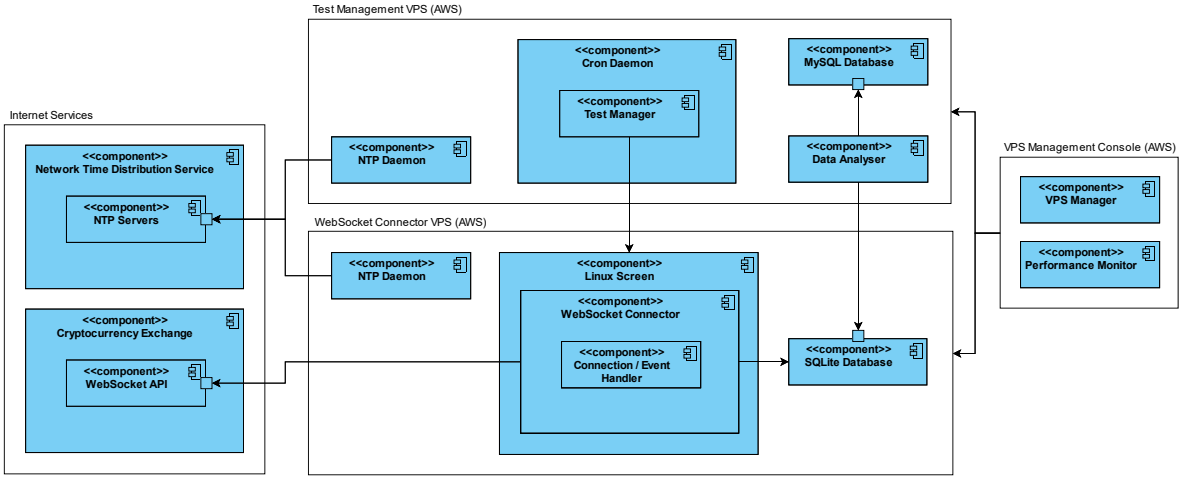

Figure 1. WebSocket connector system component architecture.

2.2. WebSocket Connector

The WebSocket connector is a CLI application and provides three main functions: it subscribes to a specific WebSocket API data stream of particular cryptocurrency exchange, measures arrival time of each received event and saves the event together with the detected timestamp to the SQLite database. High-level description of algorithm of the connector is shown in **Figure 2**.

Connector instruments exchange's WebSocket API and tries to create a connection with requested data stream by sending the opening handshake packet containing path value along with the GET request method and at least HTTP(S) 1.1 protocol version specification as a part of the header [28]. Once the exchange API server accepts the request, persistent connection between connector and exchange's WebSocket API data stream is created. As long as the WebSocket connection is open, connector listens for various event types – open, message, error and close – defined within RFC 6455 WebSocket protocol standard. Our WebSocket connector was developed to be able to subscribe to those data stream types which prescribe opening a connection via header that is a part of the handshake packet. The connection with cryptocurrency exchanges which require opening a connection with their data streams via additional message is not supported.

Connector assigns standardized Internet timestamp in milliseconds in UNIX Epoch time format to each event exactly in the moment when the event is received. NTP daemon is configured on all

WebSocket connector test servers and test management server. It uses public network time distribution service provided by ntppool.org to guarantee the measurement of undistorted time [29].

```
INPUT: set of three strings: exchange_id, exchange_api_domain, exchange_api_stream and one integer: exchange_api_port
OUTPUT: database file with unknown number of string – integer pairs: exchange_event and system_timestamp


if count of input_arguments is strictly less than 4                    # Verify number of input arguments
      exit                                                             # Number of arguments is strictly less than 4 - exit

create websocket_api_connection                                        # Create WebSocket API connection

if websocket_api_connection does not exist                             # Verify existence of WebSocket API connection
      exit                                                             # WebSocket API connection not created - exit

else                                                                   # WebSocket API connection created
      try                                                              # Try to execute following block of statements
            create database_file                                       # Create database file
            create database_tables_schema                              # Create database tables schema
            insert test_details to database                            # Insert test details into database

            while true                                                 # Execute until the connection is active
                  receive event                                        # Receive event from WebSocket API
                  get system_timestamp                                 # Get timestamp from operating system
                  insert event and system_timestamp into database      # Insert event and timestamp into database

                  if exit_file exists                                  # Verify existence of exit file
                        exit                                           # Exit file exists - exit

      catch exception                                                  # Catch exception thrown in try block
            exit                                                       # Exception caught in try block - exit
```

Figure 2. Pseudocode of WebSocket connector.

At the beginning of each test, connector creates the SQLite file database with two tables in it. Structure of the database is shown in **Figure 3**. One table contains metadata with details relevant to the particular test. The second table contains automatically incremented ID for each received event, timestamp indicating when the event was received and the raw event itself. We decided to use SQLite database because of its lightweight file-based nature, its ease of implementation and its sufficient performance. SQLite is able to do 50.000 or more insert statements within one second on average desktop computer [30]. The other study shows it took only 0.12 seconds to insert 10.984 rows to it [31].

```
-- Adminer 4.2.4 SQLite 3 dump

DROP TABLE IF EXISTS "events";
CREATE TABLE "events" (
"event_id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"datetime_received" text NOT NULL,
"event" text NOT NULL
);

DROP TABLE IF EXISTS "test_details";
CREATE TABLE "test_details" (
"test_id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
"vps_id" text NOT NULL,
"connector_id" text NOT NULL,
"exchange_id" text NOT NULL,
"stream_id" text NOT NULL,
"stream_args" text NOT NULL,
"datetime_start" text NOT NULL
);
```

Figure 3. SQLite database structure.

For the purpose of performance test, connector was started in Linux screen. Linux screen utility multiplexes physical terminal between several processes, typically interactive shells. It allows the program to be executed in the independent VT100 virtual terminal window that can be detached from user's physical terminal while the program is still running [32]. It means that user may disconnect from the server for the duration of the test without interrupting the execution of WebSocket connector.

The essential criterion for the realization of the experiment was the choice of the programming languages and their respective libraries the WebSocket connector would be developed with. There are several studies which discuss the choice of programming languages for algorithmic traders with the usual result compiled languages have better performance compared to intermediate or interpreted languages. Python were measured as much as 852.462 seconds average relative running time for all measured factors compared to average relative time 1.0 for C++ with the implementation of trading algorithm based on number of econometric and statistical tests [13]. The other study which compares programming languages in economics upon solving the stochastic neoclassical growth model shows Java is 2.10 and 2.69 times slower than C++ while PyPy implementation of Python is 44-45 times slower and traditional implementation of Python is 155-269 times slower compared to C++ [27].

To not choose the language based only on performance assumptions we also considered latest professional report from RedMonk analyst company which compares programming languages according to their popularity rank on StackOverflow and GitHub for Q1 2020 [33]. Popularity index for our programming language selection is as follows: C++ 6th (1st compiled), Go 15th, Java 2nd (1st intermediate), JavaScript 1st (1st interpreted), PHP 4th and Python 2nd (shared place). The popularity of programming languages in this report well corresponds with the popularity of programming languages shown in recent research studies relevant to high performance applications [34, 35]. Our choice of programming languages and their respective WebSocket libraries which were used for the WebSocket connector development is presented in **Table 1**. The table also includes the version of compiler or interpreter used within the project.

Table 1. Programming languages and libraries used for WebSocket connector development.

Programming Language (Version)	Programming Language Implementation	Compiler / Interpreter (Version)	WebSocket Protocol Library (Version)
C++ (17)	Compiled	g++/gcc (8.4.0)	µWebSockets (18.10.0)
Go (1.10.4)	Compiled	go (1.10.4)	Gorilla WebSocket (1.4.2)
Java (JDK 11.0.8)	Intermediate	javac (11.0.8)	Java WebSockets (1.5.1)
JavaScript (ES10)	Interpreted	Node.js (10.22.0)	ws (7.3.1)
PHP (7.2.24)	Interpreted	Zend (3.2.0)	php-wss (1.6.1)
Python (3.6.9)	Intermediate	CPython (3.6)	websockets (8.1)

For the objectiveness of our test we decided to develop WebSocket connector in balanced sample of representatives of compiled, intermediate and interpreted languages. Compiler is a translator that generates machine code from source code. We call compiled languages as those whose final output is executable (e.g. PE, ELF etc.) in native machine code format. C++ and Go belong to this group. Nowadays, there are not many implementations of purely interpreted languages. Interpreter is step-by-step executor of source code where no pre-runtime translation takes place. Almost all known and widely used implementations of programming languages that are not directly compiled into machine code employ some sort of compilation of source code into bytecode and use the virtual runtime environment for its execution. We call these languages intermediate. There are several differences in source code interpretation/compilation process between languages which belong to this group. We call interpreted languages those whose source code is interpreted/compiled into bytecode and then into machine code on-the-fly, directly in memory, while the interpreter is being executed. JavaScript and PHP belong to this group. Java source code within our implementation is converted into bytecode in form of Java class files. They are then physically stored in jar bundle on file system and executed via JVM (Java Virtual Machine). Java is considered intermediate programming language. Finally, Python is also included into intermediate group. Python pre-compiles the source code into bytecode and stores it in its so-called .pyc files. There are two possibilities how to run the program. It is either executed with Python interpreter and immediately re-compiled into physical .pyc file

which is then running within PVM (Python Virtual Machine) or the .pyc file generated from previous compilation is executed directly without re-compilation.

