information

Article

Software Platform for the Comprehensive Testing of Transmission Protocols Developed in GNU Radio

Mihai Petru Stef [1] and Zsolt Alfred Polgar [2,*]

[1] Independent Researcher, 400489 Cluj Napoca, Romania; mihai.stef@com.utcluj.ro
[2] Communications Department, Technical University of Cluj Napoca, 400114 Cluj Napoca, Romania
* Correspondence: zsolt.polgar@com.utcluj.ro; Tel.: +40-264401226

Abstract: With the constant growth of software-defined radio (SDR) technologies in fields related to wireless communications, the need for efficient ways of testing and evaluating the physical-layer (PHY) protocols developed for these technologies in real-life traffic scenarios has become more critical. This paper proposes a software testbed that enhances the creation of network environments that allow GNU radio applications to be fed with test traffic in a simple way and through an interoperable interface. This makes the use of any traffic generator possible—existing ones or one that is custom-built—to evaluate a GNU radio application. In addition, this paper proposes an efficient way to collect PHY-specific monitoring data to improve the performance of the critical components of the message delivery path by employing the protocol buffers library. This study considers the entire testing and evaluation ecosystem and demonstrates how PHY-specific monitoring information is collected, handled, stored, and processed as time series to allow complex visualization and real-time monitoring.

Keywords: software testing; GNU radio; software-defined radio; communication protocols; networking environment; monitoring messaging; protocol buffer; performance evaluation

Citation: Stef, M.P.; Polgar, Z.A. Software Platform for the Comprehensive Testing of Transmission Protocols Developed in GNU Radio. *Information* **2024**, *15*, 62. https://doi.org/10.3390/info15010062

Academic Editors: Webert Montlouis, Agbotiname Lucky Imoize and Cheng-Chi Lee

Received: 22 December 2023
Revised: 12 January 2024
Accepted: 16 January 2024
Published: 20 January 2024

1. Introduction

Software testing is an important component of the software development process and it is a significant part of software engineering. It assumes the role of ensuring that a software product fulfills its functional requirements, is free from defects and errors, and is of good quality [1,2]. The quality of a software product depends on several parameters, such as the response time, performance, reliability, maintainability, correctness, testability, usability, and reusability, to mention just a few. Software testing is time-consuming, and 40–50% of a project's budget (in some cases, even 80% [3]) can be spent on this operation according to [1,2,4]. Nonetheless, researchers have shown [2,4] that software testing is not a "silver bullet" that can guarantee the high quality of a software product. Complete testing, i.e., discovering and fixing all errors, is practically impossible because the testing process cannot be exhaustive. The number of tests that can be performed is limited by several factors, such as the input domain being too large, there being too many possible paths, and specifications being difficult to test [2]. As an important activity in software development, the testing process should be carried out smoothly [5], and it should start in the early phase of the project to avoid costs related to failed software afterward [3,6].

A fundamental issue related to testing is the generation of good test cases [6] that can be used find the errors and faults in a minimum amount of time and with minimum effort but with a high probability. The data obtained through testing are an indicator of software's reliability and quality, but the total absence of defects cannot be guaranteed. From the points of view of both software development and testing, the use of appropriate environments is also very important [7]. These environments can be very different due to differences in operating systems, databases, network servers, application services, etc. An integrated

management tool that allows the development of test scenarios and assignment of test cases, such as the tool proposed in [7], could be helpful in performing testing operations. The importance of frameworks for test execution to improve the quality of software testing was underlined in [8]. Frameworks allow repeatable tests and make automated testing easier. Frameworks can also provide a standard way of performing the main parts of the testing process: setting the initial state, invoking the functionality being tested, checking the results of the test, and performing any necessary cleanup.

Many testing techniques focus on testing functional correctness (debug testing), but performance issues are also very important in software testing, especially in cases such as web services, real-time hardware systems, and industrial systems and processes [8]. Researchers [9] have discussed various issues related to the fundamentals of software testing and shown that software testing is much more than error detection or debugging. It has been demonstrated [10] that after the release of software products, the main problems are related to performance degradation or providing the required throughput, and system crashes and incorrect system responses are usually secondary issues, as the software is extensively tested before release from the functional point of view. Performance testing involves issues such as resource usage, throughput, stimulus response time, queue length, bandwidth requirements, CPU (central processing unit) cycles, and database access. Issues such as scalability and the ability to handle heavy workloads should also be considered.

Testing communication protocols and software components used by communication equipment raise several critical issues, such as real-time processing constraints, timing and synchronization between intercommunicating modules and processes, strong interactions between software and hardware components, the hardware platforms' need for testing complex protocols and signal processing, remote access to the platform on which the tested software is running, and the need to process a large amount of data generated in tests, to mention just a few. The testing of communication protocols used in specific applications requires specific test suites due to the complexity and requirements imposed on these protocols. This is represented by the testing of communication protocols used in wireless communications, which is one of the most challenging testing operations. The conclusions researchers drew when testing transmission protocols used by mobile military networks under real-life conditions are presented in [11]. The main problems encountered during test operations were the timing constraints, test controllability, inconsistency detection, and conflicting timers.

The real-life testing of communication protocols in wireless communication scenarios with general and transmission techniques, in particular, requires the use of dedicated hardware platforms adapted to the specific test scenarios. Researchers [12] have presented a hardware platform that included DSPs (digital signal processors), FPGAs (field-programmable gate arrays), and SDR (software-defined radio) interfaces for the prototyping and testing of complex radio transceivers, such as OFDM (orthogonal frequency-division multiplexing) transceivers. Other researchers [13] have proposed a mobile platform based on universal software radio peripheral (USRP) SDR devices for testing algorithms for radio transmitters and receivers, while other still have presented signal processing algorithms and design and testing methodologies related to the implementation of radio transceivers using the concept of SDR [14].

Developing communication protocols for radio transmission systems is not a trivial task. SDR technology and open-source development libraries such as GNU radio [15] come with several tools that ease the development of PHY (physical)- and MAC (medium access control)-layer protocols for wireless transmission systems. GNU radio offers an extended library of signal processing modules that are necessary for developing, testing, and evaluating PHY- and MAC-layer communication protocols, but support is also provided for network- and transport-layer protocols.

Testing, evaluating, and troubleshooting communication protocols used in wireless communication systems is challenging. The results of many studies on this topic (see Section 2) have shown that effective automation of test suites, generating appropriate

traffic, considering timing constraints, real-time handling of monitoring data, and storage and visualization/analysis of gathered data are some of the main issues related to testing these protocols.

Many papers have presented various testing platforms adapted to specific protocols and scenarios like military or industrial communication systems. This paper proposes a generic software testing platform (testbed) for wireless communication protocols developed in GNU radio.

The proposed platform provides a solution for isolating the application under test and feeding the application with any data traffic controlled by IP or Ethernet protocols, the types of traffic used in most real-life scenarios. This feature makes it interoperable with other traffic generators and network analysis tools, which is convenient for generating different user data flows. The proposed testing platform is designed in such a way to efficiently collect, store, and analyze large amounts of real-time data. These features are essential for testing physical-layer protocols. They enable various testing operations such as functional testing, conformance testing, quality evaluation testing, and comprehensive dynamic testing. The platform can perform the mentioned tests in simulated and real-life conditions when SDR interfaces are used for communication. The entire setup and management of the testbed are implemented in Python, which means they can be easily integrate into an automation testing framework. Additionally, the platform allows dynamic reconfiguration of the application under testing through JSON (JavaScript Object Notation) objects if its implementation supports it.

In summary, this paper's contributions are the architecture and the design of a generic, practical, flexible, and scalable testing platform (testbed) for GNU radio applications that allows for the integration of various software and networking tools necessary to test and evaluate time-constrained complex signal processing applications. In addition, this paper proposes and analyzes several efficient methods to build, serialize, and parse monitoring messages from GNU radio applications. The structure of this paper is as follows: Section 2 presents a short review of the technical literature related to software testing classification and principles, relating to specific issues raised by testing communication protocols, in general, and by testing wireless communication protocols, in particular. Section 3 presents in detail the proposed software testing platform, the architecture of the systems, the networking environments proposed for sending test data to the GNU radio application under test, and the real-time monitoring data acquisition and handling solutions. Section 4 presents our experimental results and a discussion of these results, and Section 5 concludes this paper.

2. Related Work

The process of software testing can be classified in various ways, but the three main testing techniques are as follows [2]:

- Black Box Testing: this technique is based on the requirements and specifications of the software under test, and there is no need to examine the program's code. The tester only knows the set of inputs and predictable outputs.
- White Box Testing: this technique mainly focuses on the internal logic and structure of the program's code. The tester has total knowledge of the program structure, and with this technique, it is possible to test every branch and decision in the program.
- Grey Box Testing: this technique combines the benefits of black box and white box testing. It attempts and generally succeeds in achieving optimal testing outcomes. The software testing process has several phases and goals. As a result, there are a wide range of testing categories that can be identified. A more detailed categorization of the software testing process can be found in [3]. These categories are:
- Acceptance Testing: this category of testing is performed to determine whether the system or software is acceptable.
- Ad Hoc Testing: this type of testing is performed without planning or documentation. The goal is to find errors that were not detected by other types of testing.

- Alpha and Beta Testing: alpha testing is performed at the development site after acceptance testing, while beta testing is carried out in a real test environment.
- Automated Testing: this involves using automated tools to write and execute test cases.
- Integration Testing: the testing of individual units is grouped together, and the interface between these units is tested.
- Regression Testing: the test cases from existing test suites are rerun to demonstrate that software changes have no unintended side effects.
- Stress Testing: this testing determines the robustness of software by forcing the functioning of software modules beyond the limits of normal operation.
- User Acceptance Testing: this testing is performed by the end users of the software. It happens in the final phase of the testing process.
- Security Testing: this testing checks the ability of the software to prevent unauthorized access to resources and data.

