AMQP Standard Validation and Testing

Validering och testning av AMQP-standarden

Peter Caprioli
DA222X - Degree Project in Computer Science and Communication
Royal Institute of Technology
Stockholm, Sweden

Examiner: Jeanette Hällgren Kotaleski
Supervisor, KTH: Cyrille Artho
Supervisor, 84codes: Mona Dadoun

April 1, 2020

Abstract

As large-scale applications (such as the Internet of Things) become more common, the need to scale applications over multiple physical servers increases. One way of doing so is by utilizing middleware, a technique that breaks down a larger application into specific parts that each can run independently. Different middleware solutions use different protocols and models. One such solution, AMQP (the Advanced Message Queueing Protocol), has become one of the most used middleware protocols as of late and multiple open-source implementations of both the server and client side exists.

In this thesis, a security and compatibility analysis of the wire-level protocol is performed against five popular AMQP libraries. Compatibility towards the official AMQP specification and variances between different implementations are investigated. Multiple differences between libraries and the formal AMQP specification were found. Many of these differences are the same in all of the tested libraries, suggesting that they were developed using empirical development rather than following the specification. While these differences were found to be subtle and generally do not pose any critical security, safety or stability risks, it was also shown that in some circumstances, it is possible to use these differences to perform a data injection attack, allowing an adversary to arbitrarily modify some aspects of the protocol.

The protocol testing is performed using a software tester, *AMQPTester*. The tester is released alongside the thesis and allows for easy decoding/encoding of the protocol. Until the release of this thesis, no other dedicated AMQP testing tools existed. As such, future research will be made significantly easier.

Sammanfattning

Allt eftersom storskaliga datorapplikationer (t.ex. Internet of Things) blir vanligare så ökar behovet av att kunna skala upp dessa över flertalet fysiska servrar. En teknik som gör detta möjligt kallas Middleware. Denna teknik bryter ner en större applikation till mindre delar, individuellt kallade funktioner. Varje funktion körs oberoende av övriga funktioner vilket tillåter den större applikationen att skala mycket enkelt. Det finns flertalet Middlewarelösningar på marknaden idag. En av de mer populära kallas AMQP (Advanced Message Queueing Protocol), som även har en stor mängd servrar och klienter på marknaden idag, varav många är släppta som öppen källkod.

I rapporten undersöks fem populära klientimplementationer av AMQP med avseende på hur dessa hanterar det formellt definierade nätverksprotokollet. Även skillnader mellan olika implementationer undersöks. Dessa skillnader evalueras sedan med avseende på både säkerhet och stabilitet. Ett flertal skillnader mellan de olika implementationerna och det formellt definierade protokollet upptäcktes. Många implementationer hade liknande avvikelser, vilket tyder på att dessa har utvecklats mot en specifik serverimplementation istället för mot den officiella specifikationen. De upptäckta skillnaderna visade sig vara små och utgör i de flesta fall inget hot mot säkerheten eller stabiliteten i protokollet. I vissa specifika fall var det, på grund av dessa skillnader, dock möjligt att genomföra en datainjektionsattack. Denna gör det möjlig för en attackerare att injecera arbiträra datatyper i vissa aspekter av protokollet.

En mjukvarutestare, *AMQPTester*, används för att testa de olika implementationerna. Denna testare publiceras tillsammans med rapporten och tillåter envar att själv med enkelhet koda/avkoda AMQP-protokollet. Hittills har inget testverktyg för AMQP existerat. I och med publicerandet av denna rapport och AMQPTester så förenklas således framtida forskning inom AMQP-protokollet.

Contents

Chapter 1

Introduction

As IT services become more and more used across the world, companies are struggling to keep up with user demand. Scaling large IT services, especially within the cloud where applications are scattered over a large number of physical machines, requires careful planning in order not to break functionality or introduce bugs.

Application scaling will most likely become an even bigger issue in the near future, as the *Internet of Things* (IoT) is likely to generate huge amounts of data.

1.1 History of application scaling

Traditionally, dating back to the 1980s, companies have been building their own in-house solutions in order to scale their services over more than one physical computer. These solutions were often simple and did not include features which are required by today's standard, such as fault tolerance and error handling.

A popular way of scaling applications in the 1980s was *Remote Procedure Calls* (RPCs), which is a simple way of having an application execute a piece of code in another application, often on another machine over a network [1].

These RPC solutions were often not stable or fault tolerant. If a machine crashed during the execution of one such transaction, the data would most likely be lost or corrupted. Should one company want to integrate their services with some other company, they would most likely have to come up with yet another protocol as their home grown protocols would be incompatible.

As computers became more readily available in companies, programmers started realizing that their applications need to be interconnected. As the current home grown technologies were not good enough for many types of corporate uses, such as banks and government bodies, researchers started looking for a better way to solve the problem. The set of solutions they came up with were named *middleware* [2].

Middleware is an abstract term for a software based layer between applications that automatically handles delivery of messages to the correct recipient. Error checking, message re-delivery and other common issues with the previously RPC based architectures were generally handled automatically by the middleware, without the need for the programmer to anticipate for these problems. As the middleware paradigm grew, it was adopted by many companies and universities, including IBM and MIT [2, p. 25].

While these middleware systems did solve a lot of the problems back then, most of them failed to anticipate the rise of the Internet and the use of widely distributed data centers often spread across the world. In these types of scenarios, these solutions were no longer feasible. One significant problem with middleware up until this point was its synchronous and blocking nature; meaning that messages had to be delivered in the same order they were sent and that only one message could be transferred at a time.

Whilst this may not be a problem for computers communicating in the same building, it is a problem over the internet because of network latency. If a message is sent from one continent to another, there is usually a delay in the order of hundreds of milliseconds. During this time, the sending computer cannot do anything else until an acknowledgement has been received, making the whole system very inefficient.

Another considerable problem was that the middleware solutions were not flexible enough. Companies wanted to create something which is today known as *decoupled applications*, a design pattern that separates large applications by specific functions [3].

1.2 Decoupled applications

In a decoupled application, each function is very specific and separated based on its functional domain, whilst the middleware part of the application acts as a "glue" in-between them, allowing each function to send and receive messages. This often allows an application to scale linearly as it is generally just a matter of adding more computational power to a specific function in order to allow it to process more data.

As an example, consider a social network website. Every time a person uploads an image to the website, it needs to scale the image down to improve load times and reduce storage space. In a traditional scenario, the server side code running on the web server would resize the image, store it in some storage server and write some information about the image to a database.

In a decoupled system, the web server would send the picture as-is to the middleware infrastructure with some metadata describing that it needs to be resized and stored into long-term storage. After receiving the image, the middleware would review the associated metadata and make a decision on where to send it next. In this case, the next destination would be the function that resizes the image.