We intentionally did not provide any performance optimizations for interpreters or compilers where possible (e.g. -O flags for gcc/g++) because we were interested in running our performance test with WebSocket connectors compiled or interpreted with default programming language settings.

Connectors were developed with chosen programming languages and their respective WebSocket libraries. During the development process we configured the OS (operating system) and programming language environment with necessary dependencies unique for each connector. We downloaded last version of µWebSockets library from GitHub for C++ connector [36]. Once the source code was prepared it was compiled with following g++ compiler parameters -std=c++17 -luWS -lssl -lcrypto -lz -lsqlite3 -lvsqlitepp -lboost_system -lstdc++fs. A separate binary application was created. Similarly, for Go connector, we downloaded Gorilla WebSocket library from GitHub and compiled the source code to a standalone binary application [37]. Java connector required source of Java WebSockets library, SLF4J logger and SQLite libraries [38]. We compiled the source code for mentioned libraries to jar files. Once we developed the Java connector, we compiled its source code with mentioned jar files to a Java connector class, included the class file into the build directory with other mentioned libraries and prepared final jar application. Java WebSocket connector application requires Java runtime for its execution. JavaScript connector was developed with ws library downloaded via npm package manager and with the standard libraries provided by Node.js [39]. JavaScript Websocket connector source code requires the Node.js framework for its execution. Last version of php-wss library was downloaded from GitHub via Composer, which is a dependency package manager for PHP [40]. PHP WebSocket connector source can be executed with PHP interpreter. Finally, websockets library for Python was downloaded via pip3 package installer [41]. Python WebSocket connector source code can be executed with the use of Python3 runtime.

In general, WebSocket connector is executed as a CLI application. Based on its programming language implementation type, it either requires the path to interpreter or runtime at the first place, or it runs as the compiled binary application. The application itself requires four parameters: exchange id which identifies the test type which is saved as the part of metadata into test_details table in SQLite database created at the beginning of each test. The other three important parameters stand for the configuration of connection to the WebSocket API data stream: exchange WebSocket API domain, exchange WebSocket API port and exchange WebSocket API stream.

2.3. Cloud Environment

For the test purposes all WebSocket connectors were deployed on virtual private servers in cloud environment. There are three leaders in regard of cloud computing services: Amazon (AWS), Microsoft (Azure) and Google (GCE) while Amazon is considered as the market leader according to the latest Gartner report [42]. Coinbase cryptocurrency exchange, one of many financial services is also hosted on Amazon's cloud infrastructure what confirms reliability of their infrastructure [43]. AWS provides sufficient amount of virtual server types with regards to their performance and also sufficient amount of geographical location options where the server might be deployed.

After the research of Amazon's EC2 VPS instance options, we decided to eliminate possible cloud's hardware and network bottlenecks with the choice of m5d.xlarge VPS type for WebSocket connectors [44]. We used the same instance type for management test server, which was automatically checking the state of each WebSocket connector in chosen period and eventually started those connectors which were disconnected from the exchange API. M5d.xlarge instance uses second generation of Intel Xeon Platinum 8000 Series Processor with all core Turbo CPU clock speed of 3.1 GHz and runs Linux Ubuntu 18.04 LTS (64-bit x86) operating system. Its parameters are shown in **Table 2**.

AWS Management Console allowed us to manage VPS instances according to our needs, e.g. rebooting, adding new storage, creating volumes, checking system status and performance monitoring (CPU utilization [%], disk reads/writes [B], network packets in/out [B]).

Table 2. Amazon AWS EC2 VPS instance parameters.

Model	vCPU	Memory (GiB)	Instance Storage (GiB)	Network Bandwidth (Gbps)	EBS Bandwidth (Mbps)
m5d.xlarge	4	16	150 NVMe SSD	Up to 10	Up to 4,750

There are studies which observed positive linear correlation between distance of datacenters and response times [45]. Therefore, institutional algorithmic traders use colocation services and place their automated trading applications as close as possible within the exchange's datacenter [46, 47]. However, geographical location plays a major role for reducing latency of algorithmic trades in our case we have chosen Frankfurt as a cloud location for placement of our WebSocket connectors and realization of the performance test. For our experiment the location was not important. The distance between the exchange API servers and WebSocket connectors probably caused increased latency in receiving events from exchanges. We assume it proportionally affected all WebSocket connectors which were running parallelly in the same network segment created in Frankfurt datacenter location. Comparison of absolute latency time values was not relevant to us. We tended to measure relative latency differences in receiving events from the identical WebSocket API data streams of selected exchanges across all WebSocket connector implementations.

2.4. Cryptocurrency Exchanges

The most of established cryptocurrency exchanges provide their data via WebSocket API which work on notification principle. Trading application subscribes to WebSocket API and gets updates of cryptocurrency prices anytime there is an update which may happen every second or even much faster. This approach is more efficient and faster compared to REST API where a lot of GET calls must be performed in order to receive data updates.

In our case cryptocurrency exchange was a black-box we did not have control over in terms of performance. We only instrumented exchange's WebSocket API to receive required type of data. There are more than 300 cryptocurrency exchanges operating worldwide as of writing this article [48]. Before we determined quantitative criteria for selection of exchanges used within out test, we picked only centralized ones and those which implement Ticker, Candlesticks/OHLCV, Order Book and Trades streams, which we consider these were able to generate as much load as possible within one connection. The other two important technical requirements were that the cryptocurrency exchange had to support WebSocket Secure protocol (WSS) and subscription to its WebSocket API data stream via handshake packet header assembly.

We used following quantitative criteria for the exchange selection. Rating assigned by TokenInsight analyst company while one of the criteria was that the overall mark must not be worse than B. B mark means good risk control ability, possibility of a few abnormal risk and regular user ecological operation. BB and A marks stand for even more stable exchange systems [49]. The next selection criterion was that the exchange had at least 1 billion USD reported trading volume in the last 30 days (data reviewed 26.7.2020) what can be used for the assumption of sufficient event flow pushed to the WebSocket connector [50]. The legitimacy of reported cryptocurrency exchange trading volume is the last important criterion while the exchange had to have the minimum total score of 3 (score 5 means most accurate reported trading volume, 1 means inaccurate reported trading volume) [51]. **Table 3** also includes number of markets which is not recognized as an exchange selection criterion but it directly determines the theoretical number of data streams (and markets combinations within one stream) the WebSocket connector is able to subscribe to.

For our test we decided to use three cryptocurrency exchanges which provide their data through WebSocket API for our WebSocket connector. We used those which differ in trading volume and number of markets as much as possible. That was meant to be able to derive general results of WebSocket protocol performance in chosen programming languages without possibility of arguing that results gained for various WebSocket protocol libraries using exactly one exchange would

eventually completely differ for another exchange. Final sample of exchanges is composed of Binance, Bithumb and Gemini [52–54].

Table 3. Cryptocurrency exchanges used for receiving data via their WebSocket API.

Cryptocurrency Exchange	TokenInsight Rating	Trading Volume (USD/30 days)	Trading Volume Legitimacy Score	Number of Markets
Binance	A	85.64B	5	653
Bithumb	B	9.31B	3.5	105
Gemini	BB	1.02B	5	27

After choosing out exchanges, we tested instrumentation settings for their WebSocket APIs. For each exchange we downloaded the list of supported symbols via REST API and transformed it into WebSocket API query for a particular data stream. We tested combinations of data streams (e.g. Ticker, Candlesticks/OHLCV, Order Book and Trades etc.) for chosen exchanges and we selected those from which we received maximum amount of data to be able to utilize the exchange up to its limits within one connection. In case two or more streams gave as similar amount of data, we preferred the one with real-time update speed over those with 100ms or 1000ms update speed. **Table 4** shows the specification of data streams to which each of six WebSocket connectors was connected to for the duration of the performance test.

Table 4. Specification of cryptocurrency exchange WebSocket API data stream types.