Many studies have addressed the categorization of testing techniques. In addition to the categories mentioned above, random testing, functional testing, control flow testing, data flow testing, and mutation testing techniques have been identified in one such study [16]. Meanwhile, another study [17] introduces the terms "static" and "dynamic" testing and analyzes the use of testing terminology in several testing techniques.

Generating suitable test suites is a problem that has been extensively studied. According to some researchers [18], a good test suite detects real faults, and only a small number of representative use cases can be selected from a larger category of use cases [7]. Additionally, it has been shown that many errors occur at the boundaries of the input and output ranges, meaning that test cases should focus on boundary conditions.

Testing software, systems, or networks can present specific challenges depending on the requirements and how they function. For instance, testing software for systems based on service-oriented architectures (SOAs) can be challenging due to system distribution, controllability, and observability issues, as discussed in [19]. In [20], the authors propose an approach to generate tests for error-handling routines in programmable logic controllers (PLCs) used in industrial environments, ensuring the industrial process's reliability. Researchers have also considered testing large and complex network topologies with limited resources, as discussed in [21], and proposed an emulator that can run on a single virtual machine. This system is specifically designed for software-defined networks (SDNs) and OpenFlow research. The testing of cloud-based systems and cloud technologies is covered in [22], which presents a systematic literature review on the topic.

Similarly, researchers have considered testing software in systems with stringent reliability requirements, such as digital control systems integrated into nuclear plant safety software, as discussed in [23], and proposed building a real platform and a specific testing strategy. Testing embedded software is also important, especially in safety-critical domains such as automotive or aviation, as discussed in [24]. Testing embedded systems should consider limited memory, CPU usage, energy consumption, real-time processing, and the strong interaction between hardware and software. Finally, ref. [25] proposes a framework that allows test suites to detect synchronization faults in testing embedded systems.

Testing communication protocols is an important process and has been the subject of many papers [11,26–29]. Formal specifications, which often use an extended finite state machine model, can be used to design protocol testing, as described in [26]. However, to detect syntactic and semantic errors and validate the protocol design, both the control and data flow of the protocol must be considered. In [27], testing of communication protocols designed according to the OSI (open systems interconnection) model is discussed. The paper demonstrates that efficient test case generation algorithms are essential for successful testing. The importance of conformance testing in the context of the rapid development of communication protocols, which may generate many incompatible implementations, is emphasized in [28]. The paper also suggests that while automation of the testing process is desirable, it is difficult to achieve for complex models. Ref. [29] presents a survey concerning the testing of communication protocols, highlighting the importance of conformance testing,

as implementations derived from the same protocol standard can be very different. The paper also discusses the difficulty of generating suitable test suites and sequences, particularly in real-life testing. Additionally, the paper points out that the number of states of a complex protocol implementation could be vast, making exhaustive testing impractical. As a result, many testing environments/frameworks have been implemented and reported in various studies to allow more efficient, reliable, and flexible testing and evaluation of communication protocols. The issue of test case generation in communication protocol testing is also discussed in [30]. The paper analyzes and experiments with several testing methods to evaluate quality indicators such as fault detection capability, applicability, complexity, and testing time. A survey concerning the testing of control and data flows and the time aspects of communication systems is presented in [31], which also covers the generation of test suites that can detect the maximum number of errors at a minimum cost. Various research papers have discussed the testing of specific communication protocols [32–34]. In [32], the focus is on testing communication protocols used between the charging equipment and the battery management system of an electric car. In this case, the primary concern is the consistency of communication between the two devices mentioned above. In [33], the paper presents the testing of the LIN (local interconnect network) protocol used for interconnecting vehicles' electronic systems. The article examines the issues related to the conformance testing of the LIN protocol, some of which are also applicable to other link layer protocol testing. Additionally, in [34], the testing of industrial communication protocols is considered, and the paper evaluates some of the most commonly used industrial communication protocols from the software perspective.

Several researchers have also considered the testing of communication protocols in challenging wireless communication systems. In [35], the testing of military systems and applications in different communication scenarios is discussed, including changing network conditions and data flow parameters. The paper proposes a test platform that uses reproducible test methodologies to allow automated testing of military systems and applications over actual military radio equipment. Moreover, ref. [36] proposes a software testing method to evaluate the applicability, reliability, and durability of various communication equipment used in maritime satellite communications.

Researchers [37] have developed an open-access wireless testing platform for testing communication protocols used in wireless communication networks. This platform comprises a large grid of ceiling-mounted antennas connected to programmable SDR devices operating at frequencies lower than 6 GHz. The system provides computational power and hardware support for testing complex communication systems and protocols such as MIMO (multiple-input and multiple-output) communication systems, cognitive radio, 5G cellular networks, IoT (Internet of Things), and more. A multiple-antenna evaluation and testing platform was proposed in [38]. The platform includes FPGAs, DSPs, and control processors to provide the necessary processing power and flexibility. The platform interfaces require high throughput since evaluating multiple antennas generates a large amount of real-time data.

3. Platform for Real-Life Testing of Communication Protocols

3.1. Testbed Environment

The primary objective of the testing platform that has been developed is to enable the testing and evaluation of communication protocols that have been implemented in GNU radio in real traffic scenarios. Conducting tests in such circumstances is vital for verification, validation, and acceptance testing. It is also helpful for dynamic white box testing that is carried out to assess the quality indicators of the implemented protocols. An environment was created to run and evaluate GNU radio applications in real traffic scenarios. This environment allows network traffic to be sent through the GNU radio application under test. Linux network namespaces isolate the environment, making managing multiple environments on the same or multiple machines easier. This is particularly useful when

multiple SDR-based applications are running on a remote server, like in the case of the test platforms described in [37,38].

The block diagram in Figure 1 shows that TUN/TAP interfaces [39] are used to channelize the data traffic to or from the GNU radio application under test. Depending on its characteristics, the application under test is connected to one or several TUN/TAP interfaces. The framework supports two types of environments: one that allows testing of GNU radio applications in simulated conditions and another that allows testing when SDR boards are connected in the transmission chain.

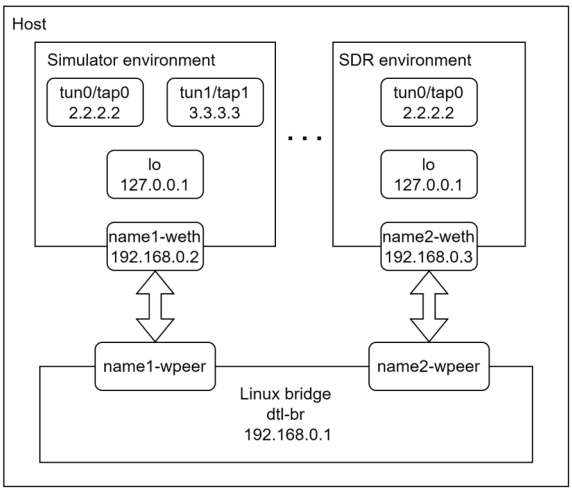

Figure 1. Schematic of the environment encapsulating the application under test.

In simulated conditions, both transceivers run in the same environment, while in a real transmission scenario involving real channels, they run in separate environments and possibly on different machines. For example, in a simplex simulated transmission, the transmitter application reads the data from one interface, and the receiver application writes the output data on the other interface. In a duplex scenario, the transmitter and receiver read and write data to their interface.

In the concrete case presented in Figure 1, the tested application connects to bidirectional tun0/tap0 and tun1/tap1 interfaces (in the left-side positioned environment) or only to the tun0/tap0 interface (in the right-side positioned environment).

In order to allow applications running in the environment to access the internet, all environments are connected to the host through a network bridge (i.e., dtl-br). The network bridge [40] is configured on the host machine, and for each environment, a pair of virtual interfaces (i.e., *-weth and *-wpeer) are set up. One interface is attached to the environment, and the other is attached to the bridge. Since Layer 2 connectivity is established between the environments and the host, only network addresstTranslation (NAT) of packets originating from the environments is required. This is illustrated in Figure 2.

```
$ iptables-legacy -t nat -L POSTROUTING
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.0.0/24       anywhere
```

Figure 2. Applying netfilter rules for NAT at the gateway.

To confirm that traffic passes through the bridge to the host and beyond the internet, we run a traceroute command in our environment (refer to Figure 3) to a public IP address outside our network (i.e., 8.8.8.8).

```
$ ip netns  exec test_tap traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
 1  192.168.0.1 (192.168.0.1)   0.043 ms   0.004 ms   0.003 ms
 2  LAPTOP-0M533AB8 (172.31.240.1)   0.374 ms   0.354 ms   0.349 ms
…
12  dns.google (8.8.8.8)   35.817 ms   37.917 ms   38.123 ms
```

Figure 3. Executing the traceroute command in the environment to a public IP address.

To ensure that the test traffic passes through the application under test, the routing rules need to be manipulated as follows:

1. It is important to ensure that the outgoing traffic through the TUN/TAP interfaces of the system is not routed through the loopback interface as the application is interacting with these interfaces. To do this, the default local routes associated with these interfaces should be removed.
2. The kernel uses the local routing table for incoming (ingress) traffic to decide if a packet is addressed to the local host. Therefore, it is necessary to create an alternative routing rule that is only used for ingress traffic. To achieve this, distinct routing decisions should be used on input and output paths, and a routing rule should be created that matches only for ingress traffic, packets with the "input interface" attribute (i.e., iif), and performs the route lookup in a custom "local" routing table (see Figure 4).
3. To pass traffic through the GNU radio application under test, two entries are created in the main routing table that routes the traffic destined to the far end of the "tunnel" (e.g., output interface) through the near-end interface. In this way, to send a packet with the destination of the far end of the tunnel (e.g., 3.3.3.3), it is routed through the near-end interface (e.g., tap0) from where the application reads (see Figure 5).
4. In the case of tap interfaces, which are layer 2 interfaces, static ARP (address resolution protocol) entries are set to eliminate ARP-specific traffic through the application under test (see Figure 6).

```
$ ip netns exec test_tap ip rule
0:      from all lookup local
32000:  from all iif tap0 lookup 19
32000:  from all iif tap1 lookup 19
32766:  from all lookup main
32767:  from all lookup default
$ ip netns exec test_tap ip route show table 19
local 2.2.2.2 dev tap0 proto static scope host
local 3.3.3.3 dev tap1 proto static scope host
```

Figure 4. Custom “local” routing table for ingress traffic.

```
$ ip netns exec test_tap ip route
default via 192.168.0.1 dev test_tap -weth proto static
2.2.2.2 dev tap1 proto static scope link linkdown
3.3.3.3 dev tap0 proto static scope link linkdown
192.168.0.0/24 dev test_tap -weth proto kernel scope link src 192.168.0.2
```

Figure 5. Main routing table of the environment.

```
$ ip netns exec test_tap arp
Address   HWtype  HWaddress           Flags Mask  Iface
2.2.2.2   ether   46:03:22:a8:fc:ea   CM          tap1
3.3.3.3   ether   46:02:db:24:d0:f2   CM          tap0
```

Figure 6. Static ARP entries.