The resizing function itself is most likely just a piece of code running on a server, which is connected to the middleware. The sole purpose of the function is to receive images and some metadata, perhaps telling it which sizes the image should be resized to. The function does not have any notion that it is part of a social network and can hence be re-used for many purposes.

Once the image has been resized, the newly created smaller image would be sent back to the middleware for further processing, such as being stored into block storage by a 2:nd function and perhaps finally have some data stored into an SQL database by a 3:rd function.

As the social network user base grows and capacity requirement for rendering images increases, the social network only needs to add more servers running more instances of the mentioned functions in order to meet user demand. In addition, if a machine running the resize function were to fail due to a hardware error or otherwise, the middleware would just re-queue the resizing job and send it to another instance of the same function.

Decoupled systems also allow for companies to expand functionality without modifying their current architectures. This is done by adding additional rules to the middleware, such as duplicating a certain type of message and delivering it to a newly introduced function. This is very useful for large applications, as there is no need to modify existing code which is already deployed.

Consider a bank that wants to start notifying its customers when a transaction over a certain amount is attempted. Traditionally, this would have been done by modifying the code which handles transactions. In many applications, especially within banks and financial institutes, this is a very time-consuming task because of all the validation required by laws and financial standards.

However, if the bank was using a decoupled system, a rule could be added in

the middleware that duplicates any transactions over a certain amount and sends it to a newly created function which in turn notifies the customer.

With this approach, none of the banks core systems would have to be modified. The worst-case scenario would be that the newly added function breaks, in which case there would be no significant impact on the banks systems other than that customers will not receive any transaction notifications.

1.3 Publish-subscribe paradigm

During the early 2000's, a new paradigm of middleware named *Publish/-Subscribe* (pub-sub) started gaining popularity [3]. Pub-sub inherits a lot of ideas from previous designs in decoupled applications. As the name implies, the architecture is based on multiple functions that either acts as publishers or subscribers (or sometimes both).

A publisher publishes messages to the middleware that generally contains some sort of payload together with some metadata. A subscriber on the other hand connects to the middleware and tells it what sort of data it is interested in. Then, when a publisher publishes a message, the middleware automatically routes the message to the correct subscriber(s) based on the metadata provided in the published message.

While this approach is similar to a "traditional" decoupled system as described in Section 1.2, the middleware itself would require less configuration and fewer rules. These are instead built dynamically as functions attach and detach from the middleware.

1.4 Introduction of AMQP

In 2007, the bank JP Morgan released a protocol and messaging model called *Advanced Message Queuing Protocol* (AMQP) as an open standard. The protocol had successfully been in production within the bank since 2006 [4]. Their protocol specification [5] includes everything needed to build a fully functional and standardized modern pub-sub middleware.

The idea is to create an ecosystem of software which all adhered to the AMQP specification, and in the long term being able to have multiple implementations from different vendors without compatibility issues, much like there are lots of web browsers and web servers, all compatible with one another.

Today, a plethora of different AMQP implementations exists. The most popular on the server-side is unarguably *RabbitMQ*, an open source project written in the Erlang programming language. AMQP libraries are available in most programming languages, including Java, C, C++, Javascript, Python, PHP, Go and C#.

1.5 Goals

As AMQP has grown in popularity, a lot of companies and institutions rely on its different implementations in order to build their large-scale applications. As such, the goal of this thesis is to research the stability and adherence of different AMQP implementations in order to come to a conclusion whether the current AMQP ecosystem is stable or not, both from a security and stability standpoint.

Were there to exist deviations from the standard in different implementations or if the standard definitions themselves are ambiguous, there may be implications which affects the security, integrity or reliability of these applications.

So far, very limited research within the AMQP protocol exists. Currently, no dedicated AMQP testing tools exists, making it hard to validate AMQP implementations and to arbitrarily send and receive AMQP traffic, due to the complexity of the AMQP protocol itself.

Together with this thesis, a software tester named *AMQPTester* written in the Java programming language is released on Github [6]. This tester allows programmers to automatically run tests against their implementations in order to detect behaviour which is not compatible with the AMQP standard.

Different test cases are included in the source code. Implementing new test cases are as easy as extending a Java class, allowing anyone to receive fully decoded AMQP traffic and programmatically creating responses. Low-level socket access is also provided in order to allow for further protocol fuzzing.

As AMQPTester is very modular, its various AMQP classes (such as different data types, protocol de-/encoding, etc) can easily be re-used for other purposes.

1.6 Scope

This thesis will only look into version 0-9-1 of the AMQP protocol and model, as it is by far the most used version.[1] Newer versions are also more ambiguous by design, as they do not specify the wire-level format of the protocol but rather keep it at an abstract level, allowing different libraries and vendors to implement the wire-level format as they see fit. [7]

This would make it hard to implement a generic tester that works with multiple libraries, as each would require their own "translator" to turn data from an abstract model into their specific network encoding. This is also the reason as to why 0-9-1 is still the most popular version [7].

AMQP 0-9-1 will only be investigated at the application layer. Protocols such as TLS, IP, Ethernet and TCP will be briefly mentioned but are out of scope for the research in this thesis. It will be assumed that these protocols are secure, stable and reliable.

Only the wire-level protocol between the *message broker* (the AMQP server) and the client will be investigated. Many AMQP brokers does support server-to-server communication and some (such as RabbitMQ) also supports message queuing via other protocols such as HTTP, all of which are out of scope for this thesis.

The AMQP protocol and the AMQ model does include Quality of Service, QoS.[2] Most of the QoS logic is however handled within the message broker and not within the client nor the network protocol. QoS is therefore out of scope for this thesis, as only AMQP clients will be tested.

1.7 Problem statement

1.7.1 Research question

How do current AMQP implementations differ from the standard and how do these differences affect real-world applications with regard to stability, determinism and security?

[1] 0-9-1 is the correct notation used to describe the protocol version of AMQP. Other notations, such as 0.9.1, are incorrect.

[2] Quality of Service increases the possibility of certain high-priority messages to reach its destination.

1.7.2 Problem evaluation

AMQPTester implements multiple test cases which tests different aspects of the client-under-test. The evaluation will be done by putting multiple clients under these tests and observing their traffic flow.

In addition, the state of the client-under-test will be examined after each test along with manually inspected network traffic to reach a verdict.

1.8 Sustainability and Ethics

Implementing middleware solutions may have the positive side effect of being able to perform more computational calculations per kWh rather than operating traditional applications. This does not only allow companies and organizations to reduce their energy costs and operate more cheaply, but also reduces the environmental impact.

