On the Usage of Grammar-based Fuzzing to Evaluate the Implementation of AMQP Brokers

Ivan Gabriel Ferreira Dias, Daniel Macêdo Batista

[1]Department of Computer Science – Institute of Mathematics and Statistics
University of São Paulo (USP) – São Paulo, SP – Brazil

ivangabrielfdias@usp.br, batista@ime.usp.br

Abstract. *Automated testing strategies, like fuzzing, can be used to ensure the security and reliability of software systems, and those that implement communication protocols have specific needs. One class of protocols that deserves to be highlighted, both for its expansion in use and for its complexity, is the publish-subscribe (pub-sub) class. Among several pub-sub protocols, AMQP does not receive much attention from the literature regarding fuzzing testing. This paper fills this gap by proposing the usage of grammar-based fuzzing to test AMQP brokers. The proposal is integrated into a new free and open-source fuzzer called AMQPGRAM. Experiments with AMQPGRAM attest to its capacity to cover 100% of the AMQP messages related to the establishment of connections.*

1. Introduction

Although client-server protocols are very used today, as evidenced by the fact that more than 87% of web pages are delivered using the HTTPS client-server protocol [W3Techs 2025], publish-subscribe protocols (pub-sub) have also been widely used, for example in Internet of Things (IoT) environments. These protocols follow a data transfer model suitable for scenarios in which several clients must receive a certain piece of information provided by another client without a direct connection to it. Such a model is extremely useful in scenarios such as smart cities [Gemirter et al. 2021] and the Internet of Robotic Things (IoRT) [Yoshino et al. 2021]. An example in smart cities would be the communication between a light sensor and hundreds of streetlights. The streetlights would be the subscribers and the sensor would be responsible for publishing information about the natural light so that streetlights can be turned on or off automatically. Since a sensor in this scenario is usually a device with restricted resources, having it communicate directly with each post, as in the case of a client-server protocol, would be prohibitive. With a pub-sub protocol, the sensor sends the information to only one device with greater capacity, called a broker, which is responsible for forwarding the information to the posts.

Among the various existing pub-sub protocols, AMQP (Advanced Message Queuing Protocol) stands out due to its low latency in certain scenarios [Yoshino et al. 2021] and because of its use in RabbitMQ [Broadcom 2025], a versatile broker used in scenarios where services need to be decoupled, when it is necessary to use Remote Procedure Calls (RPCs), when there is a need to stream videos, in addition to when there is communication with IoT devices. In particular, AMQP version 0-9-1 was introduced in 2008 to unify business to business communications [OASIS 2008]. Today, 17 years later, this version of AMQP is still fairly used, with more than 120 thousand public hosts implementing the protocol [Shodan 2025]. The interest in AMQP continues high, with recent approaches in

the literature proposing its integration with QUIC [Iqbal et al. 2023] and the improvement of its security mechanisms [Adiwal et al. 2024].

Modern software development considers that testing the written code is essential before the software is put into production. In the case of communication protocols such as AMQP, in addition to local tests that verify the implementation of components (functions, methods, classes, etc.), it is important that the entire state machine defined in the protocol specification be tested, preferably with the exchange of messages between the parts necessary for its operation. This can be done either manually, with developers analyzing the code, or in an automated way, using software written specifically to test the protocol implementation. In the case of a pub-sub protocol, testing of the broker-side implementation can be done using a client developed for the purpose of testing how the broker behaves with different message sequences, including unexpected sequences and malformed packets. This type of special client is called "fuzzer" [Liang et al. 2018].