This setup allows any traffic generator with the destination set to the far end of the tunnel to pass traffic through the application. Since the local routes have been removed, the traffic is passed according to the main routing table through the near-end interface. On the far-end interface, due to the incoming traffic local rules, the traffic is passed to the host. This means that any application listening for that traffic will receive it. For instance, in Figure 7, the ping utility is used to send probes with the record route (-R) option, with the far end of the tunnel as the destination. As anticipated, the packets are routed through the near-end interface (IP address 2.2.2.2).

```
$ ip netns  exec test_tap ping  -R  3.3.3.3
PING 3.3.3.3 (3.3.3.3) 56(124) bytes of data.
64 bytes from 3.3.3.3: icmp_seq=1 ttl=64 time=59.5 ms
RR:     2.2.2.2
        3.3.3.3
        3.3.3.3
        2.2.2.2
64 bytes from 3.3.3.3: icmp_seq=2 ttl=64 time=56.0 ms   (same route)
64 bytes from 3.3.3.3: imposed=3 ttl=64 time=53.4 ms   (same route)
64 bytes from 3.3.3.3: icmp_seq=4 ttl=64 time=59.7 ms   (same route)
```

Figure 7. Probe packets sent with the ping utility to the far end of the tunnel.

As previously discussed, when using SDR (software-defined radio) boards to test in real radio channel conditions, the Tx (Transmitter) and Rx (Receiver) applications run in separate environments. The channel is replaced by SDR's IO components, such as source/sink GNU radio blocks. Each environment has only one TUN/TAP interface (as shown in Figure 1), and the environments can be hosted on the same or different machines. If the environments are hosted on different machines, they must be synchronized to align the monitoring data acquired at both ends of the transmission system in time. PTP (precision time protocol) [41] can synchronize the physical machines running the GNU radio applications. The routes are set as presented in Figures 4–6. Python and libraries such as pyroute2 [42] and python-iptables [43] are used to perform environment setup and management programmatically. These libraries enable easy integration into automated testing frameworks. The implementation of the testing environment setup can be found in [44].

3.2. Testbed Architecture

The developed testing platform for GNU radio applications has a specific architecture, illustrated in Figure 8. The figure shows a situation in which the transmission chain under test is simulated, and the two transceiver modules of the system under test are connected through a simulated channel. In this scenario, the system under test implements a full duplex transmission.

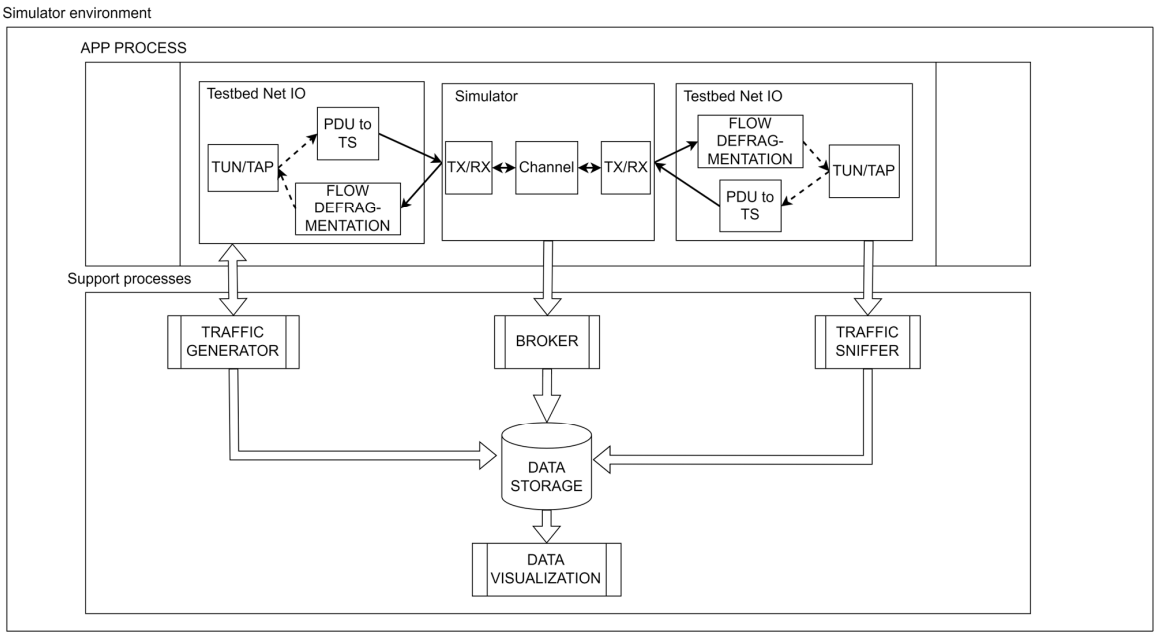

Figure 8. The architecture of the testbed for GNU radio applications evaluated through simulations.

The framework comprises two primary components: the application process and the support processes/services. The application process comprises the Net IO blocks and GNU radio simulator blocks. The support processes/services include the traffic generator, traffic sniffer, broker, database, and data visualization processes.

The testbed architecture needs to be modified to test the GNU radio application under real channel conditions using SDR boards, as shown in Figure 9. As explained in Section 3.1, when using SDR boards, the environment only consists of a single Net IO block. Traffic is either read from or written to this block, depending on whether the transmitter or receiver part of the application is being tested. The channel can be accessed through SDR IOs, i.e., source/sink blocks. The support services remain the same as when testing the GNU radio applications through simulation. If the two machines running the GNU radio application pair are different, they need to be synchronized to time-align the monitoring data.

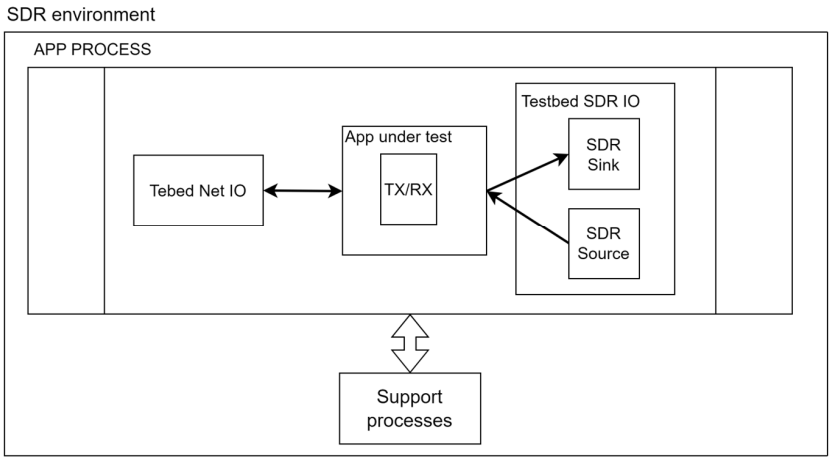

Figure 9. The architecture of the developed testing framework for GNU radio applications evaluated in real channel conditions.

3.2.1. Testbed Net IO

One of the most critical components of the testbed architecture is the Net IO blocks. These blocks connect the application being tested to the network stack of the environment,

allowing test traffic to be fed into the GNU radio application through the environment's network interfaces. The test traffic is injected into the application through the environment's TUN/TAP interfaces, as explained in Section 3.1. In the current implementation, the GNU radio built-in TUN/TAP blocks are used to read/write data from/to the interfaces. These blocks pass the protocol data units (PDUs) through the messaging-passing application program interface (API). Since the GNU radio application evaluated in this research works with tagged streams, it is essential to convert the input data flow into tagged streams. This conversion is achieved by the PDU to TS block operation.

When testing communication protocols like the PHY layer of the GNU radio transmission system, data are transferred from the PDUs to the data frames. However, this requires a size matching operation because the PHY layer frame size can be smaller than the upper layer PDU size and can change during transmission, making it impossible to directly use the MTU parameter to control the upper-layer PDU size. To deal with this issue, a PDU reconstruction/defragmentation block is added to the Net IO block on the receiving path (the flow defragmentation block in Figure 8). This block reconstructs the upper-layer PDUs before passing them further to other blocks. It is worth mentioning that fragmentation occurs when the PDUs are loaded into the PHY frames. The issue arises when the upper layer passes a PDU that does not fit into a single PHY frame. As our environment supports both TUN and TAP interfaces, the defragmentation operations are implemented for both layer 3 and layer 2 PDUs. The defragmentation mechanism discussed here assumes that the upper-layer protocol data units (PDUs) are synchronized with the physical (PHY)-layer frame. The mechanism tries to identify the beginning of the upper-layer PDU to start buffering the PHY-layer frames until the PDU length is reached or a new PDU is detected. The detection of the beginning of the PDU depends on the upper-layer protocol, which, in this case, is the TCP/IP stack-based networking.