One concrete example is shown in Section 2.1.2. Utilizing the same technique in large-scale applications can potentially be more sustainable than today's solutions where many companies run their applications on either virtualized or physical servers, which draw the same amount of power regardless if they are being fully utilized or not.

In addition, there is generally less overhead in middleware virtualization because there is no need to run a separate operative system on every virtualized machine. This could potentially mean that middleware is more efficient than older virtualization techniques.

Regarding the ethical point of view of this thesis, the conclusion was made that there are no ethical problems with the research in this thesis. The closest ethical problem may be the presented data injection attack explained in Section 4.3. This attack is however quite hard to exploit in practice, and it was decided that production systems using AMQP will most likely not be vulnerable in any significant scale.

1.9 Outline

The structure of this thesis is as follows:

- Chapter 1 explains the history of middleware from the 1980's until today. This includes the motivations for the design decisions that were

made which has lead to today's different middleware solutions, including AMQP. The motivation and aim for this thesis is also explained, along with the goals, scope, sustainability and ethics parts.

- Chapter 2 presents the current state of research into middleware and AMQP. It provides an extensive overview of the AMQP protocol, along with other details needed in order to understand the following chapters.

- Chapter 3 gives both an overview and details of the methods used to verify protocol conformity. This includes both details on the tester designed alongside this thesis, *AMQPTester*, and more insight into how different tests are performed.

- Chapter 4 presents a table of results, showing the level of conformity each client under test managed to achieve. Further details into each library is also presented.

- Chapter 5 reflects over the obtained results and suggests different relevant research which may be considered in the future. Various future improvements to the included software tester is also discussed.

- Chapter 6 answers the research questions and reflects over the research presented in this thesis.

Chapter 2

Background

This chapter explains the theoretical foundations of AMQP. First, related research into both AMQP and Middleware is presented. Then, the Advanced Message Queuing Model (AMQ model) will be briefly explained. Then, a brief introduction to transport protocols will follow, with focus on TCP/IP.

Then, a more thorough description of the AMQP protocol will be presented along with its framing format, parameter and data encoding. Then, the various high level operations such as message publishing and subscription will be explained.

Finally, a brief description of the various AMQP libraries that are tested in this thesis are presented.

2.1 Related research

This section is split up into two different sections. One section is dedicated to research within AMQP, and the other to research within middleware in general.

2.1.1 Research into AMQP

Currently, despite being a popular and widely used protocol, there exists very limited research into AMQP. While performance testing of the protocol and various message brokers and clients are abundant, research within the wire-level protocol itself is almost non-existent.

Luzuriaga et al. have written an evaluation [8] of the robustness of the AMQP and MQTT (MQ Telemetry Transport, another pub-sub protocol similar to AMQP) protocols over lossy networks, but does not consider other scenarios such as what would happen if the traffic got corrupted or was actively modified by an illicit adversary.[1]

Their work uses pre-existing server software and their client implementation uses an existing library which handles all the wire-level aspects of the AMQP protocol. In their paper, they simulate network scenarios that could be expected in moving vehicles roaming between different radio networks.

In practice, they use multiple Wi-Fi access points with the same SSID. During the test, they turn these access points on or off, forcing the connected client to roam onto another access point. During that time, they keep sending messages to and from the AMQP broker and the losses and latency were recorded.

Each Wi-Fi access point shared the same layer 2 network subnet, allowing the roaming client to keep its IP address as it switched between the different access points.

This, in turn, makes it transparent to the IP and TCP stacks that the client has changed its physical connection point to the network. As expected, roaming between access points only interrupted traffic for a small amount of time.

During this short time, their AMQP client library started queuing messages using the Last In First Out queuing discipline. This had the effect that some messages were delivered in reverse during the time of roaming. However, no messages were ever lost in their testing.

Subramoni et al. [9] have written an evaluation of AMQP over two lower level protocols, namely Ethernet and (TCP/IP over) Infiniband. In their paper, they simulate different scenarios where AMQP could be used within financial applications, such as within a stock exchange.

In order to do so, tests were performed which include clients sending messages to one or multiple other clients at the same time. These messages differ in size and quantity, and each test is performed over both Ethernet and Infiniband.

The authors tweak the underlying networking stacks (both Ethernet, Infiniband and TCP) in order to achieve maximal performance. It should be noted that these tests only were performed with Apache Qpid on the server side, and the performance results may hence be a limitation of that specific

[1] Lossy networks: Networks that are unreliable and partially drop traffic.

server software.

2.1.2 Research into middleware

Research within middleware in general is more abundant than research specifically into AMQP. Most research is however conducted on the network-layer and generally not within the application layer of the middleware.

Eugster et al. has written an excellent paper [3] about the different ways a pub-sub middleware architecture can be designed and what the positive sides and drawbacks of each designs are.

In their paper, they describe many common ways to implement middleware and discuss the positive and negative sides of each design. Different ways of routing messages from point A to point B can be either cheaper or more expensive in terms of overhead and network capacity.

Middleware also plays a crucial part in cloud development. As shown by Walraven et al. in [10], middleware can be used to create multi-tenant cloud applications at a lower cost by virtualizing inside the middleware layer rather than having a separate middleware instance for each tenant.

They argue that as Software as a Service (SaaS) and Platform as a Service (PaaS) become more and more widespread, it is important to adapt new technologies in order to more cheaply scale up the number of users with less hardware.

While this can be done closer to the hardware (buying more servers, running more virtual machine instances, etc), it is cheaper to do so in software. In particular, virtualization gets cheaper the closer to the application it is.

In their paper, they show that virtualization within the middleware can be done as cheaply as within the application itself. While this approach has significant cost advantages, they also point out that there are some problems that needs to be resolved.

One of these problems are performance control, which would be required in order to limit customers such that they cannot use all resources and slowing down the entire application for everyone else.

Another paper by Foley et al. [11] presents a practical way as to how PKI can be used to isolate different middleware systems based on their trust levels.[2] Their solution uses a role-based access control system which assigns roles to different users, depending on the level of access they need.

[2]Public Key Infrastructure (PKI): A standardized cryptographical trust hierarchy.

Each user is assigned a public-private key pair. During every operation within the middleware, their private key is used to sign the intent of the message sent over the middleware. When the intent is consumed by some function within the middleware, the signature and authorization of that particular user is validated before the action is executed.

This has the advantage of reducing risk within the middleware, as a potential adversary with access to the middleware infrastructure will be unable to produce a valid signature, rendering his or her access to interconnected systems at a minimum.

In addition, this approach causes more decoupling in-between systems, as there is no need for a central authentication or authorization service that must be online at all times.