Cryptocurrency Exchange	WebSocket API Endpoint	WebSocket Data Stream Type	Data Stream Update Speed
Binance	stream.binance.com	TRADE	Real-time
Bithumb	global-api.bithumb.pro	ORDERBOOK	Real-time
Gemini	api.gemini.com	MARKETDATA	Real-time

Figure 4 shows example of event payloads received from cryptocurrency exchanges involved in WebSocket protocol performance test. These events (the event type of message) were pushed by WebSocket APIs to connectors most frequently. The other types of events (open, error and close) were also present in the test but their count was insignificant.


```
{
   "stream":"btcusdt@trade",
   "data":{
      "e":"trade",
      "E":1596319805218,
      "s":"BTCUSDT",
      "t":370684729,
      "p":"11752.19000000",
      "q":"0.00000500",
      "b":2813775880,
      "a":2813775919,
      "T":1596319805217,
      "m":true,
      "M":true
   }
}
```

(a)

```
{
   "code":"00007",
   "data":{
      "b":[
         [
            "377.7600000000",
            "12.187121"
         ]
      ],
      "s":[

      ],
      "symbol":"ETH-USDT",
      "ver":"47562247"
   },
   "topic":"ORDERBOOK",
   "timestamp":1596319806676
}
```

(b)

```
{
   "type":"update",
   "eventId":12369938581,
   "timestamp":1596319806,
   "timestampms":1596319806063,
   "socket_sequence":1,
   "events":[
      {
         "type":"change",
         "side":"bid",
         "price":"11678.70",
         "remaining":"2.85587997",
         "delta":"2.85587997",
         "reason":"place"
      }
   ]
}
```

(c)

Figure 4. Example of event payloads received from WebSocket API data streams in JSON format: (**a**) Binance Trade event; (**b**) Bithumb Order Book event; (**c**) Gemini Market Data event.

2.5. Test Methodology

There are several studies which examine performance of WebSocket protocol. They are primarily focused on measuring concurrency number, data flow, connections and resource occupancy in laboratory environment where the server and client are built for this purpose. One

study considers 100kB as a large event in size for flow test which is relevant to us, while differences in performance of various Java frameworks were measured for events ranging from 1kB to 100kB in their sizes [23]. This study did not consider differences of various programming languages and did not discuss duration of test performed. There is also a paper which focuses on measuring data transmission performance of files of different image formats using WebSocket protocol running the test for 20 seconds only in laboratory environment [21].

We aimed to push the border of not only mentioned studies especially for flow test with measuring performance of WebSocket protocol implementations in different programming languages in real-world conditions connected to cryptocurrency exchange APIs for extended period of time. The sufficient performance test time is generally considered one or two days [55], while we decided to run the test for 100 hours (more than 4 days). We started our test on August 1st 2020 at 0:00 CET and stopped it on August 4th 2020 at 4:00 CET. Scenario parameters are shown in **Table 5**.

Table 5. Performance test scenario parameters.

Scenario ID	Cryptocurrency Exchange	WebSocket Data Stream Type	Markets Count	Workload	Duration (hrs.)
1	Binance	TRADE	628	High	100
	Bithumb	ORDERBOOK	157	High	100
	Gemini	MARKETDATA	1	High	100

To get the sufficient amount of data for our further analysis of events latency we connected each of our six connectors (written in C++, Go, Java, JavaScript, PHP and Python) to three chosen cryptocurrency exchange WebSocket APIs (Binance, Bithumb and Gemini) while the exchanges were instrumented to send the maximum amount of real-time data from their data streams to connectors. Each connector resided on one virtual server. General infrastructure scheme for performance test is presented in **Figure 5**.

To achieve accurate and effective results, we followed one-factor-at-a-time experiment principle for our test scenario [56]. We explicitly focused on measuring the event latency which represents the time it took the event sent by the cryptocurrency exchange to come to and be received by the WebSocket connector.

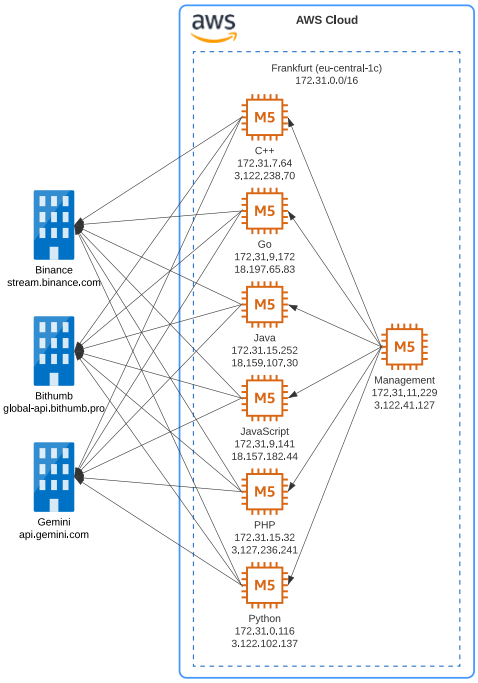

Figure 5. Performance test infrastructure scheme.

Virtual servers were placed in private network segment 172.31.0.0/16 in Frankfurt eu-central-1c location in Amazon AWS cloud. Each server was assigned its own public IP address. Hardware and OS layer configuration were identical for all servers (detailed information is described in **Chapter 2.3**). WebSocket connector application dependencies were installed and configured separately for each programming language implementation. We generated one SSH key pair for all servers so that we were able to manage them comfortably from either management server for the test purpose or from our local computer residing in Prague. Based on the fact that servers were placed in one private network segment, they were all interconnected.

The next step was the installation and configuration of NTP daemon to preserve identical time on all servers. Four servers provided by the NTP Pool Project network time distribution service – [0-3].pool.ntp.org – were used to synchronize the local time on test servers.

WebSocket connectors were intensively tested to verify that they work properly with no runtime errors and that they are ready for the test.

The last activity before starting the test was creating two shell scripts (management and exit script) and configuring cron daemon. First script was aimed to start and manage the WebSocket connectors automatically during the test and the second to stop all WebSocket connectors at the end of the test. First script was created in a way that it instruments each connector to connect to Binance, Bithumb or Gemini exchange. The connector startup procedure was as follows. The management script running on management server started the connector with appropriate WebSocket API data stream instrumentation in Linux screen virtual terminal on the test server. Three instances of WebSocket connector, each connected to the particular exchange, were created in connector server which was dedicated to the connector written in particular programming language. This was done for all six connector servers at the same time.

Cron daemon was configured on management server to execute the management script for each test (connector – exchange) regularly every 10 seconds. Management script verified whether the test is still running. When the script detected that the test is stopped for some reason (e.g. exchange closed the connection) the script started the test again so that the connector continued receiving messages from the exchange.

At the end of the test cron daemon was stopped manually on management server to prevent re-starting the tests again. The second – exit script – was executed from management server to stop all running connectors correctly on WebSocket connector servers. Several SQLite databases containing events received from exchanges and their appropriate timestamps indicating the time when they were received by the connector were created as the raw data outcome of the performance test.

3. Results

In this section, we summarize the results obtained during performance testing of WebSocket protocol implementation in chosen programming languages. We provide detailed description of data transformation procedure and data sets cleaning which was required for accurate calculations. Further we analyze the performance test results. The chapter ends with the comparison of WebSocket protocol implementations within chosen programming languages and their absolute placement with respect to each other.

3.1. Test Summary

The performance of WebSocket protocol implementations in referential libraries of six programming languages was tested experimentally from the cloud environment with the use of WebSocket connectors. The connectors were receiving events from cryptocurrency exchanges for 100 hours. The WebSocket API data streams were instrumented to provide the maximum possible workloads for WebSocket connectors. We obtained SQLite databases which contain raw exchange events received by connectors and timestamps referring to the time when the events were received. Based on these database records, we were able to calculate the latency of each event which was parallelly pushed by the exchange to each subscribed connector. Subsequently, it was possible to evaluate the difference between latencies for each WebSocket library/programming language. The

results which are discussed further correlate with similar programming languages research which used comparable performance testing methods.

After the test was completed, we verified the presence of data in each generated SQLite database. We prepared introductory overview of how the test ran. **Table 6** shows how many messages in total were pushed from particular cryptocurrency exchange API to all WebSocket connectors and how many messages in total were pushed by the particular API to all connectors per one minute and per one second.

Table 6. Total count of messages and message throughput from cryptocurrency exchange WebSocket APIs to all WebSocket connectors.

Cryptocurrency Exchange	Messages Count	Messages per 1 min.	Messages per 1 sec.
Binance	206.982.664	34.497	575
Bithumb	235.393.476	39.232	654
Gemini	92.833.799	15.472	258

Table 7 shows how many messages were approximately pushed from particular cryptocurrency exchange API to one WebSocket connector and how many messages were approximately pushed by the particular API to one connector per one minute and per one second.

Table 7. Total count of messages and message throughput from cryptocurrency exchange WebSocket APIs to one WebSocket connector in average.

Cryptocurrency Exchange	Messages Count	Messages per 1 min.	Messages per 1 sec.
Binance	34.497.111	5.750	96
Bithumb	39.232.246	6.539	109
Gemini	15.472.300	2.579	43

In general, more than 0.5 billion messages were received from all exchange APIs by WebSocket connectors in total what equals to 120GB in terms of saved file system data. The minimal size of the event received by the connector during the test was 124B, while the maximal event size was 805kB.

3.2. Data Transformation

Based on the asynchronous nature of the whole performance test, when particular tests were started and re-started automatically by the shell management script from cron daemon, there was suspicion that there are events which were not received by all connectors equally from the exchange. This could happen when exchange's WebSocket API terminated the connector connection. Therefore, before we started with detailed analysis of performance of WebSocket protocol implementations in various programming languages, we had to filter out such events.