In the case of the TUN interface setup (layer 3), the algorithm identifies the beginning of the PDU by using the IP header checksum. On the other hand, in the case of the TAP interface setup (layer 2), the algorithm uses the destination MAC address in the Ethernet header, which is known at the receiver, to identify the start of the PDU.

3.2.2. Testbed Support Services

Support services are a set of processes that are launched alongside the GNU radio application under test. These processes help generate traffic, monitor the traffic passing through the application, and collect, store, and visualize the monitoring data. One can easily use any network tool like ping or iperf3 to send traffic to the tested application. Alternatively, you can develop custom traffic generators/analyzers using Python and the Scapy library [45], a powerful packet manipulation library.

In cases where you need to analyze one-way performance or simplex transmission, traffic can be captured and analyzed in a separate process. The Scapy library provides a convenient API to send and sniff packets on L2 and L3. However, using Scapy may introduce significant delays due to the overhead it adds to each packet. For example, the L3 send API selects the interface according to the L3 header for each packet, which is not needed in the testbed since the traffic is injected in the GNU radio through a single interface. This can be overcome by selecting the interface only once and using the socket API. An essential support service in the testbed is the database service, which stores the monitoring data collected during the testing for subsequent analysis. The current implementation employs MongoDB [46] for storage, mainly due to its versatility, which better suits the purpose of the testbed. Several papers have suggested that MongoDB outperforms structured databases, such as MySQL [47]. As MongoDB is schemeless, there is no requirement for preparation for different monitoring data structures. In the testbed environment, the database service is deployed on-premises. The broker process plays a crucial role in collecting monitoring data. It parses the data and writes them to the database. Due to its strong interdependence with the messaging system, the broker will be presented in the next section.

Apart from the broker, another important support service is the data visualization service. This service helps to visualize the data collected from the GNU radio application under test and monitor the application in almost real-time. Currently, the data visualization service employs Grafana [48] to query and visualize the MongoDB database. Grafana, like the database service, is deployed on-premises.

3.3. Monitor Messaging

- When creating GNU radio applications, the monitoring messages typically come from a GNU radio block and are built within the block's work thread. This means it is important to find efficient methods for constructing these messages. Additionally, since the broker combines messages from multiple sources, improving deserialization is crucial. Refer to Figure 10 for the main components involved in transferring monitoring messages. The message generators are in charge of constructing messages and operate in the same thread as the GNU radio blocks.
- The monitor probes collect messages from multiple generators within the GNU radio application and send them to the broker.
- The broker is responsible for receiving messages from various probes and subsequently processing them.

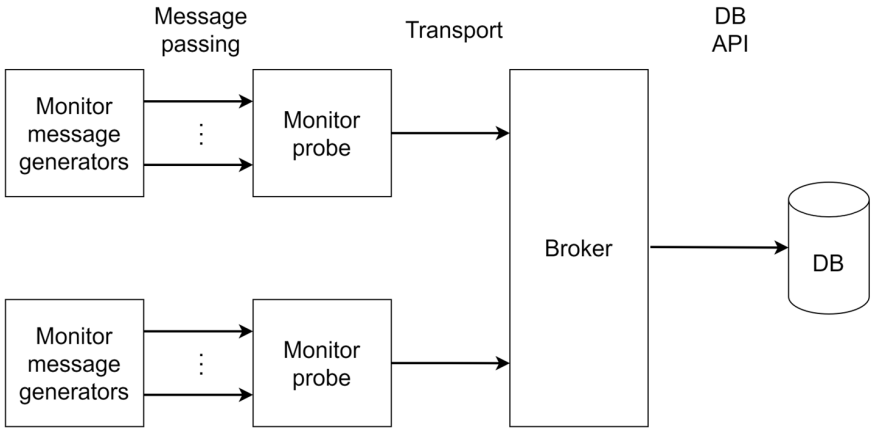

Figure 10. The architecture of the messaging system.

A message being sent from its origin to a collector (such as a broker) goes through two distinct channels:

- Message passing uses the GNU radio messaging system between individual blocks. This system uses PMTs (polymorphic types) [49] to carry different objects as messages between blocks.
- The transport channel carries the message from the GNU radio application to the message collector (i.e., the broker).

This paper suggests using the protocol buffers library [50] to reduce the size of monitoring messages and the time spent building and parsing them. This library is a language-neutral, platform-neutral extensible mechanism for serializing structured data, similar to JSON [51]. Since both the message generators and probes need to add information to the monitoring messages, the library must be available to both application-side components. While most of the monitoring data come from the GNU radio block that does the work, the probe has information that should be tracked, such as messages sent over the transport channel and the message passing queue size.

3.3.1. Monitoring Message Content

As data transmission chains follow a highly regular pattern, the monitoring data can be viewed as a time series, and therefore, each message must include a timestamp. The timestamp is added to the message when it is created. Additionally, two optional fields

are included to gather information about the probe, filled in before the message is sent over the transport channel. To ensure that the broker knows which parser to use for each incoming message, the payload type (payload ID) is included in the proto-carrier message. The content of the monitoring message is summarized in Table 1.

Table 1. Structure of the monitoring messages.

Fields	Mandatory	Filled by	Description
Timestamp	Yes	GNU radio block	Timestamp when the message was built.
Probe queue size	No	Monitor probe	GNU radio message passing API queue size.
Probe message counter	No	Monitor probe	Number of messages sent.
Payload	Yes	GNU radio block	Monitoring data.
Payload ID	Yes	GNU radio block	Indicates the payload type for the parser.

3.3.2. Monitoring Messaging Methods

To develop the implemented framework, we explored a few messaging approaches, two of which are based on the protocol buffer library [50], while a third is a baseline implementation that solely uses PMT. The PMT method sends messages as pmt::dict (see Figure 11). Both PROTO-based methods differ in the way they transmit the message between the monitoring message generator and the probe, as well as how they set the probe-specific fields in the message:

- The first method serializes the proto message immediately after building the message in the GNU radio block working thread and passes the serialized data to the probe as pmt::blob. This method is known as PROTO-BLOB.
- The second method passes the proto message object as pmt::any (i.e., boost::any), and this method is known as PROTO-ANY.

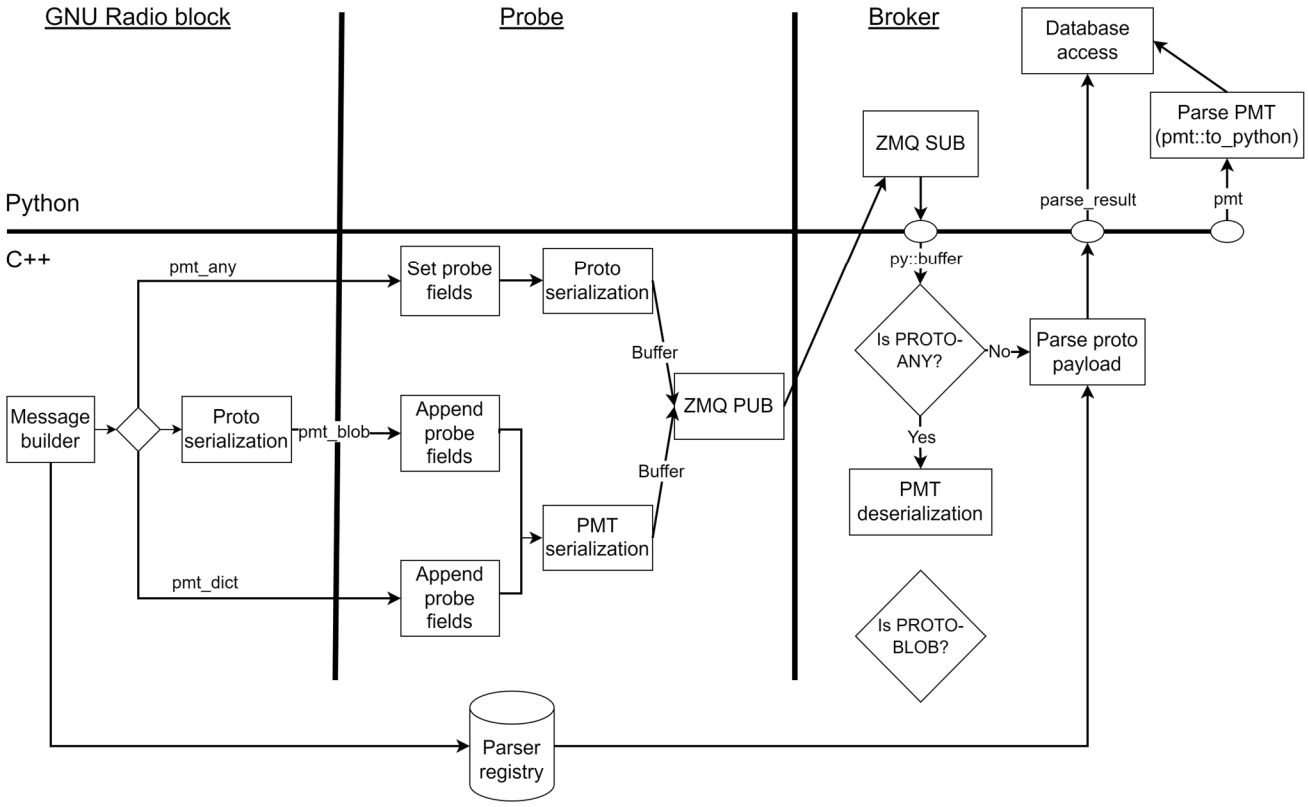

Figure 11. The generation and transfer of monitoring messages in the messaging system for the implemented messaging methods.

Figure 11 shows how messages flow through the three components of the messaging system using the three messaging methods (see Figure 10) and two programming languages.

3.3.3. The PMT-Based Messaging Method

The monitoring system's baseline implementation uses only PMT data structures built in GNU radio and is versatile. The PMT type requires no structure or schema like JSON, and the messages are self-contained. For this method, the Payload ID field is not required.