Artho et al. have tested [12] the MQTT protocol robustness using a model-based state machine known as Modbat[13]. As explained in their paper, MQTT has three different Quality of Service levels. Depending on the QoS level, MQTT either delivers a message at most once, at least once or exactly once.

In their paper, two MQTT clients were set up along with one server. Between each client and the server, a TCP proxy was inserted, allowing Modbat to either close the TCP connection or delay data delivery. Which of these actions were taken and at what time was (randomly) decided based on weights in the graph that models the testing behaviour.

The implemented TCP proxy never modified any application data, nor did it deliver messages out-of-order. The tested MQTT implementations were shown to be stable with regards to how the different MQTT QoS levels are defined.

The MQTT protocol was formally modelled by Manel Houimli et al. in [14] using UPPAAL SMC, a model checker used to model timed automata [15]. An informal model, using multiple UML diagrams, is also presented. Using both of the models, a performance evaluation was made.

UPPAAL SMC allows for queries to be made against the model, after which a decision on whether the query is satisfiable or not is given by the software. Multiple queries were made in order to formally prove different properties of the MQTT protocol, such as the ability of one client being able to publish a message to another client within a certain time frame and the available performance based on the number of connected clients. Queries testing liveness and safety were also performed. [15]

2.2 The AMQ model

The AMQ model describes the behaviour of a message broker and client at a higher level; it is a part of the AMQP specification and it defines how a message broker and client must offer each other services and how messages should be handled and routed in order to be compliant with the standard.

The AMQP model defines three [16, p. 14-17] components which must be implemented in a broker:

1. Queues
2. Exchanges
3. Bindings

2.2.1 Queues

A queue is a component that stores messages which are to be delivered to one or multiple clients. A queue can store store messages either on disk or in RAM, depending on how the queue was set up. AMQP also supports topic queues, which is a type of queue that performs routing based on what topics a sender included in a messages metadata.[3] When a client subscribes to a topic queue, it also has to specify which topics it is interested in receiving.

2.2.2 Exchanges

An exchange is a component to which clients publish messages. When a message arrives at an exchange, the exchange is responsible for routing the message to the correct queue(s). It is also possible for one exchange to route messages to another exchange, effectively creating a routing chain. Exchanges can (depending on the implementations and extensions) support multiple modes of routing behaviour. For example, a *fanout exchange* delivers messages to all its known destinations, while a *topic exchange* deliver messages based on what topics are defined within each message metadata.

[3] A topic is defined by the publisher of a message and is part of the message metadata. A topic is usually a dot-delimited string such as *image.jpeg.resize.thumbnail*. As to extend on the previously discussed social message website example, such a topic could indicate that the message contains a JPEG image destined to be resized to a thumbnail format.

2.2.3 Bindings

Binding are rules that define the routing behaviour of exchanges; effectively creating the rules dictating how they should route incoming messages. When a binding is created, the affected exchanges are notified and given the set of rules on how it should route incoming messages.

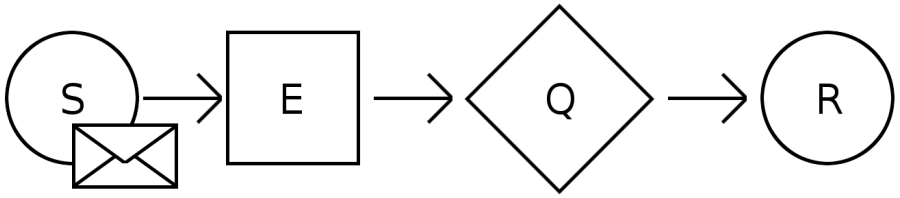


Figure 2.1: Trivial example of the AMQ model.

Figure 2.1 depicts a trivial example of the AMQ model. It contains a publisher S, and exchange E, a queue Q and a subscriber R. The message (depicted by the letter) would flow from the left to the right, until it eventually reaches its destination and is consumed by the function application running on the R node.

2.3 Transport protocols

While AMQP can technically be used over any [5, p. 22] communication network supporting error checking and a byte-stream oriented data transport, the *Transmission Control Protocol* (TCP) over IP tends to be the preferred underlying transport protocol which most AMQP implementations are using.[4,5]

TCP and IP (amongst others) are protocols that make up the wider parts of the internet. These are part of the *Open Systems Interconnection Model* (OSI Model) [17] as depicted in Figure 2.2. This is an abstract and a conceptual model which standardizes the various protocols needed in order to build a functional larger network, such as the internet.

The model consists of 7 layers, where each layer is responsible for a certain aspect of the communications network. Lower layers tend to be more hardware bound, while higher layers usually are implemented in software.

[4]Error checking: If data is lost or corrupted over the network, the sender automatically and transparently re-sends the same piece of data until it has been correctly delivered.

[5]Byte-stream oriented: Data is sent as a stream of bytes (i.e. the total number of sent bits is divisible by 8), each byte is delivered in the same order as it was sent.