Being a popular protocol, with some implementations carrying more than 10 years of market presence, it is expected that AMQP implementations are subjected to specific communication software testing techniques. However, even with AMQP's considerable presence, there is a lack of literature regarding tests of this protocol. To the best of our knowledge, there are no fuzzers publicly available to test the protocol, those being more abundant in works about another pub-sub protocol, MQTT. For instance, in [Kwon et al. 2021] the use of fuzzing tests to search for vulnerabilities in RabbitMQ has shown positive results. However, the authors focus on the RabbitMQ's implementation of MQTT. More recently, in [Luo et al. 2024], the performance of a parallel fuzzing framework to test implementations of different pub-sub protocols was evaluated. Despite AMQP appearing as one of the tested protocols, with Qpid [Apache 2015] being the broker under test, the framework design considers the MQTT protocol. Unsurprisingly, the authors could not find problems in the implementation for AMQP, only for MQTT.

With that knowledge, this paper presents a strategy to run fuzzing tests in AMQP 0-9-1 implementations and describes AMQPGRAM, a grammar-based fuzzer for AMQP 0-9-1 (Grammar-based fuzzers use the specification of the protocol to generate messages to be sent to the broker [Rodriguez 2023]). AMQPGRAM is used to test two implementations of brokers for AMQP 0-9-1: RabbitMQ 2.5.0 and RabbitMQ 4.1[1]. Experiments show the capacity of the fuzzer to traverse the state diagram of the protocol by exchanging expected and unexpected packets. As an additional contribution, AMQPGRAM is publicly available as free and open-source software[2].

The remainder of this paper is organized as follows: Section 2 describes the fuzzer design. Section 3 explains the planned experiments. The results obtained in these experiments are presented in Section 4, and Section 5 concludes the paper and lists some ideas for future works.

2. Fuzzer Design

A fuzzer can be implemented using diverse techniques [Rodriguez 2023]. For instance, mutation-based fuzzers introduce small changes to existing test cases, whereas

[1]The latest commit at the time the testbench was prepared, `https://github.com/rabbitmq/rabbitmq-server/commit/21c242288aa3873d2ac7586b22bd90463e099616`

[2]`https://github.com/ivangfdias/amqpgram`

generation-based fuzzers generate test cases from scratch. These test cases created by generation-based fuzzers can be crafted by means of a grammar, which in turn is based on the specification of the protocol.

A grammar-based approach for generating the messages of an AMQP fuzzer presents itself as a good fit for our purpose, as it would leverage the specification of the protocol itself. Such approach, applied to the MQTT protocol, presented better results in regards to statement coverage and resource usage when compared to others [Araujo Rodriguez and Batista 2021]. Moreover, it highlighted the shortcomings of the other approaches, most notably difficulty to reach deep protocol states or correctly test core capabilities such as publishing. With AMQP being much more complex in respect to protocol states than MQTT, such shortcomings make opting for a grammar-based approach even more appealing.

2.1. A Grammar for AMQP 0-9-1

The protocol specification document and the XML-derived specification, available at [OASIS 2008], both provide a grammar for the *structure* of the protocol. The first one details how a frame, the term used in the specification to refer to a packet, is structured and what parts compose each of the many fields of the different frames. The second provides the "happy flow", expected message sequence, for each class, which is a protocol method. In this paper we focus on the Connection class, responsible for a peer to establish and keep a network connection with another.

The provided grammar for the protocol is not compliant with any of the Augmented Backus-Naür Form (ABNF) definitions ([Crocker and Overell 2008] [Kyzivat 2014]). While easily fixable, it also does not provide specific definitions for each of the many classes and methods specified by the protocol, requiring writing a new document to include that information.

As our scope in this work is limited to the Connection class, we limited our efforts to creating a document for this class, excluding the Connection.Secure and Connection.Secure-OK methods. Those methods were excluded as they are not used when using the "plain" authentication method and do not test the AMQP protocol, only the subjacent authentication method.

The grammar was derived from the available documentation and used to generate packets to test against a modern version of RabbitMQ, the *de facto* standard for the protocol. When the grammar proved not adequate, adjustments were made to comply with what RabbitMQ implemented.