In the first phase, we developed a PHP script which transformed data from SQLite databases created during the test into one central MySQL database used for further data analysis. We have chosen MariaDB (open source version of MySQL RDBMS) with TokuDB as a storage engine because it is designed for high performance on write-intensive workloads which is achieved with Fractal Tree indexing. Additionally, TokuDB supports up to 25x data compression, hot schema changes, hot index creation and hot table columns modifications [57]. These features helped us to speed up database schema development, database operations during data transformation, data analysis and physically minimize the overall size of collected data.

Generic MySQL table structure for test data is shown in **Figure 6**. Eighteen tables were created while each table stored data from the particular exchange (variable <exchange_id>) received by WebSocket connector developed with specific programming language (variable <websocket_connector_id>).

```
-- MySQL dump 10.16  Distrib 10.1.44-MariaDB, for debian-linux-gnu (x86_64)
--
-- Host: localhost    Database: wsc
-- ------------------------------------------------------
-- Server version  10.1.44-MariaDB-0ubuntu0.18.04.1

--
-- Table structure for table `events_<exchange_id>_<websocket_connector_id>`
--

DROP TABLE IF EXISTS `events_<exchange_id>_<websocket_connector_id>`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `events_<exchange_id>_<websocket_connector_id>` (
`event_id` bigint(20) NOT NULL AUTO_INCREMENT,
`datetime_received` bigint(20) NOT NULL,
`event` mediumtext NOT NULL,
`event_hash` varchar(32) DEFAULT NULL,
`datetime_event` bigint(20) DEFAULT NULL,
`datetime_diff` int(11) DEFAULT NULL,
PRIMARY KEY (`event_id`),
UNIQUE KEY `ev_hash_uq_idx` (`event_hash`)
) ENGINE=TokuDB DEFAULT CHARSET=utf8mb4;
/*!40101 SET character_set_client = @saved_cs_client */;
```

Figure 6. Generic MySQL table structure for test data.

PHP script took each SQLite database as input, read specified number of rows in a while loop, excluded those rows which did not contain string pattern that corresponded with expected event type using regular expression. For each relevant row taken from SQLite database, the script parsed the time when the event was sent by the exchange's WebSocket API to the connector from the event payload. Then the script calculated the time difference between parsed time and the time when the connector received the event by subtracting these two values. Moreover, the script calculated MD5 hash for the event payload for the purpose of further data transformation. Such data prepared by the script was inserted into MySQL database within a SQL transaction. This data cleaning procedure helped us to filter out events which were not relevant for our test because they did not contain timestamp value referring to time when the event was pushed from exchange's WebSocket API to WebSocket connector (e.g. open and close events). At the same time, we applied MD5 hash function to each record in event table so that we could continue with data cleaning.

MD5 algorithm for creation of a hash string for event record was chosen because of its speed in regard to the need to generate the hash for more than 0.5 billion events. MD5 outperforms other hashing algorithms like SHA-1, SHA-256 or SHA-512 in terms of speed [58]. Despite it is generally known that MD5 vulnerabilities were discovered, probability of producing collisions in which two different messages have the same hash values is extremely low in reality. Collisions occur mainly as a result of prepared attack [59].

The second phase involved data cleaning based on prepared MD5 hashes. In the database, there were events which were not received from particular exchange by all connectors equally during the test. This happened when one connector connection was closed by the API. Shell management script started the closed connection again within maximum of 10 seconds in such cases. During this small downtime of one connector the other connectors were still receiving events and storing the information when each event was received. These messages had to be cleared so that we could compare the difference between the time when the event was sent byt the exchange's WebSocket API and the time when the event was received by the connector for those events which were received by all six connectors only.

We created three new tables (binance_event_hash, bithumb_event_hash, gemini_event_hash) in MySQL database using inner join SQL statement for all connectors (all programming languages) that were connected to particular exchange. The statement is presented in **Figure 7**. Each of new tables represented unique set of MD5 hashes referring to events that were relevant for data analysis.

```
CREATE TABLE <exchange_id>_event_hash AS
SELECT a.event_hash
FROM events_<exchange_id>_<websocket_connector_id1> a
INNER JOIN events_<exchange_id>_<websocket_connector_id2> b
      ON b.event_hash = a.event_hash
INNER JOIN events_<exchange_id>_<websocket_connector_id3> c
      ON c.event_hash = b.event_hash
```

```
INNER JOIN events_<exchange_id>_<websocket_connector_id4> d
      ON d.event_hash = c.event_hash
INNER JOIN events_<exchange_id>_<websocket_connector_id5> e
      ON e.event_hash = d.event_hash
INNER JOIN events_<exchange_id>_<websocket_connector_id6> f
      ON f.event_hash = e.event_hash;
ALTER TABLE <exchange_id>_event_hash
ADD CONSTRAINT <exchange_id>_hash_uq_idx
UNIQUE KEY (event_hash);
```

Figure 7. Generic MySQL table structure for storing relevant MD5 hashes for particular exchange.

Irrelevant events had to be cleared from data tables according to MD5 hashes stored in new tables. SQL statements for the delete operation are presented in **Figure 8**.

```
DELETE FROM events_<exchange_id>_<websocket_connector_id>
WHERE event_hash NOT IN
      (SELECT event_hash
      FROM <exchange_id>_event_hash);
```

Figure 8. Generic SQL statement used to delete irrelevant events within data tables.

At the end of data transformation procedures, the events stored in MySQL data tables events_<exchange_id>_<websocket_connector_id> were cleared from events irrelevant for data evaluation.

3.3. Data Evaluation

Having the data cleaned and prepared for the analysis we performed basic statistical tests to discover its nature in terms of normality. We tested each numeric data set representing the difference between the time when the event was sent byt the exchange's WebSocket API and the time when the event was received by the connector using Shapiro-Wilk, D-Agostino K^2 and Anderson-Darling tests to evaluate whether the data sets follow Gaussian-like or non-Gaussian distribution [60, 61]. Then we could better decide what statistical and graphical methods to use so we were be able to provide qualified explanation of test results. All three tests are implemented as statistical functions within stats module in SciPy library for Python programming language thus they can be easily implemented to check the nature of data distribution [62].

Usually one normality test type provides correct result. Our data sets were unusually huge (tens of millions of records) and while we created Shapiro-Wilk test we noticed that its algorithm implementation in SciPy works correctly for data sets with number of items less than 5000 only. Therefore, we decided to test our data sets with all three mostly used normality tests implemented in SciPy. Using these tests for each of eighteen times difference data sets (column datetime_diff in events_<exchange_id>_<websocket_connector_id> table) the result was the same – the data was non-Gaussian nature. This finding indicated that data sets were skewed.

No single numeric measure is very useful for describing skewed distributions, what is usual for symmetric distributions. We applied "the five-number summary" which consists of median (Q2), the quartiles Q1 and Q3, and the smallest and largest individual observations – altogether minimum, Q1, median, Q3 and maximum. This statistical method filters out the outliers, which are values falling at least 1.5 x IQR above Q3 and below Q1. IQR is the distance between Q1 and Q3, mathematically it is a subtraction of Q1 from Q3. Five-number summary is visually incorporated within the boxplot type of graph which provides accurate insight into the data distribution. Because this type of graph is ideal for comparisons of several sets of compatible skewed data [63-65], we used it for representation of our performance test results.

Numerical calculations for data sets were performed by utilizing boxplot_stats function in simple algorithm we developed for this purpose. This function belongs to cbook module of matplotlib library for Python programming language [66]. Important calculations for our data sets are shown in **Table 8** which represents five-number summary metrics for WebSocket event latencies within Binance exchange, **Table 9** within Bithumb exchange and **Table 10** within Gemini exchange.

Table 8. Five-number summary metrics for Binance WebSocket API event latencies.

	WebSocket Event Latency [ms] @ Binance Exchange					
Metric	**C++**	**Go**	**Java**	**JavaScript**	**PHP**	**Python**
Maximum	135	138	**178** ↓	135	**134** ↑	137
Q3	126	128	**144** ↓	**125** ↑	127	128
Median	**121** ↑	123	**125** ↓	**121** ↑	123	124
Q1	**120** ↑	121	**120** ↑	**120** ↑	**122** ↓	**122** ↓
Minimum	**112** ↑	114	114	113	**115** ↓	114

Symbol ↑ indicates the best and symbol ↓ indicates the worst value for a metric.

Table 9. Five-number summary metrics for Bithumb WebSocket API event latencies.

	WebSocket Event Latency [ms] @ Bithumb Exchange					
Metric	**C++**	**Go**	**Java**	**JavaScript**	**PHP**	**Python**
Maximum	**100** ↑	101	**116** ↓	102	102	101
Q3	**90** ↑	91	**97** ↓	**90** ↑	92	91
Median	**86** ↑	87	**88** ↓	**86** ↑	**88** ↓	**86** ↑
Q1	83	84	84	**82** ↑	**85** ↓	84
Minimum	79	79	79	**78** ↑	**80** ↓	**80** ↓

Symbol ↑ indicates the best and symbol ↓ indicates the worst value for a metric.