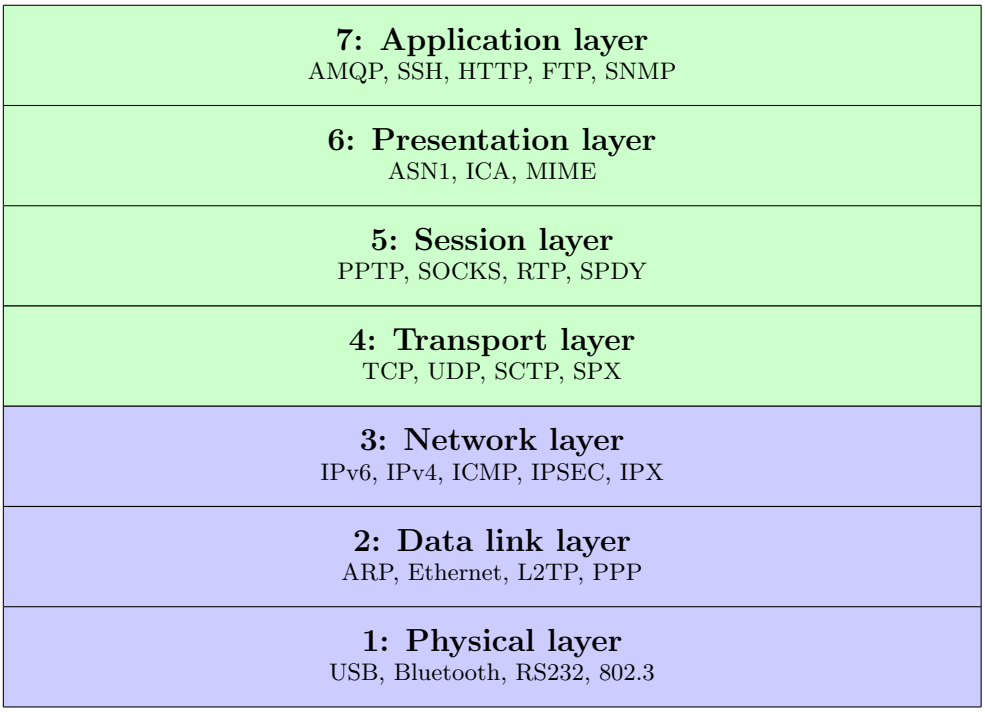


Figure 2.2: The OSI model along with some commonly used protocols at each layer

Layer 1 defines how data should be modulated and transmitted physically over a network cable (or other physical means) between two peers. This layer does not concern itself about anything else, such as how the transmitted data is structured or what its contents are. Layer 2 uses the functionality of layer 1 in order to send frames to peers on a locally connected network, such as a switched LAN, often called a Layer 2 network. Layer 3 builds upon the framing structure in layer 2 in order to provide routing through other peers on the network, effectively creating a much larger network (such as the internet).

The *Internet Protocol* (IP) is by far the most widely used protocol within the 3rd layer of the OSI Model. It defines [18], amongst other things, how a network of peers should transmit datagrams (packets) in-between themselves in order for two peers to communicate without being directly connected.

Whilst IP does include some error checking on its own headers, it does not provide any error checking on the transmitted data payload [18, p. 14]. This means that layer 4 and above has no guarantee that the destination peer will receive the exact same datagram as the source peer originally transmitted.

There is also no guarantee that datagrams will be delivered at all, nor that

they will be delivered in the same order as they were transmitted [18, p. 3].

One way to resolve these issues is by using TCP. TCP is running directly on top of IP within layer 4. It provides, amongst others, automatic re-transmission and error checking, making up for the lack of these features within the IP layer. Should a TCP/IP datagram be lost or inadvertently altered, TCP will automatically re-transmit the data until it has arrived in a correct manner.

TCP is inarguably the most widely used protocol within the 4th layer of the OSI model and is hence very well supported in most network environments.

In addition to error checking and correction, TCP also provides a connection oriented byte-stream data transport. In practice, this means that any data transmitted on one end comes out in the same order at the other end with a negligible risk of corruption. Should the transmitted data be longer than the length of an IP datagram, it is automatically split into multiple datagrams by the TCP implementation.

TCP does not provide any security against an illicit adversary which actively modifies traffic over the network, but other security protocols such as TLS, IPSEC or SSH can be used to mitigate this type of attack. Generally, TLS is used to secure AMQP in most real-world scenarios [7], providing both encryption and authentication of the transmitted data.

2.4 Protocol notation

In this section, the notation used to describe the AMQP protocol in the rest of this thesis will be presented. In order to understand the AMQP protocol, examples containing the AMQP wire-level framing format will be presented.

These will be shown with boxes around them using different notation depending on what type of data is being presented. Text within apostrophes are to be interpreted as ASCII strings, meaning each character is exactly one byte (or 8 bits) long:

'Hello World'

Integers are to be interpreted as one byte each, ranging from 0 to 255:

72 101 108 108 111 32 87 111 114 108 100

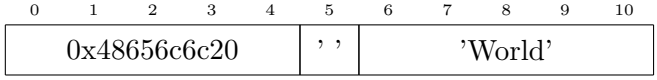
Hexadecimal strings will always start with *0x* and are to be interpreted as one byte for every 2 character, ranging from 0x00 to 0xFF. Hexadeximal strings can also be concatinated to represent a longer byte sequence:

```
0x48656c6c206f576f726c64
```

Comments may be used to clarify wire-level data:

```
0x48656c6c20 // Hello
0x6f // Space
0x576f726c64 // World
```

In addition, packet structures will also be used to represent more complex data structures, allowing for a better overview of certain packet and frame types:



Each number on top of the box denotes eight bits or one byte of data.

All of the above boxes represents the same wire-level data ("Hello World" in ASCII), represented in different formats.

The chosen format for the examples will differ depending on the situation and what type of data is being represented. Replacing "0x" with "0b" follows the same notation as in most programming languages, denoting values in base 2 instead of base 16.

2.5 AMQP protocol

In this section, the AMQP wire-level protocol will be thoroughly explain.

2.5.1 Handshake and framing format

Similar to most other TCP protocols, it is the client that initiates the connection to the broker. In AMQP, the standard TCP port of the server is 5672. Once the TCP connection has been established, a "handshake" message is sent by the client:

```
'AMQP' 0x00 0x00 0x09 0x01 //AMQP v0-9-1
```

The broker received the handshake string and validates it and its embedded AMQP version. Should the message broker use an incompatible version, the broker responds by sending the expected header string and then closes the TCP connection.

After the handshake has been accepted by the broker, both peers will there-

after only transmit AMQP *frames* [5, p. 21], as explained below. The handshake string is the only exception as to when a peer is allowed to transmit any data not adhering to the framing format.

The AMQP framing format consists of 4 parameters and an *End of Frame* (EOF) byte, which is always transmitted in the same order:

1. Frame type - 1 byte
2. Channel number - 2 bytes
3. Payload size - 4 bytes
4. Payload - Variable length according to point (3)
5. EOF - 1 byte - Set to `0xCE` as per the AMQP standard

An example of a complete frame containing the message "Hello world" sent on an AMQP logical channel 4 is depicted in Figure 2.3.

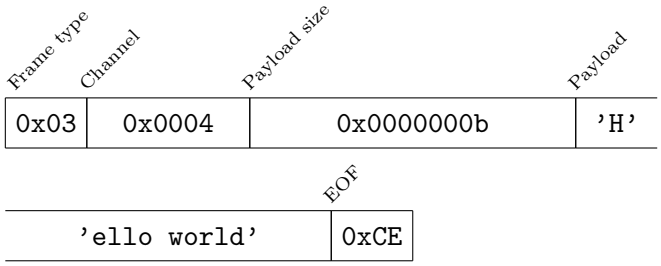


Figure 2.3: Trivial AMQP frame

The first byte in every frame, the frame type parameter, describes what sort of data is contained within the frame. Regardless of which frame type is being transmitted, the frame structure is always the same. AMQP defines 4 different frame types:

1. Method frames
2. Header frames
3. Body frames
4. Heartbeat frames

2.5.2 Method frames

Method frames act much as an RPC request to the other peer, making a request for it to execute some kind of action such as opening a new logical channel or creating a queue.