One problem with using the grammar to fuel the packet generation is that ABNF can never represent the protocol logic correctly, as it lacks mechanisms for specifying relationships between parts of the same rule definition. For instance, this is the given definition of a short string field in a table:

$$short - string = OCTET \quad * string\text{-}char;$$

i.e., a $short\text{-}string$ is an $OCTET$ followed by any number of $string\text{-}char$, including zero. However, it does not capture the fact that the first $OCTET$ is supposed to

be the length of the following sequence of $string\text{-}char$, and this relation is impossible to capture with ABNF representation. This problem is solved by replacing these grammar definitions for auxiliary functions, described in the next section.

2.2. Fuzzer Structure

The fuzzer itself is divided in two major parts: the *generator*, and the *client*. The generator is responsible for encoding the grammar into symbol tables, managing such symbol tables, and generating valid data from them. Those symbol tables accept only strings as keys, and may accept strings, string arrays, byte arrays or functions as values. The client is responsible for the networking, protocol state, message receiving and sending, and creating the auxiliary functions used by the generator. Given an environment – linked list of symbol tables – E and a string S, the generation algorithm G is:

1. Search for the value of S in each element of E, successively
2. If S is not present in any of E, return an error.
3. Else, being V the first value found for S in E, do:
    - If V is a terminal or sequence of terminals, decode it and return both the decoded value and its length;
    - If V is a range of terminals, randomly choose one between the possible values and return both the decoded value and its length;
    - If V is a byte array, return V and its length;
    - If V is a function, return the results of $V(E)$;
    - If V is a string array, randomly select a v in V, then return the results of $G(v, E)$;
    - If V is a string, return the results of $G(V, E)$.

Using this algorithm, we can easily fix the previously presented $short\text{-}string$ problem. First, create a function F that first generates an $OCTET$, then generates a number of $string\text{-}char$ based in that octet, and then returns all generated values in sequence and the length of the sequence. Then, create a new symbol table T, associate "short-string" to F in it, and make T the head of the environment (which contain the rest of the grammars). Now, whenever a rule refers to "short-string", it will reach F and generate a syntactically valid and logically correct output.

This is useful not only to correct those abnormalities, but also to pin field values when constructing a valid packet. The call to create the packet sent would have received an environment as presented in Figure 1, so the process could build each of the fields separately, associate them with their respective keys in the head of the environment, then prompt the generator with a single rule to join all into a valid packet.

2.3. Fuzzer Configuration

In order to easily run diverse test scenarios, the fuzzer can be configured with the following attributes, which are mostly self-explanatory: Target IP Address; Target TCP Port; Stop criteria (either elapsed seconds or exchanged packets); "Rule Chaos"; "Packet Chaos"; and Verbosity levels.

Both Rule and Packet Chaos are an integer between 0 and 4, which represent divergence from protocol logic. Rule Chaos at 0 means that every rule which has a function

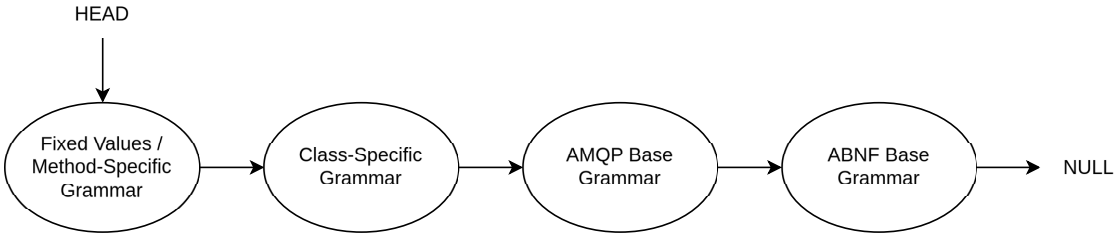

Figure 1. Environment used to generate a valid AMQP packet

Table 1. Machine specification for each software used

Software	Memory (GiB)	CPU	Available Cores
AMQPGRAM	1	Intel Atom N450 @1.66GHz	2
RabbitMQ 2.5	8	Intel Core i5-8250U @1.60GHz	4
RabbitMQ 4.1	20	Intel Core i5-8250U @1.60GHz	8

associated with it, such as "short-string" in our previous examples, would use that function to resolve that rule, while a value of 4 means that the function would be ignored in favour of the raw ABNF definition. Packet Chaos at 0 means that a frame type can only be sent if the fuzzer is in the respective state for it, and at 4 means that frame types are sent at random. Intermediary values for either attribute decide randomly whether or not follow the protocol logic in steps of 25% chance of not following it. This configuration allows 25 distinct test scenarios, with varying degrees of adherence to the protocol.