Table 10. Five-number summary metrics for Gemini WebSocket API event latencies.

	WebSocket Event Latency [ms] @ Gemini Exchange					
Metric	**C++**	**Go**	**Java**	**JavaScript**	**PHP**	**Python**
Maximum	**146** ↑	204	**350** ↓	244	164	248
Q3	**90** ↑	115	**175** ↓	131	101	133
Median	**61** ↑	67	**73** ↓	68	66	67
Q1	**53** ↑	56	**57** ↓	55	**57** ↓	55
Minimum	46	46	**45** ↑	**47** ↓	46	46

Symbol ↑ indicates the best and symbol ↓ indicates the worst value for a metric.

We used seaborn for visual representation of five-number summary for each data set. Seaborn is a Python data visualization library [67]. We again prepared a simple algorithm which took data sets obtained by six WebSocket connectors within a particular exchange as the input and by utilizing seaborn's boxplot function it rendered appropriate boxplot graph. **Figure 9** shows boxplots generated for data sets which represent WebSocket event latencies belonging to programming languages and their respective WebSocket libraries. Each graph represents six boxplots that are data sets obtained by a connector within a particular exchange. Each boxplot represents approximately 15.5 to 34.5 of millions of events.

Y axis of each graph is dynamically adapted to the range between minimal and maximal value across all boxplots located inside the graph. Differences between Y axis ranges in provided graphs are logically caused by the nature of each particular exchange – its server implementation of WebSocket protocol together with internal server workloads reflecting the 30-days trading volume and number of traded pairs shown in **Table 3**. It is necessary to emphasize that this experiment is looking for relative differences in measured values and its purpose is not to explain absolute values.

Comparison of WebSocket protocol event latency based on five-number summary with its visual representation in boxplot graphs brought interesting insight in how the latency is distributed in data sets with regards to a particular exchange. The first fact we have to mention is that there were several unexpected outliers ranging from thousands to tens of thousands of milliseconds. These outliers were present in data sets obtained within each exchange and each programming language. We did not notice any unusual utilization of server performance metrics when we searched for the cause of their occurrence. Also, the local time on all test servers was synchronized against the Internet time service.

Therefore, we assume that either the exchange sent the old data to the connectors or the exchanges faced significant momentary utilizations of their APIs what caused that events were occasionally sent with an extreme delay. This characteristic may be a stimulus for specific further research.

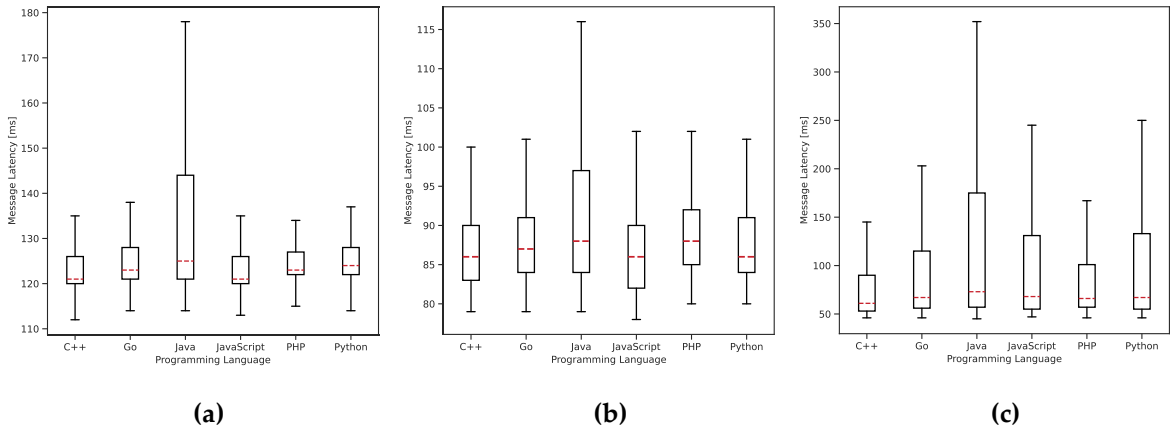

Figure 9. Boxplot graphs representing five-number summary for event latency data sets belonging to chosen programming languages and their respective libraries. Graphs represent event latencies in relation to: (**a**) Binance exchange; (**b**) Bithumb exchange; (**c**) Gemini exchange.

Looking at all graphs present in **Figure 9**, the most distinct observation is, that Java together with its implementation of WebSocket protocol in Java WebSockets library had the worst performance. Not only its maximum value is much higher than maximum values of other languages, but also IQR where 50% of all values from the data set fall, is approximately two times as long as the IQR length within the rest of languages. Interesting fact is, that minimum and Q1 values do not deviate much from values of other languages. This observation indicates that implementation of WebSocket protocol in Java WebSockets library is probably not optimized for higher event workloads with event throughput up to 109 events per second ranging between 124B and 805kB. On the other side, there is C++ implementation with µWebSockets library which reached the best results for most of five-number summary metrics. Especially, C++ reached the lowest median. It indicates that C++ was the fastest language in receiving the total 50% of events sent by all three exchanges. Based on the data presented with Table 8-10 the other performant language is JavaScript with its ws library and Node.js runtime. The latency of events was approximately equal to C++ within tests running with Binance and Bithumb exchanges, while it showed slightly worse results with Gemini exchange. Despite there are small recognizable differences in results for the remaining three languages and their respective libraries – Go with Gorilla WebSocket, PHP with php-wss and Python with websockets – it was difficult to evaluate it visually. Therefore, we employed very simple scoring model which helped us to determine the exact order of programming languages and their libraries in terms of WebSocket protocol performance.

We assigned a number to each value present in **Table 8-10** representing how good or bad the value is compared to other values within the same metric. The number assigned to each metric value was ranging from 6 (the best – assigned to lowest metric value which represented lowest event latency) to 1 (the worst – assigned to highest metric value which represented highest event latency). In case there were two similar values for various programming languages within the same metric, those two values were assigned the same score and the next value in sequence was assigned the score subtracted by the number of times the previous value occurred. Generic formula used for calculation of metric score per programming language:

$$\begin{aligned} &=\text{ROUND}((\text{SUM}(<\textit{MetricIdValue@LangId@Binance}>;\\ &\qquad <\textit{MetricIdValue@LangId@Bithumb}>;\\ &\qquad <\textit{MetricIdValue@LangId@Gemini}>)/3);2), \end{aligned} \tag{1}$$

With this method we calculated the score for each metric type aggregated across all three exchanges within a particular programming language. Score results are shown in **Table 11** and their visual representation is shown in **Figure 10**. The highest the score, the better the result.

Table 11. Comparison of metric scores within each programming language.

	Metric Score @ All Exchanges					
Metric	**C++**	**Go**	**Java**	**JavaScript**	**PHP**	**Python**
Maximum	5.67	3.67	1.00	3.67	4.67	3.33
Q3	5.67	3.67	1.00	5.00	3.67	3.00
Median	6.00	3.67	1.33	4.67	3.67	4.00
Q1	5.67	3.33	4.00	5.67	1.67	3.67
Minimum	5.33	4.67	5.00	4.00	2.67	3.67

Having the scores of metrics in relation with each programming language, we could perform second calculation to determine the final placement of each tested programming language. We calculated a sum of scores per programming language from **Table 11** and subtracted the sum with the numer of metrics as shown in formula:

$$\begin{aligned} &=\text{ROUND}((\text{SUM}(<\textit{MaximumMetricScore@LangId}>;\\ &\quad <\textit{Q3MetricScore@LangId}>;\\ &\quad <\textit{MedianMetricScore@LangId}>,\\ &\quad <\textit{Q1MetricScore@LangId}>;\\ &\quad <\textit{MinimumMetricScore@LangId}>)/5);2) \end{aligned} \tag{2}$$

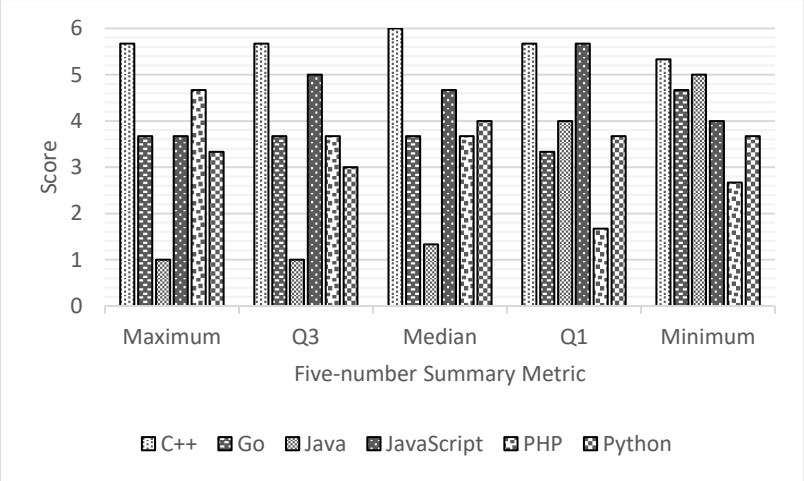

Figure 10. Metric scores within each programming language.