Method frames are by far the most complex frame type in AMQP as they contain nested data structures, often in multiple layers and with different encoding schemes. All of this data is encoded within the frame payload.

Some method frames are defined as *content-carrying frames*. These are encoded just like any other frame, but are formally specified such that they inform the other peer that application data, such as an image or file, is about to be transmitted.

2.5.3 Header frames

A header frame is always transmitted after a content carrying frame [5, p. 24]. Header frames carry metadata, such as content type, content length and various flags depending on what type of content is being transmitted. To illustrate this, consider the succession of sent frames in Figure 2.4.

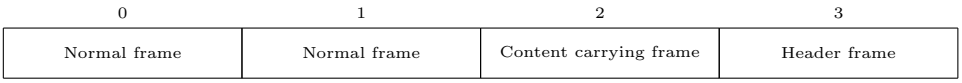


Figure 2.4: Trivial example of content carrying frame ordering

Exactly one header frame is sent in succession after a content-carrying frame.

2.5.4 Body frames

Body frames only contain application data and are arguably the simplest frame type in AMQP, as they only contain the payload length, channel number and the payload itself. Multiple body frames can be transmitted after one another in order to carry larger payloads. Body frames are only transmitted in succession after a header frame as depicted in Figure 2.5.

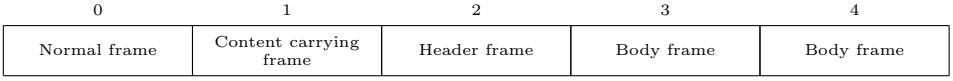


Figure 2.5: Multiple body frames being transmitted

The previously presented frame in Figure 2.3 is a valid AMQP body frame.

2.5.5 Logical channels

Each frame has a channel number that is used to separate one single TCP connection into multiple logical ones. This allows AMQP to very cheaply multiplex data without the need to open multiple connections in order to perform multiple actions such as receiving and sending messages at the same time.

Channel 0 is a reserved channel number and is used by default for all initial communication between the two peers. It is also used as an administrative channel. Application data (such as payload data transmitted by a consumer) is never transmitted on channel 0.

2.5.6 AMQP data type encoding

The AMQP specification defines multiple different data types such as integers of different sizes, strings of different sizes, booleans, different types of arrays and other nested data structures.

Some of these data types (such as strings) have variable length. When data is sent over the wire, AMQP encodes the length of the data in the beginning, just as in most other network protocols. For example, a data type called a *short string* consists of 1 byte indicating the length L of the string, and then the string itself. This is depicted in Figure 2.6.

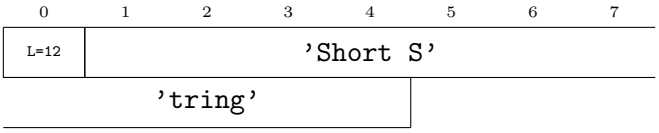


Figure 2.6: Encoding of a short string

A *long string* uses the same encoding with the exception that it uses 4 bytes to denote its length, as seen in Figure 2.7.

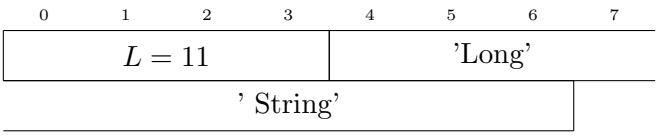


Figure 2.7: Encoding of a long string

A *field table* is a data structure that reassembles an associative array. It contains a dynamic number of key-value pairs where each key is always a short string bound to a value of varying type. An example of a trivial field table containing only one key and one value is shown in Figure 2.8.

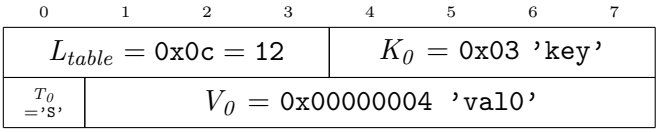


Figure 2.8: AMQP field table data type encoding

The various fields are explained as follows:

1. L_{table} is the length (in number of bytes) of the entire encoded field table, excluding L_{table} itself. If the field table were to have zero elements, L_{table} would be zero as well.
2. K_0 is the key for the first value, which is always implicitly defined as being of type short string. Here, `0x03` is the length of the short string "key".
3. T_0 is a single byte describing which data type V_0 consists of, as specified in [5, p. 31-32]. In this case, an upper case ASCII letter 'S' corresponds to the data type being a long string.
4. V_0 is the value corresponding to key K_0. Because it is defined as a long string, it must contain 4 bytes denoting the string length, directly followed by the string itself.

In the above example, the key `key` maps to the long string value `val0`. Field tables can contain any other AMQP defined data type, including nested field tables and arrays.

As each value always has a well-defined length (either fixed or prepended depending on the data type), there is no need for the field table to store lengths of each encoded value as these can be derived during the decoding process.

The AMQP specification contains definitions of integer types of lengths between 1–8 bytes, which are always unsigned [5, p. 22]. *Booleans* are encoded using C-style, i.e. a single byte which is to be interpreted as `false` if all bits are zeroed out, otherwise it is to be interpreted as `true`.

2.5.7 Property flags and lists

AMQP defines two data types called *property flags* and *property lists*, which both works in conjunction to create a list of values, much like different flags can be set in TCP or 802.11 type frames.

This data encoding is more compact than field tables and does not contain any data type declarations sent over the wire, nor does it contain any keys. These are instead defined in the specification, making it an unnecessary overhead to include within the protocol itself.

The property flags encoding is always the length of a multiple of 16 bits or 2 bytes. The 15 MSB (Most Significant Bits) in each 2 byte sequence defines whether each corresponding field is being set or not.

The LSB (Least Significant Bit) is set to one if there are more property flags to follow, otherwise zero. Consider a property field containing 20 possible options, with option 0 and 19 enabled, as depicted in Figure 2.9.

0	1
0b10000000 0b0000000 0b1	
0b00010000 0b0000000 0b0	

Figure 2.9: Property flags encoding in base-2

The blue bits denote that options 0 and 19 are set, as they reside on the 0:th and 19:th index position respectively. The green bits denotes whether or not more property flags are to follow. In this example, the first green bit is set, because there are more options to follow.

Directly after the property flags, the list of each encoded property must follow in the same order as the flags are numbered. As each data type is already well-defined in the AMQP specification, there is no need to transmit any data types nor any value lengths, as these are derived in the same manner as in the previously explained field tables.

Consider the property flags in Figure 2.9. Under the premise that

1. Option 0 is defined as a short string carrying data named *username* with the value `J.Smith` and
2. Option 19 is defined as short integer of 2 bytes containing the maximum frame size S_{frame} with the value `1500`

the encoding of both the property flags and property list would be as shown in Figure 2.10. $L_{username}$ is the length of the short string named *username*.