3. Experiments

The fuzzing approach presented in the previous section was implemented in a fuzzer called AMQPGRAM, using the C programming language. This fuzzer was evaluated in experiments based in five campaigns of each of the test scenarios, each running for 30 minutes in a different machine than the broker. Table 1 shows the hardware specifications for each machine. The machines were connected via a direct 10 Mbps Ethernet cable without switches. Such efforts were made in order to prevent hidden optimizations (such as use of Unix sockets instead of network sockets), minimize physical media effects (as might happen in Wi-Fi), and ensure a fair sampling of the scenario.

Each campaign tries to test, as many times as possible, the full cycle of an AMQP Connection. This means, when sending packets in order, sending the protocol header, then receiving a Connection.Start, then sending a Connection.Start-OK, and so on, until one of the peers sends a Connection.Close-OK, the socket is closed, or the broker sends the protocol header back. When that happens, the cycle is closed and another one begins.

In order to test the behaviour of implementations across time, we performed the test against two deployments, one using Ubuntu 14 running RabbitMQ 2.5.0, and other using Arch Linux, updated as per $2025/01/20$, running RabbitMQ 4.1. The machines also ran Wireshark for packet capture and a bash script for performance data collection.

As the test scenarios depend on two attributes – Rule Chaos and Packet Chaos – we represent both as a single measure, "Overall Chaos", using the following formula:

$$Overall\ Chaos = (5 * Rule\ Chaos) + Packet\ Chaos.$$

AMQPGRAM also provides, indirectly, a way to measure protocol coverage. Every frame type sent and received is logged, and such logs can be filtered and compared to the set of possible messages for our tests. If a same message could be sent from either the client or the broker, it is treated as two separate messages. Our message set is as follows:

- **Client:** Protocol Header, Connection.Start-OK, Connection.Tune-OK, Connection.Open, Connection.Close, Connection.Close-OK
- **Broker:** Protocol Header, Connection.Start, Connection.Tune, Connection.Open-OK, Connection.Close, Connection.Close-OK

4. Results

The first, most important metric captured in the experiments is the amount of packets exchanged between broker and fuzzer. In Figure 2 we see a prominent peak for Overall Chaos values of 0 and 1, meaning packets are well-formed and sent, at most, 25% of the time out of order. As more packets are sent out of order, the packets exchanged drop rapidly, as the protocol specifies that "[f]ollowing successful protocol header negotiation [...] and prior to receiving Open or Open-Ok, a peer that detects an error MUST close the socket without sending any further data." [OASIS 2008] As the fuzzer implements a timeout period to detect closed sockets, this creates a small delay between test cycles, which compounds across the campaign.

As Rule Chaos increases, even when packets are perfectly ordered, the broker silently closing the socket has an even greater impact. Unordered packets, in fact, end up *improving* exchange rates, evident by the sawtooth aspect, as the broker responds to an unexpected first packet with the protocol header. This enables skipping over the timeout detection and restarting the cycles faster.