The PMT type has a significant drawback in that it adds overhead to the message size and negatively impacts the performance of serialization/deserialization of the messages. This occurs because it is necessary to send all of the message field names, and for each value, additional information is needed to indicate the value type. Additionally, the traversal of the data structure is performed recursively, which negatively affects performance for highly nested messages.

In the current implementation of the testing platform, a flat dictionary (i.e., pmt::dict), which contains both header information and payload data, is used. When building the message, the timestamp and payload are added, and the probe-related fields are easily inserted in the dictionary before serialization at the probe (see Figure 11). To make things more convenient, a syntactic sugared message builder API was implemented as a variadic function, which can take any number of (field, value) pairs as parameters, as shown in the example presented in Figure 12.

```
pmt::pmt_t msg = gr::dtl::monitor_msg(
                 std::make_pair("field_int", 10),
                 std::make_pair("field_float", 1.5),
                 std::make_pair("field_str", "string"));
```

Figure 12. Building a PMT monitoring message.

3.3.4. The PROTO-Based Messaging Method

The Protocol Buffer [50] is a tool used to work with structured data. It defines the structure (schema) in a language-neutral form that needs to be compiled to obtain the structure in the language the application is built in, like C++. This additional step is required before the application can be compiled. However, because it integrates well with CMake, using it in a GNU radio OOT (Out Of Tree) [52] build pipeline is relatively easy.

All PROTO messages have a fixed part that contains the timestamp, payload, payload ID, and probe-related information. The fixed part of the message is defined as a separate PROTO message and referred to as the main one. Each payload is defined as an independent PROTO message aggregated into the main PROTO message through a proto::any field. This allows for a single definition for the main PROTO message. Figure 13 illustrates the implementation of the PROTO messaging.

The PROTO messaging system is implemented in C++ to allow integration in the GNU radio application at a lower level. It uses a message template that is specialized for each payload message, along with the payload ID used by the message receiver to choose the parser. The protocol ID was used instead of proto::any's type_url field because it needed to be known at the compile time and used as a template parameter. The type_url field is a string. The Message Registry variadic template is specialized with all messages as a parameter pack and registers the parse methods in the parser dictionary. The messaging system is built as a shared library used by both the GNU radio application under test and the messaging broker.

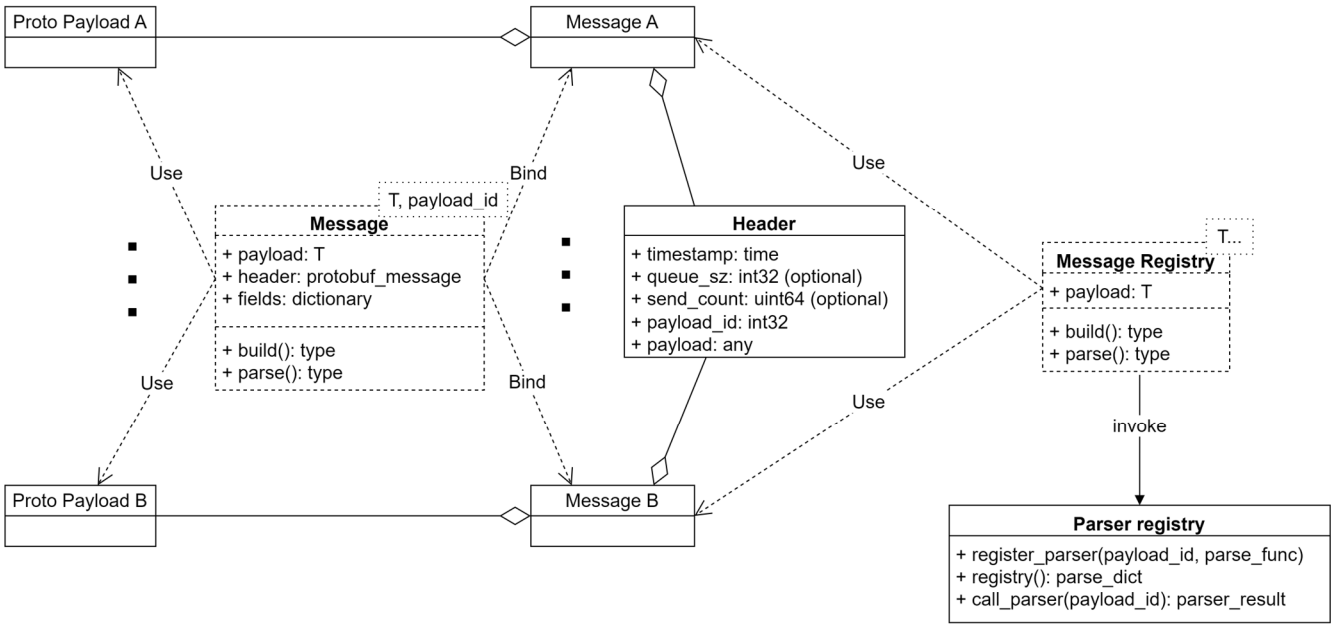

Figure 13. UML diagram of the PROTO messaging.

To maintain a consistent message-building API similar to the baseline method and to enable dynamic referencing of message fields by their names at runtime, it was necessary to create a dictionary that maps field objects to their respective names. When the message builder was constructed, the protocol buffer reflection feature was used to build this dictionary, resulting in an API similar to the PMT-only method. Figure 14 provides an example of how to make a PROTO message.

```
pmt::pmt_t msg = msg_builder.build(
                 std::make_pair("field_int", 10),
                 std::make_pair("field_float", 1.5),
                 std::make_pair("field_str", "string"));
```

Figure 14. Building a PROTO monitoring message using an API like that of the PMT method.

Since the payload information is now carried by a PROTO message, it is necessary to use the payload ID field to tell the broker which parser to use.

The PROTO-BLOB method does not use the optional probe-related fields in the main PROTO message and only sets the timestamp, payload, and payload ID when the message is generated. After that, the PROTO message is serialized and passed to the probe as a pmt::blob. The probe creates a PMT cons list containing the queue size, the sender counter, and the blob message received. After that, it serializes the new message and sends it to the transport channel. Using a cons list allows you to add fields of different types without having field names (like tuples). It adds a bit of overhead when the structure is traversed for serialization and deserialization. Still, the overhead is negligible since the number of elements in the counter list is small (three elements). With the PROTO-ANY method, the PROTO messages between GNU radio blocks are carried as pmt:any, and the method casts the main PROTO message at the probe. In this way, it is possible to use the optional probe-related fields of the main PROTO message. Once these fields are set, the PROTO message is serialized and sent over the transport channel. Because the PROTO message is not encapsulated in a PMT message, it is necessary to signal to the parser that it was not serialized with PMT. For this, it exploits the tagging mechanism that PMT uses to indicate the field type and adds a custom tag that PMT does not use. In this way, all three methods are consistent, with the first byte indicating the type of the outermost element of

the message. So, the parser only needs to look at the first byte in the message to identify which method was used. The implementation of the monitoring mechanism is available in [53].

3.3.5. The Transport of the Monitoring Messages

The platform uses the ZeroMQ [54] library to monitor messages between the probes and the broker. This messaging library is known for its high efficiency. GNU radio already has a ZMQ module to implement the most commonly used messaging patterns supported by the ZMQ library. However, since some of the monitoring and serialization logic used in the platform is part of the monitor probe, a custom ZMQ block is implemented.

To send messages, the publish/subscribe (pub/sub) [55] messaging pattern is used, with the monitor probe acting as the publisher. In most cases, the broker runs on the same machine as the GNU radio application being tested. To allow multiple monitor probes in the GNU radio process flow, the ZMQ subscriber socket on the broker side is bound, and the publish sockets in the GNU radio process flow connect to the subscriber. This way, the same port can be used on the message transport channel.

3.3.6. The Message Broker

The message broker has been developed in Python to take advantage of the improved database access support. The protocol message parsing is performed at the library level in C++ to prevent the need for compiling the PROTO files in Python. This approach ensures that the message structure remains internal to the library, which only exposes the parser. In Figure 11, you can see that the ZMQ subscriber is implemented in Python and passes the raw messages it receives to the parser through Python's buffer protocol (as pybind11 py:buffer argument) [56]. The parser returns the parse result structure that contains the message payload type (PMT or PROTO) and the payload. If the PMT payload type is used, the message is parsed in Python using the pmt::to_python implementation. A PMT message is simply a shared pointer to the PMT structure, so the amount of data copied between C++ and Python is minimal. If the PROTO payload type is used, the data are parsed by the registered parser, and a dictionary (i.e., std::unordered_map) with the result is set in the parsed result structure. To avoid copying the dictionary, pybind11 opaque types [57] are used. The parse result object is returned to Python as a unique_ptr to transfer the ownership and release it via the garbage collector once the Python object is collected. As mentioned earlier, the parser implementation is part of the monitoring library, and the broker implementation is available in [44].

4. Results and Discussions

This section discusses two main issues: the performance of the monitoring messaging methods implemented in the testing platform and the evaluation of a complex transmission system using GNU radio. As previously explained, the testing and evaluating communication protocols at the PHY layer generate many monitoring messages that must be handled in real-time. These messages' size generation and handling are essential and will be assessed in the developed platform. The aim of testing an example transmission system is to demonstrate the capabilities of the proposed platform.

4.1. Evaluation of the Monitoring Messaging Methods

4.1.1. The Message Size

Monitoring messaging using PROTO instead of PMT results in a smaller serialized message size. This is because message field names and value types do not need to be sent. We analyzed the message sizes for all three messaging method implementations described in Section 3.3 for different numbers of fields in a message. Figure 15 shows the significant difference in message size between the PMT and PROTO methods, which increases with the number of fields in the message. PROTO-ANY and PROTO-BLOB generate messages with similar sizes regardless of the number of fields in the message.