We calculated the final score for each programming language involved in performance test of WebSocket protocol as shown in **Table 12**. The highest the score, the better the result. The final placement of programming languages is shown in **Figure 11**.

Table 12. Final score of programming languages involved in WebSocket protocol performance test.

	C++	**Go**	**Java**	**JavaScript**	**PHP**	**Python**
Final Score	5.67	3.80	2.47	4.60	3.27	3.53

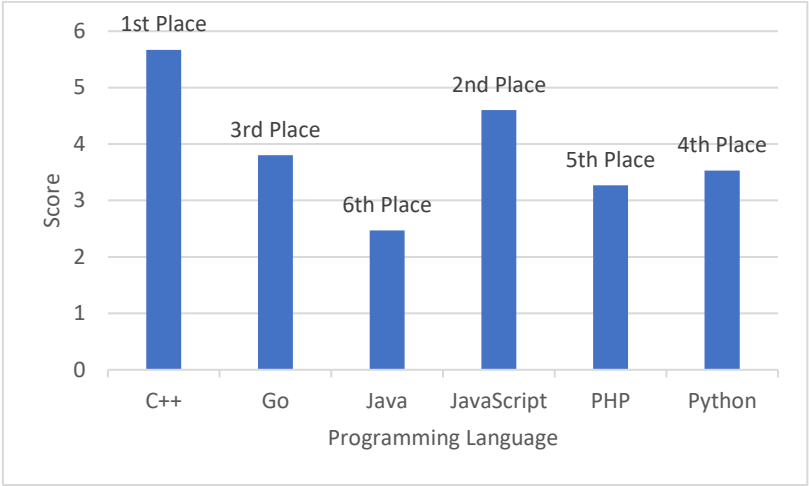

Figure 11. Aggregated scores and final placement of programming languages.

4. Discussion

Building a profitable real-time cryptocurrency algorithmic trading system is a challenging task. Professional traders are aware of the fact that being profitable not only comes with the sophisticated trading strategy but a reliable trading system which is capable of immediate trade executions is similarly important. Trading system is usually composed of several parts – exchange connector responsible for exchange connection and handling of market data, signal generator responsible for prediction analysis (technical, fundamental, combined etc.), risk allocator responsible for the allocation of exact amount of capital to traded pair, executor responsible for buying or selling within the particular exchange etc. All mentioned modules are usually driven by their own specific algorithms and optimization processes. Most of serious and established cryptocurrency exchanges enforce the communication with their APIs via WebSocket protocol due to its advantageous attributes like connection persistency, latency reduction, optimization of CPU and bandwidth utilization and simplicity. However, the implementation of WebSocket protocol on the client side, while the protocol considered "the layer under" serves within important processes like subscription to the exchange's API and receiving of events, might be a possible bottleneck negatively affecting the overall speed of the system. Therefore, we decided to experimentally test the performance of native WebSocket protocol implementation in several widely used programming languages while we intended to prove that the choice of suitable communication protocol implementation within a programming language has significant impact on the whole performance of the trading system. Moreover, we are convinced that building of such trading system must start with basic decisions concerning communication with the exchange.

Findings from our performance test experiment proved that there are significant performance differences in implementation of WebSocket protocol between various types of programming languages and their respective libraries. Differences in implementation of WebSocket protocol within Java frameworks (Netty, Undertow, Vert.x, Grizzly and Jetty) were also measured by Wang in laboratory conditions [23]. Outcome of their research is usable for those who make development and performance decisions with focus on Java programming language. In contrast with Wang our research showed that Java WebSockets library which is the reference implementation of WebSocket protocol in Java was the least performant implementation compared to other languages. Java, considered as an intermediate programming language, uses some form of JIT compilation and optimization processes while running the bytecode. Generally, it is also the first choice for development of enterprise applications. Therefore, its worst performance was not expected to be the one of the outputs of our research. This opens interesting new research question: How would Java WebSockets library do in performance comparison with other Java libraries which implement WebSocket protocol? We assume that Java WebSockets is probably not optimized for the size of

workloads occurred within our performance test. Our research showed that Java WebSockets library should not be considered for the development purposes of trading systems.

On contrary, we recommend using C++ for building trading system communication interface with the exchange. While μWebSockets library was the most performant one in our use case we deduce it would definitely be also suitable for other systems where performance is the top criterion. The only fact which needs to be considered is the complexity and speed of the development process. The same goes for the Go programming language and its Gorilla WebSocket library. These conclusions well correspond with research focused on comparison of performance of compiled and interpreted languages [13, 27].

JavaScript's performance approached C++ which was measured the highest performance in our test. Since Google developed V8 engine for Chrome browser which is also utilized by Node.js – an asynchronous event-driven JavaScript runtime – its usage in building scalable network application has grown enormously. JavaScript is nowadays the most used and loved programming language according to GitHub [33]. While many cryptocurrency exchanges provide their API snippets and describe the use of their APIs in JavaScript, we recommend it together with the Node.js runtime as the ideal candidate for development of WebSocket API subscription and event handling module. We also support this recommendation with the fact that there is a really strong community behind both JavaScript and Node.js.

Comparing the final programming language/library score of 3.53 for Python and 3.27 for PHP, it is obvious that they had approximately the same performance during the test. The score between 3 and 4 is not bad. These languages with their implementations and WebSocket libraries are offered as an alternative decision for further development of trading system communication components. Despite they have undoubtedly many advantages, e.g. speed of producing the source code which is fat better compared to C++, we would not recommend them for high-performance tasks where high speed and low latency are the main criteria referring to the setup used within our experiment.

Based on the announcement of PHP release managers, PHP version 8.0 will be officially released on November 26, 2020 [68]. The most acclaimed future coming with this new version is completely new JIT compiler which promises significantly better performance for numerical code and slightly better performance for typical web application. The performance comparison of PHP 7.2 used within the experiment and new PHP 8.0 in context of receiving events from cryptocurrency exchanges utilizing WebSocket protocol may give interesting results. The PyPy implementation of Python which already involves pure JIT techniques has been measured 4.4x faster compared to CPython [69], which we used within our test. The inclusion of PyPy into next experiment would produce other valuable results. We also propose to extend the WebSocket's data stream subscription method with the possibility of sending separate WebSocket JSON payload request what was not feasible with current version of WebSocket connector which only supported the subsription method via WebSocket header values provided in initial WebSocket handshake packet. This will allow many other cryptocurrency exchanges to be involved in the test. The expansion of the tested sample of programming libraries is also expected within future developments.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

This appendix summarizes requirements for the content and format of the academic article created as a part of the master's project in computing for the potential submission to the Computers journal. Incorporation of these requirements into appendix is set out in the assessment grid for MSc research paper. The Computers journal article template states that all appendix sections must be cited in the main text. In this case, Appendix A is logically not connected with the research and therefore it is not cited in the main text.

Aims

Computers (ISSN 2073-431X) is an international, open access journal which provides an advanced forum for computer sciences. It publishes reviews, regular research papers and short communications. Our aim is to encourage scientists to publish their experimental and theoretical results in as much detail as possible. There is no restriction on the length of the papers. The full experimental details must be provided so that the results can be reproduced.

Subject Areas

- computer science & engineering
- computers and computation
- information systems
- software, graphics, programming
- computer software
- computer networking and Internet
- computer programming; programming languages
- communications and control
- etc.

Types of Publications

Computers has no restrictions on the length of manuscripts, provided that the text is concise and comprehensive. Full experimental details must be provided so that the results can be reproduced. *Computers* requires that authors publish all experimental controls and make full datasets available where possible (see the guidelines on Supplementary Materials and references to unpublished data).

Articles: Original research manuscripts. The journal considers all original research manuscripts provided that the work reports scientifically sound experiments and provides a substantial amount of new information. Authors should not unnecessarily divide their work into several related manuscripts, although Short *Communications* of preliminary, but significant, results will be considered. Quality and impact of the study will be considered during peer review.

General Considerations

Research manuscripts should comprise:

- ***Front matter:*** Title, Author list, Affiliations, Abstract, Keywords
- ***Research manuscript sections:*** Introduction, Results, Discussion, Materials and Methods, Conclusions (optional).
- ***Back matter:*** Supplementary Materials, Acknowledgments, Author Contributions, Conflicts of Interest, References.

Abbreviations should be defined in parentheses the first time they appear in the abstract, main text, and in figure or table captions and used consistently thereafter.

SI Units (International System of Units) should be used. Imperial, US customary and other units should be converted to SI units whenever possible.

Equations: If you are using Word, please use either the Microsoft Equation Editor or the MathType add-on. Equations should be editable by the editorial office and not appear in a picture format.

Front Matter

These sections should appear in all manuscript types.