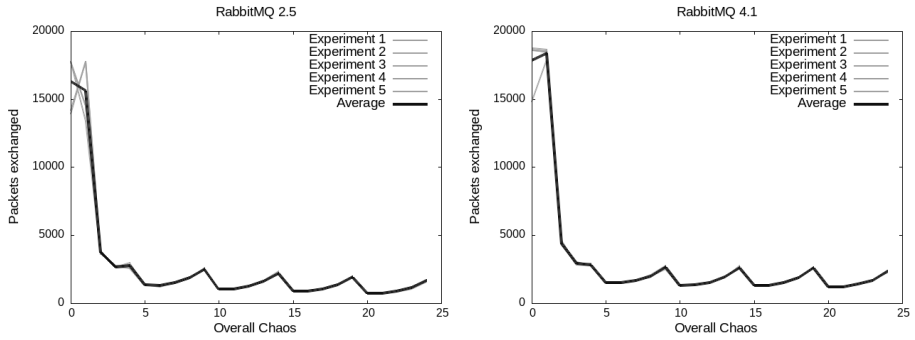

Figure 2. Amount of packets exchanged as a function of the overall chaos (One line for each repetition plus the average)

Our data also supports finding implementation issues with the fuzzer. Figure 3 indicates that average memory usage by the fuzzer is linked to packets exchanged. As each test cycle is supposedly self-contained, this hints towards memory leaks between test cycles, while not exceedingly harmful – 200KiB over 30 minutes.

The protocol coverage reinforced our assumptions about variability in testing. While the "happy flow" corresponds to 82% of the implemented messages, introducing out of order packets led us to covering the whole scope. When introducing "Rule Chaos", the coverage reduced so much that, in average, the happy flow was not covered, and for

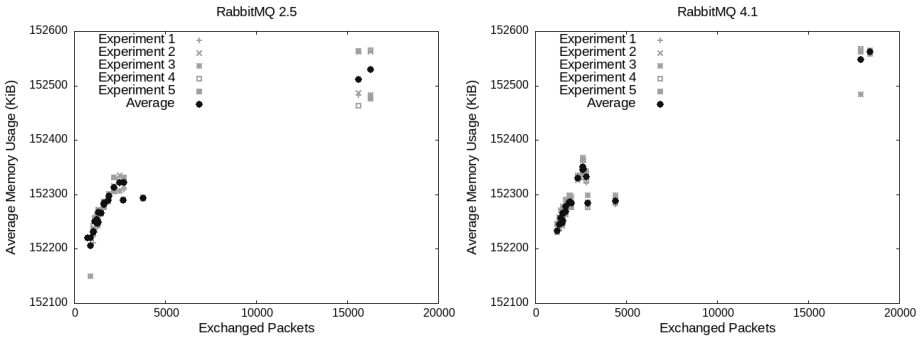

Figure 3. Memory usage of the fuzzer as a function of the amount of packets exchanged (Points for each repetition plus the interpolation of the average)

higher values of Rule Chaos, lower Packet Chaos led to abysmal performance, covering only three out of twelve possible messages. Higher values of Packet Chaos led to mostly stable coverage levels, as while deeper broker packets are not reached, the fuzzer ends up generating all client-side messages.

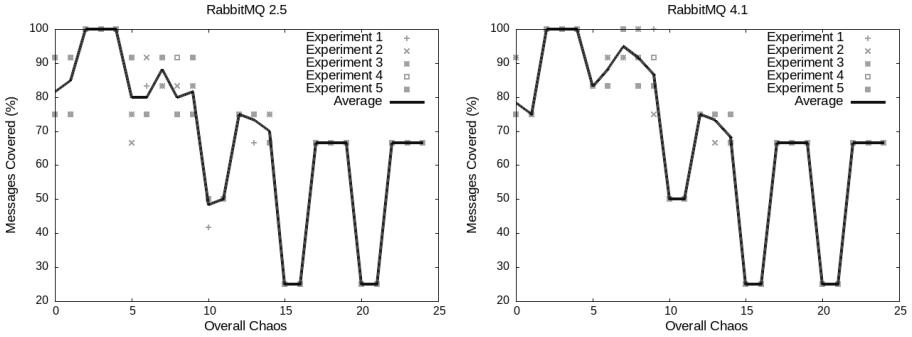

Figure 4. Coverage as a function of the overall chaos (Points for each repetition plus the interpolation of the average)