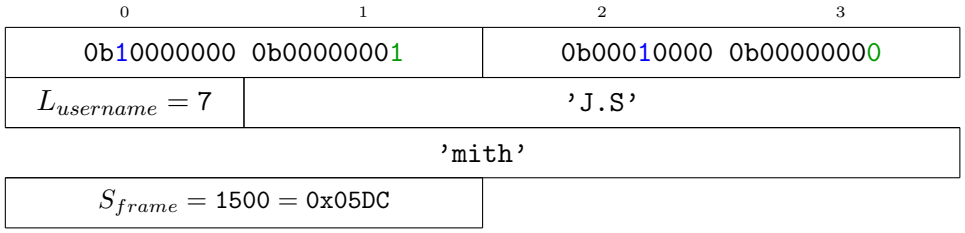


Figure 2.10: Complete example of property flags and a property list

Property flags can also be used without any following property list in order to enable or disable certain functions within the protocol. Consider an example where the client would notify the broker that, in case a message cannot be delivered, it should be returned to the sender instead of being dropped. In such cases it is more efficient to simply use the property flags to directly encode a `Boolean` rather than indicating that a boolean is to follow.

2.5.8 Method frame RPC

In order for two AMQP peers to communicate and issue requests and intents to each other, AMQP uses both the AMQ model and the method frame wire-level format. The AMQ model describes which, how and when the various pre-defined method frames should be transmitted to the other peer[5].

Each method frame consists of a class and a method name along with an argument list. The argument list is sometimes optional, depending on which class and method is being used.

Each class represents a set of methods within a functional domain and each method is generally very specific to a certain task. While this design does result in quite a big set of total methods, it decreases overall complexity as each method is often easier to understand since its functionality is very specific.

Each class/method pair is defined as being either client-to-broker or broker-to-client, or both in some cases.

2.5.9 Opening channels

Consider an example of two peers using AMQP over a network; one message broker and one client, as depicted in Figure 2.11.

Figure 2.11: A broker and a client in a connected and authenticated state

As a rough analogy, each AMQP class (`Connection`, `Channel`, etc) may be viewed as Java class running within a JVM on each peer. Each of these Java classes would have their own state machine, as defined by the formal AMQP specification.

As the two peers send method frames to one another, the receiving peer calls the corresponding method on each object together with the received set of arguments, changing its state machine and possibly performing other tasks.

Consider the `Channel.Open()` call, depicted in Figure 2.12, which is the method used to open a new logical channel in an already existing AMQP connection.

The client sends a method frame with the class ID `Channel` and method ID `Open`. Once received by the broker, The *Channel* class allocates the newly opened channel and updates the brokers state.

Upon successfully allocating the requested channel, the broker responds with `Channel.Open-OK()`, indicating that the channel is ready to be used. As neither of these methods contains any arguments as defined in the AMQP specification, the argument list is empty.[6]

The `Channel.Open()` method frame is directly sent on the channel number for which the client wants to open. This way, the channel integer number does not have to be included in the argument list, thus making the protocol more compact.

[6] `Channel.Open()` and `Channel.Open-OK()` do define one argument each, but both are reserved for future use and are nulled out in AMQP 0-9-1.

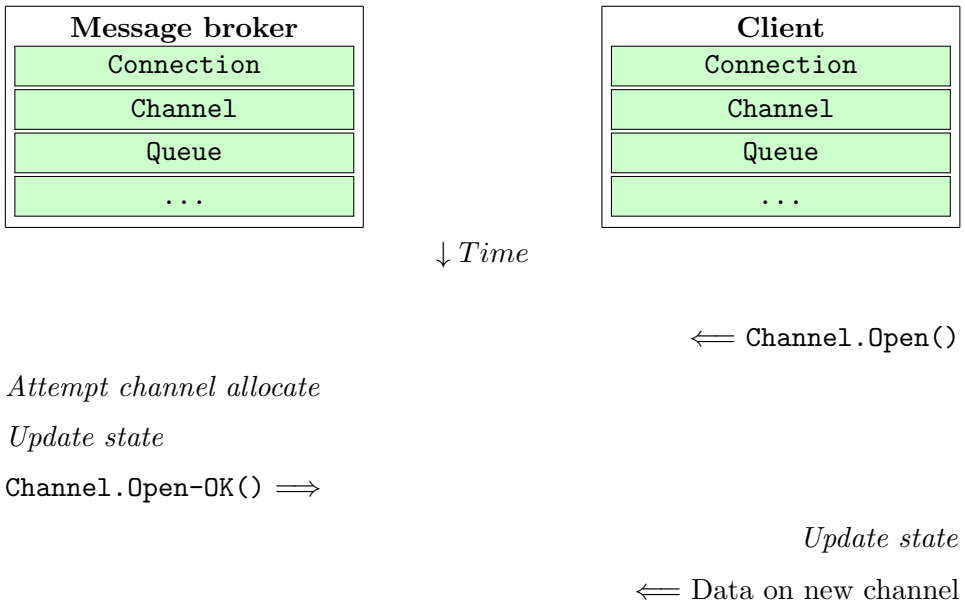


Figure 2.12: Successfully opening a new channel

2.5.10 Protocol exceptions

Similarly to Java, AMQP defines exceptions within the protocol. Should a method frame call fail, the receiving peer will respond with the `Close()` method on either the `Channel` or `Connection` object, depending on the severity of the exception. The `Close()` method call is sent using the same method framing format as used by non-exception method calls.

The `Close()` call will contain information about the class and method call that triggered the exception, together with an error code and a human-readable string that explains what went wrong. The purpose of the human readable string is for logging and debugging purposes.

Consider the previous scenario where a client would like to open up a new channel. In this example, the broker has been configured with a limit of the maximum number of allowed channels, which the client has already exceeded.

As the client sends `Channel.Open()`, the broker responds with a method frame `Channel.Close(args)`, as depicted in Figure 2.13. It should be noted that `Channel.Close()` is an exception call, in this case sent to the `Channel` object as the severity is not high enough to terminate the entire connection.