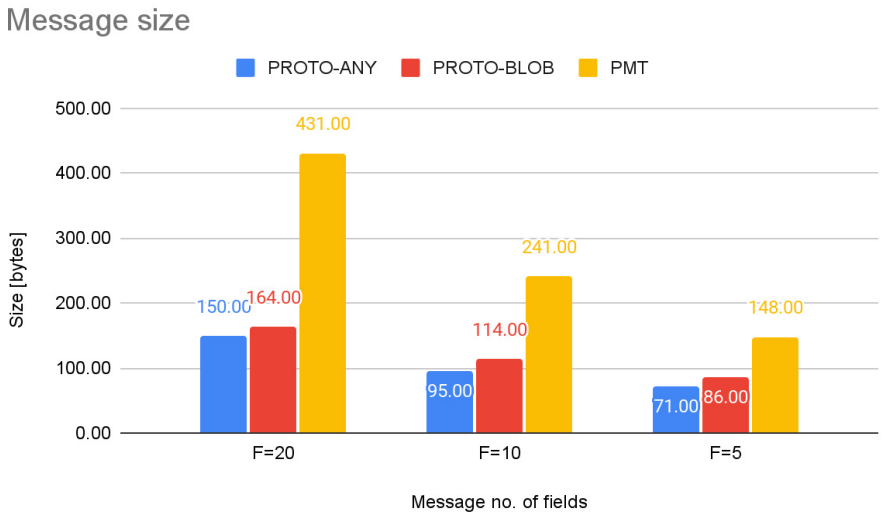

Figure 15. Monitoring message size obtained in specific conditions using the implemented messaging methods.

4.1.2. The Message Build and Parsing Times

The time required to build and parse monitoring messages was measured for 1,000,000 messages with different numbers of fields (F) to compare different messaging methods. Both operations were performed using Python bindings. The testing was conducted on a machine equipped with an AMD Ryzen 7 PRO 4750U processor and 16 GB of RAM, running Ubuntu 22.04 in WSL2. The results presented in Figure 16 show that the PROTO methods provide significantly shorter build and parsing times for messages of all sizes (number of fields) when compared to the PMT methods. The time difference between the PROTO and PMT methods decreases with the number of fields. The PROTO-ANY and PROTO-BLOB methods exhibit similar message-building and parsing times, with the PROTO-ANY method requiring less time in both cases.

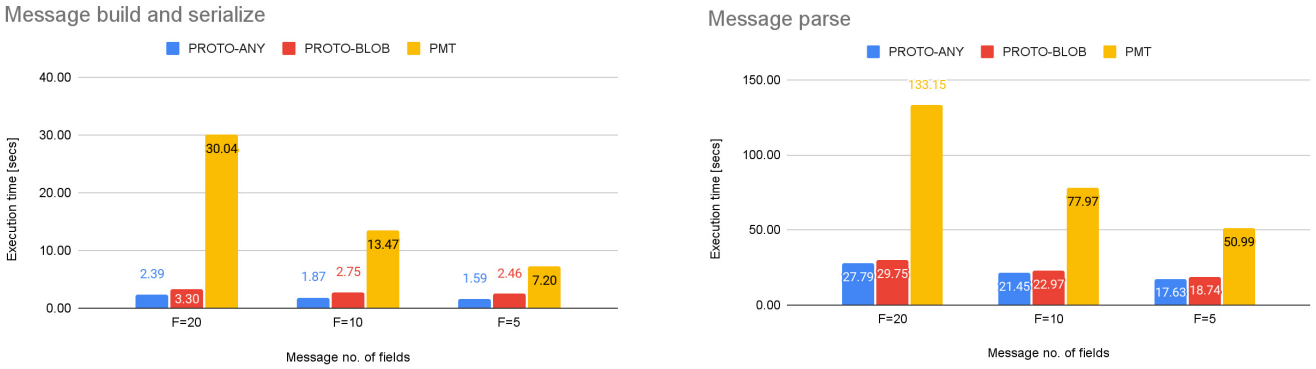

Figure 16. The time necessary to generate and process the monitoring messages in specific conditions using the implemented messaging methods.

4.1.3. End-to-End Testing of the Messaging Methods

A custom GNU radio block was created to perform end-to-end testing of monitoring messaging methods. This block generates messages of varying sizes and rates using the messaging methods described in Section 3.3. To achieve this, a small GNU radio process flow was built, which contains S message generators and a single probe (see Figure 17). The GNU radio application was executed in the networking environment presented in Section 3.1 in different scenarios, and the CPU usage was measured. The obtained results are shown in Figure 18 and indicate that PROTO-based messaging outperforms pure PMT, especially on the broker's side. This is because the broker collects data from all probes.

The gain in CPU usage at the GNU radio application is not as significant as the computing time gains presented before because the network operations are only slightly improved by the smaller size of the PROTO message. The results in Figure 18 present an average value based on more than 100 messaging-method-evaluation application runs.

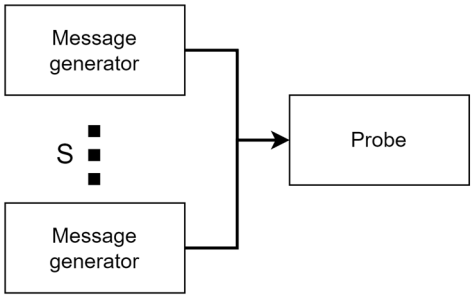

Figure 17. GNU radio process flow for evaluation of the messaging methods.

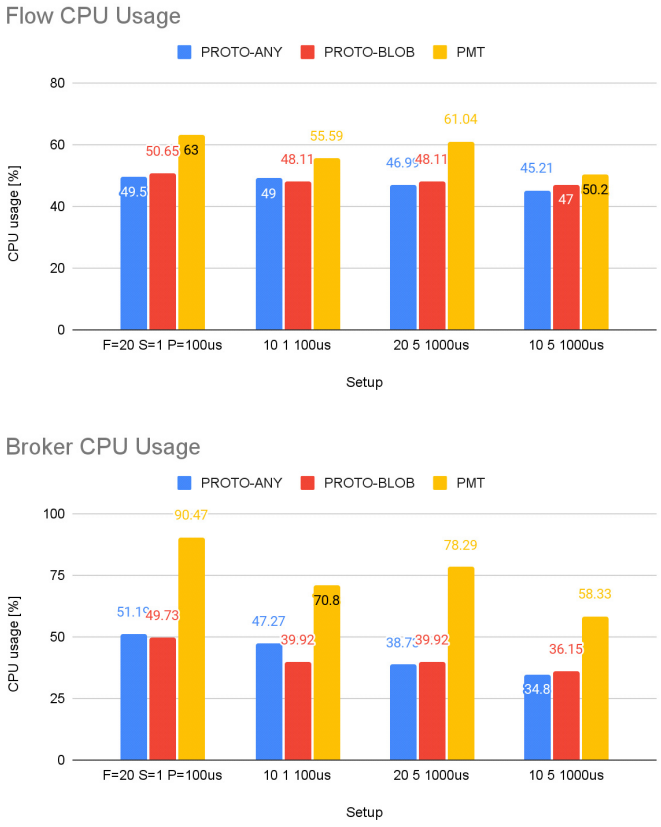

Figure 18. The CPU usage of the GNU radio process flow and of the broker in specific conditions using the implemented messaging methods.

To verify the CPU usage results presented earlier, we conducted a Wilcoxon signed-rank test [58], which considered the slight difference between the average values shown in Figure 18, illustrating the flow CPU usage. The test was performed under the alternative hypothesis that the CPU usage values obtained for PROTO-based messaging methods are lower than those obtained for PMT messaging. The p-values obtained for all the scenarios were very close to 0, indicating a high degree of confidence that the PROTO-based messaging methods always perform better than the method that uses only GNU radio PMT messages.

4.2. System under Test

To demonstrate the effectiveness of the developed testing platform in testing complex communication protocols, an OFDM transmission system with transceivers that have adaptive modulation and coding capabilities was used. This system is complex enough to exhibit the test platform's capabilities fully. Specifically, two OFDM systems were tested. The first system implements a simplex transmission with a reverse channel to convey channel state information from the receiver to the transmitter. The second system is a full-duplex transmission system where the channel state information acquired by each receiver is multiplexed with the data flows to be sent to the corresponding transmitter. Figure 19 provides a simplified schematic of the simplex OFDM transmission system. For additional information regarding the architecture of this system with adaptive modulations but without adaptive coding, please refer to [59]. The implementation of the OFDM modem under test is available in [53].