5. Conclusions and Future Work

This paper presented a proposal to use a grammar-based fuzzing to test implementations of the AMQP protocol. The proposal was integrated in a new free and open-source fuzzer called AMQPGRAM. By analyzing the results, it is possible to see that the exchanged packets data leads to the conclusion that introducing variability during the AMQP Connection exchanges contributes negatively towards testing the protocol, as deeper states, and especially the pub-sub capabilities, are locked behind them. Despite that, the fuzzer was able to reach 100% of messages covered in certain configurations. Future works include: implement the AMQP Classes Channel, Queue and Basic to enable pub-sub testing capabilities; and implement separate Connection class configuration to prevent gating testing behind it.

Acknowledgments

This research is part of the STARLING project funded by FAPESP proc. 2021/06995-0. It is also part of the CNPq proc. 129028/2024-0.

References

Adiwal, S., Ahmed, S. S., Rajendran, B., Misbahuddin, M., and Sudarsan, S. D. (2024). Role of PKI in Securing AMQP Communication. In *Proc. of the IEEE PKIA*, pages 1–8.

Apache (2015). AMQP - Apache Qpid. `https://qpid.apache.org/amqp/index.html`. Accessed at 03/25/2025.

Araujo Rodriguez, L. G. and Batista, D. M. (2021). Towards Improving Fuzzer Efficiency for the MQTT Protocol. In *Proc. of the IEEE ISCC*, pages 1–7.

Broadcom (2025). RabbitMQ: One broker to queue them all | RabbitMQ. `https://www.rabbitmq.com/`. Accessed at 03/25/2025.

Crocker, D. and Overell, P. (2008). Augmented bnf for syntax specifications: Abnf. `https://www.rfc-editor.org/rfc/rfc5234`. Accessed at 03/25/2025.

Gemirter, C. B., Çağatay Şenturca, and Şebnem Baydere (2021). A comparative evaluation of amqp, mqtt and http protocols using real-time public smart city data. In *Proc. of the UBMK*, pages 542–547.

Iqbal, F., Gohar, M., Karamti, H., Karamti, W., Koh, S.-J., and Choi, J.-G. (2023). Use of QUIC for AMQP in IoT networks. *Computer Networks*, 225:109640.

Kwon, S., Son, S.-J., Choi, Y., and Lee, J.-H. (2021). Protocol Fuzzing to Find Security Vulnerabilities of RabbitMQ. *Concurrency and Computation: Practice and Experience*, 33(23):e6012.

Kyzivat, P. (2014). Case-sensitive string support in abnf. `https://datatracker.ietf.org/doc/html/rfc7405`. Accessed at 03/25/2025.

Liang, H., Pei, X., Jia, X., Shen, W., and Zhang, J. (2018). Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218.

Luo, Z., Yu, J., Du, Q., Zhao, Y., Wu, F., Shi, H., Chang, W., and Jiang, Y. (2024). Parallel Fuzzing of IoT Messaging Protocols Through Collaborative Packet Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3431–3442.

OASIS (2008). AMQP Working Group 0-9-1 | AMQP. `https://www.amqp.org/specification/0-9-1/amqp-org-download`. Accessed at 03/25/2025.

Rodriguez, L. G. A. (2023). *Mechanisms to Improve Fuzz Testing for Message Brokers*. PhD in Computer Science, Institute of Mathematics and Statistics, USP.

Shodan (2025). AMQP version 0-9 - Shodan Search. `https://beta.shodan.io/search?query=AMQP+version+0-9`. Accessed at 03/25/2025.

W3Techs (2025). Usage Statistics of Default protocol https for Websites, March 2025. `https://w3techs.com/technologies/details/ce-httpsdefault`. Accessed at 03/25/2025.

Yoshino, D., Watanobe, Y., and Naruse, K. (2021). A highly reliable communication system for internet of robotic things and implementation in rt-middleware with amqp communication interfaces. *IEEE Access*, 9:167229–167241.