Title: The title of your manuscript should be concise, specific and relevant. It should identify if the study reports (human or animal) trial data, or is a systematic review, meta-analysis or replication study.

Author List and Affiliations: Authors' full first and last names must be provided. The initials of any middle names can be added. The PubMed/MEDLINE standard format is used for affiliations: complete address information including city, zip code, state/province, and country. At least one author should be designated as corresponding author, and his or her email address and other details should be included at the end of the affiliation section. Please read the criteria to qualify for authorship.

Abstract: The abstract should be a total of about 200 words maximum. The abstract should be a single paragraph and should follow the style of structured abstracts, but without headings: 1) Background: Place the question addressed in a broad context and highlight the purpose of the study; 2) Methods: Describe briefly the main methods or treatments applied. Include any relevant preregistration numbers, and species and strains of any animals used. 3) Results: Summarize the article's main findings; and 4) Conclusion: Indicate the main conclusions or interpretations. The abstract should be an objective representation of the article: it must not contain results which are not presented and substantiated in the main text and should not exaggerate the main conclusions.

Keywords: Three to ten pertinent keywords need to be added after the abstract. We recommend that the keywords are specific to the article, yet reasonably common within the subject discipline.

Research Manuscript Sections

Introduction: The introduction should briefly place the study in a broad context and highlight why it is important. It should define the purpose of the work and its significance, including specific hypotheses being tested. The current state of the research field should be reviewed carefully and key publications cited. Please highlight controversial and diverging hypotheses when necessary. Finally, briefly mention the main aim of the work and highlight the main conclusions. Keep the introduction comprehensible to scientists working outside the topic of the paper.

Materials and Methods: They should be described with sufficient detail to allow others to replicate and build on published results. New methods and protocols should be described in detail while well-established methods can be briefly described and appropriately cited. Give the name and version of any software used and make clear whether computer code used is available. Include any pre-registration codes.

Results: Provide a concise and precise description of the experimental results, their interpretation as well as the experimental conclusions that can be drawn.

Discussion: Authors should discuss the results and how they can be interpreted in perspective of previous studies and of the working hypotheses. The findings and their implications should be discussed in the broadest context possible and limitations of the work highlighted. Future research directions may also be mentioned. This section may be combined with Results.

Conclusions: This section is not mandatory, but can be added to the manuscript if the discussion is unusually long or complex.

Back Matter

Conflicts of Interest: Authors must identify and declare any personal circumstances or interest that may be perceived as inappropriately influencing the representation or interpretation of reported research results. If there is no conflict of interest, please state "The authors declare no conflict of interest." Any role of the funding sponsors in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript, or in the decision to publish the results must be declared in this section. If there is no role, please state “The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results”. For more details please see Conflict of Interest.

References: References must be numbered in order of appearance in the text (including table captions and figure legends) and listed individually at the end of the manuscript. We recommend preparing the references with a bibliography software package, such as EndNote, Reference

Manager or Zotero to avoid typing mistakes and duplicated references. We encourage citations to data, computer code and other citable research material. If available online, you may use reference style 9. below.

Citations and References in Supplementary files are permitted provided that they also appear in the main text and in the reference list.

In the text, reference numbers should be placed in square brackets [ ], and placed before the punctuation; for example [1], [1–3] or [1,3]. For embedded citations in the text with pagination, use both parentheses and brackets to indicate the reference number and page numbers; for example [5] (p. 10). or [6] (pp. 101–105).

The reference list should include the full title, as recommended by the ACS style guide. Style files for Endnote and Zotero are available.

References should be described as follows, depending on the type of work:

Journals or Articles:

1. Author 1, A.B.; Author 2, C.D. Title of the article. *Abbreviated Journal Name* **Year**, *Volume*, page range.

Books and Book Chapters:

2. Author 1, A.; Author 2, B. *Book Title*, 3rd ed.; Publisher: Publisher Location, Country, 2008; pp. 154–196.
3. Author 1, A.; Author 2, B. Title of the chapter. In *Book Title*, 2nd ed.; Editor 1, A., Editor 2, B., Eds.; Publisher: Publisher Location, Country, 2007; Volume 3, pp. 154–196.

Unpublished work, submitted work, personal communication:

4. Author 1, A.B.; Author 2, C. Title of Unpublished Work. status (unpublished; manuscript in preparation).
5. Author 1, A.B.; Author 2, C. Title of Unpublished Work. *Abbreviated Journal Name* stage of publication (under review; accepted; in press).
6. Author 1, A.B. (University, City, State, Country); Author 2, C. (Institute, City, State, Country). Personal communication, 2012.

Conference Proceedings:

7. Author 1, A.B.; Author 2, C.D.; Author 3, E.F. Title of Presentation. In Title of the Collected Work (if available), Proceedings of the Name of the Conference, Location of Conference, Country, Date of Conference; Editor 1, Editor 2, Eds. (if available); Publisher: City, Country, Year (if available); Abstract Number (optional), Pagination (optional).

Thesis:

8. Author 1, A.B. Title of Thesis. Level of Thesis, Degree-Granting University, Location of University, Date of Completion.

Websites:

9. Title of Site. Available online: URL (accessed on Day Month Year).

References

1. Farell, R. An Analysis of the Cryptocurrency Industry. Wharton University of Pennsylvania, 2015. [CrossRef]
2. Vo, A.; Yost-Bremm, C. A High-Frequency Algorithmic Trading Strategy for Cryptocurrency. *J. of Comp. Inf. Sys.* **2018**, pp.1-14. [CrossRef]
3. Păuna, C. Arbitrage Trading Systems for Cryptocurrencies. Design Principles and Server Architecture. *Inf. Econ.* **2018**, *22(2)*, pp.35-42. [CrossRef]
4. Liu, J.; Serletis, A. Volatility in the Cryptocurrency Market. *O. Econ. Rev.* **2019**, *30(4)*, pp.779-811. [CrossRef]
5. Fang, F.; Ventre, C.; Basios, M.; Kong, H.; Kanthan, L.; Martines-Rego, D.; Wu, F.; Li, L. Cryptocurrency Trading: A Comprehensive Survey. *Elsevier* under review. [CrossRef]

6. Treleaven, P.; Galas, M.; Lalchand, V. Algorithmic trading review. *Comm. of the ACM* **2013**, *56(11)*, pp.76-85. [CrossRef]
7. Mendes, A. Algorithmic and high-frequency trading strategies: A literature review. *MAGKS J. Disc. Pap. Ser. in Econ*. **2016**, 25-2016. [CrossRef]
8. Miller, R.; Shorter, G. High Frequency Trading: Overview of Recent Developments. Congressional Research Service 2016, R44443. [CrossRef]
9. Hasbrouck, J.; Saar, G. Low-latency trading. *J. of Fin. Mark.* **2013**, *16(4)*, pp.646-679. [CrossRef]
10. Prechelt, L. An empirical comparison of seven programming languages. *Comp.* **2000**, *33(10)*, pp.23-29. [CrossRef]
11. Nanz, S.; Furia, C.A. A Comparative Study of Programming Languages in Rosetta Code. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16-24 May 2015. [CrossRef]
12. Kochhar, P.; Wijedasa, D.; Lo, D. A Large Scale Study of Multiple Programming Languages and Code Quality. In IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, 14-18 March 2016. [CrossRef]
13. Vergel Eleuterio, P.; Thukral, L. Programming Language Choices for Algo Traders: The Case of Pairs Trading. *Comp. Econ.* **2018**, *53(4)*, pp.1443-1449. [CrossRef]
14. Ma, K.; Sun, R. Introducing WebSocket-Based Real-Time Monitoring System for Remote Intelligent Buildings. *Int. J. of Distr. Sens. Net.* **2013**, 9(12). [CrossRef]
15. Puranik, D.; Feiock, D.; Hill, J. Real-Time Monitoring using AJAX and WebSockets. In 2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), Scottsdale, AZ, USA, 22-24 April 2013. [CrossRef]
16. Skvorc, D.; Horvat, M.; Srbljic, S. Performance evaluation of Websocket protocol for implementation of full-duplex web streams. In 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 26-30 May 2014. [CrossRef]
17. Ma, K.; Zhang, W. Introducing browser-based high-frequency cloud monitoring system using WebSocket proxy. *Int. J. of Grid and Util. Comp.* **2015**, *6(1)*, pp.21-29. [CrossRef]
18. Babovic, Z.; Protic, J.; Milutinovic, V. Web Performance Evaluation for Internet of Things Applications. *IEEE Access* **2016**, *4*, pp.6974-6992. [CrossRef]
19. Guyang, W.; Shulin, Y.; Xuelei, R.; Bin, W. Research on WebSocket-Based Authentication System. In Proceedings of the 2017 VI International Conference on Network, Communication and Computing (ICNCC 2017), Kunming, China, 8-10 December 2017; pp.102-105. [CrossRef]
20. Shen, K.; Si, Z.; Zhang, L. Research and Implement of HTML5 Game Based on WebSocket. *Lect. Not. in El. Eng.* **2017**, pp.423-428. [CrossRef]
21. Rahmatulloh, A.; Darmawan, I.; Gunawan, R. Performance Analysis of Data Transmission on WebSocket for Real-time Communication. In 2019 16th International Conference on Quality in Research (QIR): International Symposium on Electrical and Computer Engineering, Padang, Indonesia, 22-24 July 2019. [CrossRef]
22. Imre, G.; Mezei, G. Introduction to a WebSocket benchmarking infrastructure. In 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC), Novi Sad, Serbia, 1-2 June 2016. [CrossRef]
23. Wang, Y.; Huang, L.; Liu, X.; Sun, T.; Lei, K. Performance Comparison and Evaluation of WebSocket Frameworks: Netty, Undertow, Vert.x, Grizzly and Jetty. In 2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN), Shenzhen, China, 15-17 August 2018. [CrossRef]
24. Goldstein, M.; Kumar, P.; Graves, F. Computerized and High-Frequency Trading. *Fin. Rev.* **2014**, *49(2)*, pp.177-202. [CrossRef]
25. Popławski, P. Connectivity Solutions in Automated Trading. *Int. J. of Elect. and Telecom.* **2015**, *61(4)*, pp.403-408. [CrossRef]
26. Huang, B.; Huan, Y.; Xu, L.; Zheng, L.; Zou, Z. Automated trading systems statistical and machine learning methods and hardware implementation: a survey. *Ent. Inf. Sys.* **2018**, *13(1)*, pp.132-144. [CrossRef]
27. Borağan Aruoba, S.; Fernández-Villaverde, J. A comparison of programming languages in macroeconomics. *J. of Econ. Dyna. and Cont.* **2015**, *58*, pp.265-273. [CrossRef]
28. The WebSocket Protocol. Available online: https://tools.ietf.org/html/rfc6455 (accessed on 17 March 2020).
29. NTP Pool Project. Available online: https://www.ntppool.org/en/ (accessed on 10 April 2020).
30. SQLite. Available online: https://www.sqlite.org (accessed on 10 April 2020).