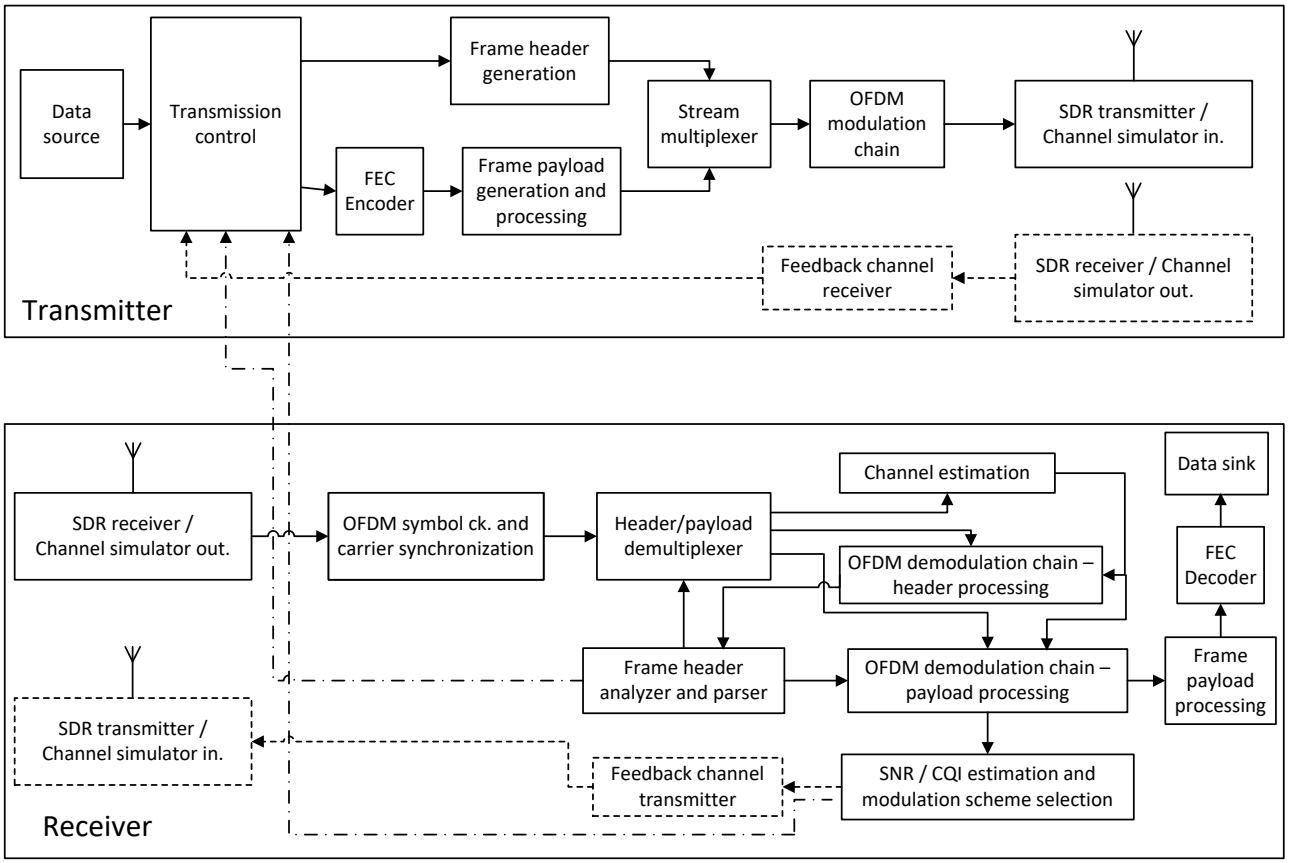

Figure 19. The simplified architecture of the OFDM transmission system tested using the developed testing platform.

The OFDM transmission system that was tested and evaluated consists of several complex signal processing blocks, including the OFDM clock and carrier synchronization block, the channel transfer function estimator, the OFDM equalizer, the forward error correction (FEC) encoder and decoder (using LDPC codes), the SNR estimation block, the transmission control block, and the framing block. To perform a thorough dynamic white box testing of these signal processing blocks, acquiring and analyzing a large amount of data is necessary. This requires a fast and efficient monitoring system that can handle real-time signal acquisition and processing of a large amount of data. The need to acquire a large amount of monitoring data in real-time is a common characteristic of any platform used for testing and evaluating complex PHY-layer algorithms [38].

Evaluation of the System under Test

The panels in Figure 20 show the changes in key parameters of an OFDM transmission system being tested over time. The parameters include the number of iterations of the LDPC decoder, the error rate of the transport block/frame, the CPU usage of the application, the estimated SNR, the bits/symbol of the modulation scheme used, and the one-way travel time of the IP packets loaded in the transport blocks. It is worth noting that this paper is not focused on detailed testing and evaluation of an OFDM transmission system with adaptive coded modulation. Instead, the system in Figure 20 is used to showcase the capabilities of the developed testing platform. The parameters in Figure 20 are evaluated frame by frame or packet by packet. However, other parameters, such as the equalization coefficients and the decision error of the QAM symbols composing the OFDM symbol, need to be evaluated at each OFDM symbol, generating a more significant amount of monitoring data. The traces representing the variation over time of the considered parameters were generated with the Grafana utility, which was also used to query the database, storing the parameter values and time stamps. The results were obtained in the simulated duplex transmission system, where both OFDM transceiver applications run on the same physical machine. In the simulated transmission system, the parameters of the wireless link can be controlled, and more relevant results can be obtained. If the SDR IO modules are integrated into the GNU radio applications, the two OFDM transceivers can be connected by a real wireless link with no other changes necessary.

Figure 20. Evolution in time of some parameters of the tested adaptive OFDM transmission system developed in GNU radio.

4.3. Threats to Validity Discussion

Various factors can threaten the reliability of test results and monitoring data. It is essential to identify these threats to take corrective actions. In the case of the communication system being tested on GNU radio, the main limitation is the minimum sampling frequency, which is determined by the bandwidth of the communication link. If the sampling frequency is too high, signal processing could be impossible on resource-limited machines. It could also result in data loss if too many data are acquired or the monitoring frequency is too high. To prevent this, the testbed monitors the CPU usage of the system under test and that of the broker to alert in case of processing resource starvation. The monitoring data can be corrupted if the transmission nodes are not synchronized or the synchronization needs to be more precise. A synchronization mechanism that is precise enough for the monitoring data frequency needs to be used to avoid this.

Ensuring that the IP packets and Ethernet frames sent and received by the application are adequately fragmented and defragmented is essential. If these operations fail, it can cause issues with the data flow at the output of the system being tested, resulting in incorrect statistics. To avoid this problem, the IP and Ethernet headers should be set according to the defragmentation mechanism implemented in the testbed.

5. Conclusions

In this study, we aimed to create a software testbed to assess communication protocols for the PHY and MAC layers developed through GNU radio. This testbed should be easy to use, capable of interacting with any traffic generator, and should enable the collection and analysis of PHY-layer monitoring data. To achieve this, the paper outlines how to set up a network environment that can be used for end-to-end testing of both simulation and real channel applications. These network environments are isolated, making them easier to manage and allowing multiple environments on the same machine—which is particularly important when multiple tests are executed simultaneously on powerful servers. The setup and management are implemented in Python to facilitate integration in testing automation, using the Linux kernel's virtual network devices (tun/tap) to feed test traffic into the applications (i.e., transmission chain for SDR) to make it compatible with any network traffic generator. In this paper, we have suggested and examined various methods for obtaining monitoring data from the physical layer. One approach employs only the polymorphic types (PMTs) built into GNU radio, while the other two use the protocol buffer library to improve the efficiency of message generation, serialization, and parsing processes. The PMT method is highly versatile and user-friendly, as it is integrated into the GNU radio runtime and does not necessitate schema definitions. Testing communication protocols generates significant monitoring data that need to be acquired in real-time. It is important to use suitable methods to ensure efficiency and faster messaging. We analyzed the performance of protocol-buffer-based methods in terms of the computation time of the main components, CPU usage in end-to-end tests, and message size. The results showed that the protocol-buffer-based methods outperformed the PMT method in all scenarios considered. In this paper, we have explored the complete evaluation system, covering PHY-layer monitoring, data storage, and visualization. We have demonstrated how to use this system for white box testing of wireless transmission protocols developed with GNU radio. The testbed's capabilities were demonstrated using a complex OFDM duplex transmission system with adaptive coded modulations. Although we have not explicitly addressed the automation of the testing process, it is clear the proposed framework has the potential to facilitate the integration of such functionality with relative ease. Custom traffic generators can be constructed to align with the test suite while implementing testbed management in Python, which allows it to be easily exposed to automation testing frameworks.

Author Contributions: Conceptualization, M.P.S. and Z.A.P.; methodology, M.P.S.; software, M.P.S.; validation, Z.A.P. and M.P.S.; formal analysis, Z.A.P.; investigation, M.P.S.; resources, Z.A.P.; data curation, Z.A.P.; writing—original draft preparation, M.P.S.; writing—review and editing, Z.A.P.; visualization, M.P.S.; supervision, Z.A.P.; project administration, M.P.S.; funding acquisition, Z.A.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

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

References

1. Okenzie, F.; Odun-Ayo, I.; Bogle, S. A Critical Analysis of Software Testing Tools. *J. Phys. Conf. Ser.* **2019**, *1378*, 042030. [CrossRef]
2. Bansal, A. A Comparative Study of Software Testing Techniques. *Int. J. Comput. Sci. Mob. Comput.* **2014**, *3*, 579–584.

3. Khan, M.E.; Khan, F. Importance of Software Testing in Software Development Life Cycle. *Int. J. Comput. Sci. Issues* **2014**, *11*, 120–123.
4. Hanna, M.; El-Haggar, N.; Sami, M.A. Review of Scripting Techniques Used in Automated Software Testing. *Int. J. Adv. Comput. Sci. Appl.* **2014**, *5*, 194–202. [CrossRef]
5. Kannan, S.; Pushparaj, T. A Study on Variations of Bottlenecks in Software Testing. *Int. J. Comput. Sci. Eng.* **2014**, *2*, 8–14.
6. Chauhan, R.K.; Singh, I. Latest Research and Development on Software Testing Techniques and Tools. *Int. J. Curr. Eng. Technol.* **2014**, *4*, 2368–2372.
7. Zhang, H. Research on Software Development and Test Environment Automation based on Android Platform. In Proceedings of the 3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019), Dalian, China, 29–30 March 2019; pp. 749–753.
8. Orso, A.; Rothermel, G. Software testing: A research travelogue (2000-2014). In Proceedings of the Future of Software Engineering (FOSE 2014), Hyderabad, India, 31 May–7 June 2014; pp. 117–132.
9. Goyat, J.; Dhingra, S.; Goyal, V.; Malik, V. Software Testing Fundamentals: A Study. *Int. J. Latest Trends Eng. Technol.* **2014**, *3*, 386–390.
10. Weyuker, E.J.; Vokolos, F.I. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Softw. Eng.* **2000**, *26*, 1147–1156. [CrossRef]
11. Uyar, M.U.; Fecko, M.A.; Duale, A.Y.; Amer, P.D.; Sethi, A.S. Experience in Developing and Testing Network Protocol Software Using FDTs. *Inf. Softw. Technol.* **2003**, *45*, 815–835. [CrossRef]
12. Jamieson, C.; Melvin, S.; Ilow, J. Rapid Prototyping Hardware Platforms for the Development and Testing of OFDM Based Communication Systems. In Proceedings of the 3rd Annual Communication Networks and Services Research Conference (CNSR'05), Halifax, NS, Canada, 16–18 May 2005; pp. 57–62.
13. Popescu, O.; Abraham, S.; El-Tawab, S. A Mobile Platform Using Software Defined Radios for Wireless Communication Systems Experimentation. In Proceedings of the 2017 ASEE Annual Conference & Exposition, Columbus, OH, USA, 24–28 June 2017.
14. Serkin, F.B.; Vazhenin, N.A. USRP Platform for Communication Systems Research. In Proceedings of the 15th International Conference on Transparent Optical Networks (ICTON 2013), Cartagena, Spain, 23–27 June 2013; pp. 1–4.
15. GNURadio. The Free & Open Software Radio Ecosystem. Available online: https://www.gnuradio.org (accessed on 20 November 2023).
16. Kaur, M.; Singh, R. A Review of Software Testing Techniques. *Int. J. Electron. Electr. Eng.* **2014**, *7*, 463–474.
17. Roggio, R.F.; Gordon, J.S.; Comer, J.R.; Khan, F. Taxonomy of Common Software Testing Terminology: Framework for Key Software Engineering Testing Concepts. *J. Inf. Syst. Appl. Res.* **2014**, *7*, 4–12.
18. Just, R.; Jalali, D.; Inozemtseva, L.; Ernst, M.D.; Holmes, R.; Fraser, G. Are Mutants a Valid Substitute for Real Faults in Software Testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), Hong Kong, China, 16–21 November 2014; pp. 654–665.
19. Li, W. Design and Implementation of Software Testing Platform for SOA-Based System. In Proceedings of the 2021 IEEE 6th International Conference on Computer and Communication Systems (ICCCS), Chengdu, China, 23–26 April 2021; pp. 1094–1098.
20. Rosch, S.; Tikhonov, D.; Schutz, D.; Vogel-Heuser, B. Model-Based Testing of PLC Software: Test of Plants' Reliability by Using Fault Injection on Component Level. *IFAC Proc. Vol.* **2014**, *47*, 3509–3515. [CrossRef]
21. Kaur, K.; Singh, J.; Ghumman, N.S. Mininet as Software Defined Networking Testing Platform. In Proceedings of the International Conference on Communication, Computing & Systems (ICCCS 2014), Ferozepur, India, 8–9 August 2014; pp. 139–142.
22. Bertolini, A.; De Angelis, G.; Gallego, M.; Garcia, B.; Gortazar, F.; Lonetti, F.; Marchetti, E. A Systematic Review on Cloud Testing. *ACM Comput. Surv.* **2019**, *52*, 1–42. [CrossRef]
23. Xi, W.; Liu, W.; Bai, T.; Ye, W.-P.; Shi, J. An Automation Test Strategy Based on Real Platform for Digital Control System Software in Nuclear Power Plant. *Energy Rep.* **2020**, *6*, 580–587. [CrossRef]
24. Garousi, V.; Felderer, M.; Karapicak, C.M.; Yilmaz, U. Testing Embedded Software: A Survey of the Literature. *Inf. Softw. Technol.* **2018**, *104*, 14–45. [CrossRef]
25. Masood, S.; Khan, S.A.; Hassan, A.; Fatima, U. A Novel Framework for Testing High-Speed Serial Interfaces in Multiprocessor Based Real-Time Embedded System. *Appl. Sci.* **2021**, *11*, 7465. [CrossRef]
26. Sarikaya, B.; Bochmann, G.V.; Cerny, E. A Test Design Methodology for Protocol Testing. *IEEE Trans. Softw. Eng.* **1987**, *SE-13*, 518–531. [CrossRef]
27. Wang, B.; Hutchison, D. Protocol Testing Techniques. *Comput. Commun.* **1987**, *10*, 79–87. [CrossRef]
28. Dssouli, R.; Saleh, K.; Aboulhamid, E.; Bediaga, A.; En-Nouaary, A.; Bourhfir, C. Test Development for Communication Protocols: Towards Automation. *Comput. Netw.* **1999**, *31*, 1835–1872. [CrossRef]
29. Lai, R. A Survey of Communication Protocol Testing. *J. Syst. Softw.* **2002**, *62*, 21–46. [CrossRef]
30. Dorofeeva, R.; El-Fakih, K.; Maag, S.; Cavalli, A.R.; Yevtushenko, N. FSM-Based Conformance Testing Methods: A Survey Annotated with Experimental Evaluation. *Inf. Softw. Technol.* **2010**, *52*, 1286–1297. [CrossRef]
31. Dssouli, R.; Khoumsi, A.; Elqortobi, M.; Bentahar, J. Chapter Three-Testing the Control-Flow, Data-Flow, and Time Aspects of Communication Systems: A Survey. *Adv. Comput.* **2017**, *107*, 95–155.
32. Bai, Y.; Tang, P.; Zhang, J.; Zhang, J. Test Method of Communication Protocol of Standard Group Components of Electric Vehicle Charging Equipment. *J. Phys. Conf. Ser.* **2021**, *2066*, 012032. [CrossRef]

33. Lawrenz, W. Communication Protocol Conformance Testing—Example LIN. In Proceedings of the 2006 IEEE International Conference on Vehicular Electronics and Safety, Shanghai, China, 13–15 December 2006; pp. 155–162.
34. Tapia, E.; Sastoque-Pinilla, L.; Lopez-Novoa, U.; Bediaga, I.; López de Lacalle, N. Assessing Industrial Communication Protocols to Bridge the Gap between Machine Tools and Software Monitoring. *Sensors* **2023**, *23*, 5694. [CrossRef]
35. Rettore, P.H.L.; Loevenich, J.; Lopes, R.R.F. TNT: A Tactical Network Test Platform to Evaluate Military Systems over Ever-Changing Scenarios. *IEEE Access* **2022**, *10*, 100939–100954. [CrossRef]
36. Zhu, L.; Zhao, Y.; Gao, L. Software Testing Method Based Mobile Communication Equipment of Maritime Satellite. *IOP Conf. Ser. Earth Environ. Sci.* **2019**, *234*, 012059. [CrossRef]
37. Bertizzolo, L.; Bonati, L.; Demirors, E.; Al-shawabka, A.; D'Oro, A.; Restuccia, F.; Melodia, T. Arena: A 64-Antenna SDR-Based Ceiling Grid Testing Platform for sub-6 GHz 5G-and-Beyond Radio Spectrum Research. *Comput. Netw.* **2020**, *181*, 107436. [CrossRef]
38. Li, Y.; Zhu, X.; Hu, L. General Multiple Antenna Evaluation Platform. In Proceedings of the 2005 2nd Asia Pacific Conference on Mobile Technology, Applications and Systems, Guangzhou, China, 15–17 November 2005; pp. 57–62.
39. Tun/Tap Interface Tutorial. Available online: https://backreference.org/2010/03/26/tuntap-interface-tutorial/index.html (accessed on 22 November 2023).
40. Introduction to Linux Interfaces for Virtual Networking. Available online: https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking# (accessed on 24 November 2023).
41. Introduction to Precision Time Protocol (PTP). Available online: https://networklessons.com/cisco/ccnp-encor-350-401/introduction-to-precision-time-protocol-ptp (accessed on 10 January 2024).
42. Pyroute2 Netlink Library. Available online: https://docs.pyroute2.org/ (accessed on 30 November 2023).
43. Python-Iptables. Available online: https://python-iptables.readthedocs.io/en/latest/intro.html (accessed on 30 November 2023).
44. Testbed for GNU Radio Applications. Available online: https://github.com/mihaipstef/dtl-testbed (accessed on 12 December 2023).
45. Scapy. Available online: https://scapy.net/ (accessed on 24 November 2023).
46. MongoDB Documentation. Available online: https://www.mongodb.com/docs/ (accessed on 24 November 2023).
47. Eyada, M.M.; Saber, W.; El Genigy, M.M.; Amer, F. Performance Evaluation of IoT Data Management Using MongoDB Versus MySQL Databases in Different Cloud Environments. *IEEE Access* **2020**, *8*, 110656–110668. [CrossRef]
48. Grafana Documentation. Available online: https://grafana.com/docs/grafana/latest/ (accessed on 24 November 2023).
49. Polymorphic Types (PMTs). Available online: https://wiki.gnuradio.org/index.php/Polymorphic_Types_(PMTs) (accessed on 27 November 2023).
50. Protocol Buffers Documentation. Available online: https://protobuf.dev/overview/ (accessed on 27 November 2023).
51. Introducing JSON. Available online: https://www.json.org/json-en.html (accessed on 27 November 2023).
52. OutOfTreeModules. Available online: https://wiki.gnuradio.org/index.php/OutOfTreeModules (accessed on 28 November 2023).
53. Adaptive OFDM Modem and Monitoring Library in GNU Radio. Available online: https://github.com/mihaipstef/gr-dtl (accessed on 12 December 2023).
54. ZeroMQ. An Open-Source Universal Messaging Library. Available online: https://zeromq.org/ (accessed on 28 November 2023).
55. What Is Pub/Sub? Available online: https://cloud.google.com/pubsub/docs/overview (accessed on 28 November 2023).
56. Buffer Protocol. Available online: https://docs.python.org/3/c-api/buffer.html (accessed on 29 November 2023).
57. STL Containers. Available online: https://pybind11.readthedocs.io/en/stable/advanced/cast/stl.html (accessed on 29 November 2023).
58. Wilcoxon, F. Individual Comparisons by Ranking Methods. *Biom. Bull.* **1945**, *1*, 80–83. [CrossRef]
59. Polgar, Z.A.; Stef, M. OFDM Transceiver with Adaptive Modulation Implemented in GNU Radio. In Proceedings of the 2023 46th International Conference on Telecommunications and Signal Processing (TSP), Prague, Czech Republic, 12–14 July 2023; pp. 37–42.