31. Umre, J.; Batra; K.; Vaidya, V. Comparative Performance Analysis of Mysql and Sqlite Relational Database Management Systems in Windows10 Environment. *Int. J. of Lat. Tr. in Eng. and Tech.* **2017**, *8(1)*, pp.342-349. [CrossRef]
32. GNU Screen. Available online: https://www.gnu.org/software/screen/ (accessed on 27 August 2020).
33. The RedMonk Programming Language Rankings: January 2020. Available online: https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20 (accessed on 12 April 2020).
34. Ray, B.; Posnett, D.; Devanbu, P.; Filkov, V. A large-scale study of programming languages and code quality in GitHub. *Comm. of the ACM* **2017**, *60(10)*, pp.91-100. [CrossRef]
35. Amaral, V.; Norberto, B.; Goulão, M.; Aldinucci, M.; Benkner, S.; Bracciali, A.; Carreira, P.; Celms, E.; Correia, L.; Grelck, C.; Karatza, H.; Kessler, C.; Kilpatrick, P.; Martiniano, H.; Mavridis, I.; Pllana, S.; Respício, A.; Simão, J.; Veiga, L.; Visa, A. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Par. Comp.* **2020**, *91*, 102584. [CrossRef]
36. µWebSockets. Available online: https://github.com/uNetworking/uWebSockets (accessed on 11 July 2020).
37. Gorilla WebSocket. Available online: https://github.com/gorilla/websocket (accessed on 11 July 2020).
38. Java WebSockets. Available online: https://github.com/TooTallNate/Java-WebSocket (accessed on 11 July 2020).
39. Ws: a Node.js WebSocket library. Available online: https://github.com/websockets/ws (accessed on 11 July 2020).
40. Php-wss. Available online: https://github.com/arthurkushman/php-wss (accessed on 11 July 2020).
41. Python websockets. Available online: https://github.com/aaugustin/websockets (accessed on 11 July 2020).
42. Magic Quadrant for Cloud Infrastructure as a Service, Worldwide. Available online: https://www.gartner.com/doc/reprints?id=1-1CMAPXNO&ct=190709&st=sb (accessed on 13 April 2020).
43. Coinbase Case Study. Available online: https://aws.amazon.com/solutions/case-studies/coinbase (accessed on 13 April 2020).
44. Regions, Availability Zones, and Local Zones. Available online: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (accessed on 13 April 2020).
45. Ben-David, Y.; Hasan, S.; Pearce, P. Location Matters: Limitations of Global-Scale Datacenters. Berkeley: The Department of Electrical Engineering & Computer Sciences, University of California, 2011. [CrossRef]
46. Garvey, R.; Wu, F. Speed, distance, and electronic trading: New evidence on why location matters. *J. of Fin. Mark.* **2010**, *13(4)*, pp.367-396. [CrossRef]
47. Adrian, J. Informational Inequality: How High Frequency Traders Use Premier Access to Information to Prey on Institutional Investors. Duke Law School, Duke University, 2015. [CrossRef]
48. Top Cryptocurrency Spot Exchanges. Available online: https://coinmarketcap.com/rankings/exchanges/ (accessed on 15 April 2020).
49. Cryptocurrency Exchange Industry Research Report. Available online: https://tokeninsight.com/api/upload/levelPdf/34f311be77328eaf6caaff890a97daf6.pdf (accessed on 14 April 2020).
50. Crypto Exchange Volume Ranking. Available online: https://www.worldcoinindex.com/exchange (accessed on 15 April 2020).
51. Investigation into the Legitimacy of Reported Cryptocurrency Exchange Volume. Available online: https://ftx.com/volume-report-paper.pdf (accessed on 14 April 2020).
52. Web Sockets Streams for Binance. Available online: https://github.com/binance-exchange/binance-official-api-docs/blob/master/web-socket-streams.md (accessed on 7 July 2020).
53. Bithumb WebSocket Document. Available online: https://github.com/bithumb-pro/bithumb.pro-official-api-docs/blob/master/ws-api.md (accessed on 7 July 2020).
54. Gemini WebSocket API Reference. Available online: https://docs.gemini.com/websocket-api/ (accessed on 7 July 2020).
55. Performance Testing Guidance for Web Applications. Available online: https://pdfs.semanticscholar.org/a2ff/c8cca5b3aa3302dcb3a05517e8c763314a1f.pdf?_ga=2.10551412.196933797.1585508934-1365939409.1585508934 (accessed on 19 April 2020).
56. Jain, R. Introduction to Experimental Design. In *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling*. John Wiley & Sons: New York, USA, 1991.

57. TokuDB Introduction. Available online: https://www.percona.com/doc/percona-server/LATEST/tokudb/tokudb_intro.html (accessed on 5 August 2020).
58. Long, S. A Comparative Analysis of the Application of Hashing Encryption Algorithms for MD5, SHA-1, and SHA-512. In 3rd International Conference on Electrical, Mechanical and Computer Engineering, Guizhou, China, 9-11 August 2019. [**CrossRef**]
59. Rasjid, Z.E.; Soewito, B; Witjaksono, G; Abdurachman, E. A Review of Collisions in Cryptographic Hash Function Used in Digital Forensic Tools. In 2nd International Conference on Computer Science and Computational Intelligence 2017, Bali, Indonesia, 13-14 October 2017. [**CrossRef**]
60. Univariate Analysis and Normality Test Using SAS, STATA, and SPSS. Available online: http://cef-cfr.ca/uploads/Reference/sasNORMALITY.pdf (accessed on 7 August 2020).
61. Razali, N.M.; Wah, Y.B. Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests. Malaysia Institute of Statistics, Faculty of Computer and Mathematical Sciences, Universiti Teknologi MARA (UiTM), 2010. [**CrossRef**]
62. Statistical functions (scipy.stats). Available online: https://docs.scipy.org/doc/scipy/reference/stats.html (accessed on 7 August 2020).
63. Han, J.; Kamber, M.; Pei, J. Getting to Know Your Data. In *Data Mining: Concepts and Techniques*, 3rd ed.; Morgan Kaufmann Publishers: Burlington, Massachusetts, USA, 2012; pp.39-82. [**CrossRef**]
64. Visualizing Samples with Box Plots. Available online: https://www.nature.com/articles/nmeth.2813.pdf (accessed on 8 August 2020).
65. A Gentle Introduction to Data Visualization Methods in Python. Available online: https://machinelearningmastery.com/data-visualization-methods-in-python/ (accessed on 8 August 2020).
66. Matplotlib.cbook. Available online: https://matplotlib.org/3.1.1/api/cbook_api.html#matplotlib.cbook.boxplot_stats (accessed on 8 August 2020).
67. Seaborn.boxplot. Available online: https://seaborn.pydata.org/generated/seaborn.boxplot.html (accessed on 8 August 2020).
68. PHP 8.0 Preparation Tasks. Available online: https://wiki.php.net/todo/php80 (accessed on 17 August 2020).
69. PyPy. Available online: https://www.pypy.org/ (accessed on 17 August 2020).