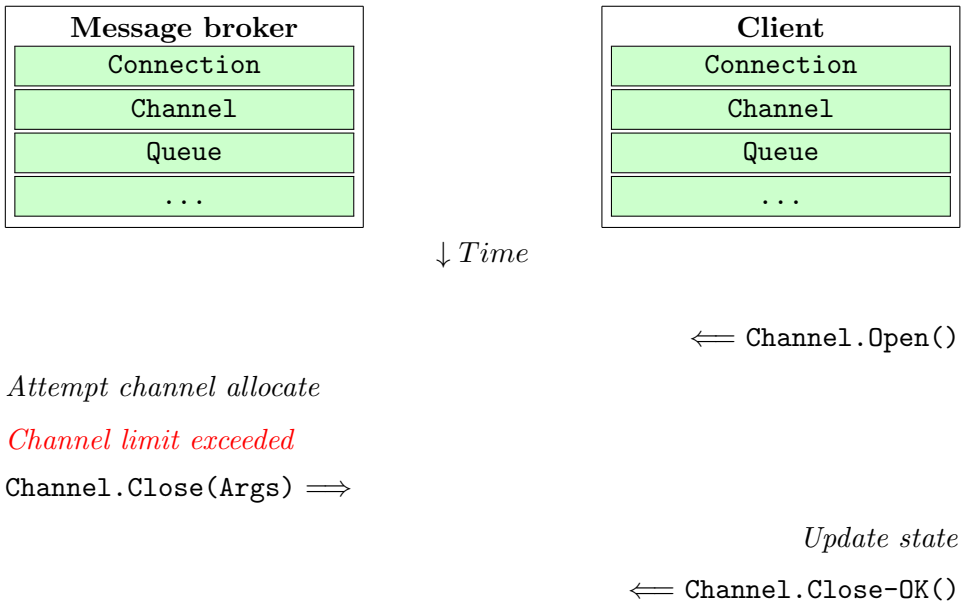


Figure 2.13: Exception when opening channel

The argument list provided by the broker would contain information about why the channel was closed:

```
Args = {
      Reply-code:  406, //The 406 code is defined by the
                        //specification for this error
      Reply-text:  "Reached channel limit",
      Class-id:  Channel,
      Method-id:  Open
}
```

When the client receives the exception, it will deallocate the specific channel and will not use it for any further communication until it has been successfully re-opened.

In many AMQP implementations, the protocol exception is also converted to a programmatic exception and thrown to the implementation of the middleware function code, making the programmer who wrote the function code responsible for handling these types of protocol exceptions.

Whenever an exception is thrown, the connection or channel can no longer be used. When an exception occurs in the `Connection` object, the entire TCP connection is torn down by the peer that threw the exception. This also implies tearing down all logical channels.

2.5.11 Synchronous method frames

The AMQ model defines two types of method frame blocking modes; *synchronous* and *asynchronous*. If a peer sends a synchronous frame on a channel, the receiving peer is obliged to return an acknowledgement [5, p. 18].

During the period in-between sending such a frame and receiving an acknowledgement, the sending peer is not allowed to send any more synchronous frames. Asynchronous frames can still be sent regardless of whether there is an ongoing synchronous frame in transit or not.

Synchronous blocking is applied per logical channel, not per AMQP connection. Most methods in AMQP are asynchronous and requires no confirmation in order to reduce protocol chattiness and improve transmission speed. It is assumed that these frames are delivered unless an exception or other error is raised.

Whether a class/method call is synchronous or not is formally defined in the AMQP specification; it is not negotiated between the client and broker during the connection.

2.5.12 Method inner frames

Method frames are nested within the normal AMQP frame format. In this thesis, the term *inner frame* will be used to denote a method frames inner structure.[7]

Inner frames are colored in light green whilst the outer parts of a frame are colored in light blue. An example of an inner frame encoded in an AMQP frame is depicted in Figure 2.14. Together, the inner and outer frame makes up a complete method frame.

[7] The term "inner frame" is used in AMQPTester code as well [6].

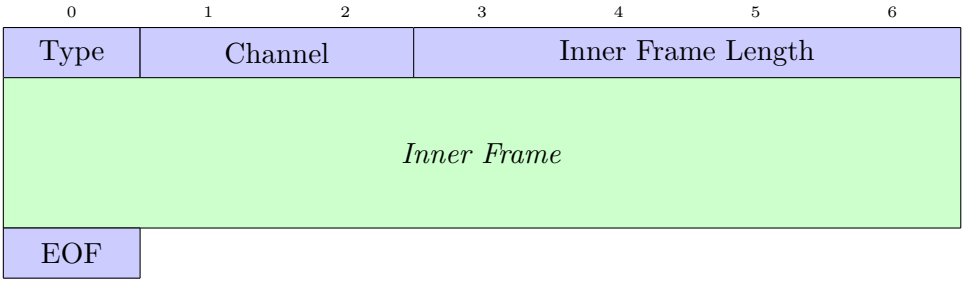


Figure 2.14: Inner frame nested within a method frame

Each inner frame carries exactly one class name and one method name, both represented each as a 16 bit or 2 byte pre-defined integer to make wire-level frames more compact.

A variable length of arguments depending on the class/method pair is encoded within the inner frame, colored in light red and depicted in Figure 2.15.

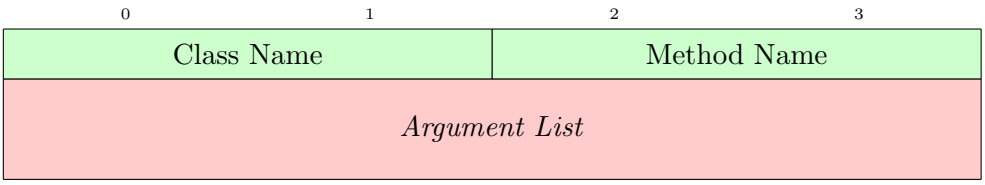


Figure 2.15: A complete inner frame with an encoded argument list

The *argument list* is yet another nested data structure that contains additional data related to the class/method call.

2.5.13 Inner frame argument list encoding

Each method frame has a mandatory list of argument that needs to be supplied when being transmitted over the wire. This mandatory list is defined for each class/method call in the AMQP specification[5].

The specification defines the data types for each argument within the argument list. Some method calls do not specify an argument list, in which case it is left out from the wire-level protocol.

The encoding [5, p. 31] of the argument works the same way as the previously mentioned property lists. Each argument is appended after the previous one, in the same order as listed in the AMQP specification.

As the specification defines which arguments (including their data types) should exist in each method frame, there is no need to include the length of the argument list within the inner frame.

In the specification, each argument has a name associated with it together with a description of the argument. For example, when executing `Queue.Declare(args)`, the specification mandates that `args` contains an argument named `queue` which is the name of the queue to be declared.

Some method frame contains both property lists and property arguments, as described previously. A complete example of a method frame is presented in Figure 2.16, depicting the class and method call `Connection.Tune()` used in the initial connection phase to negotiate parameters between the broker and the client.

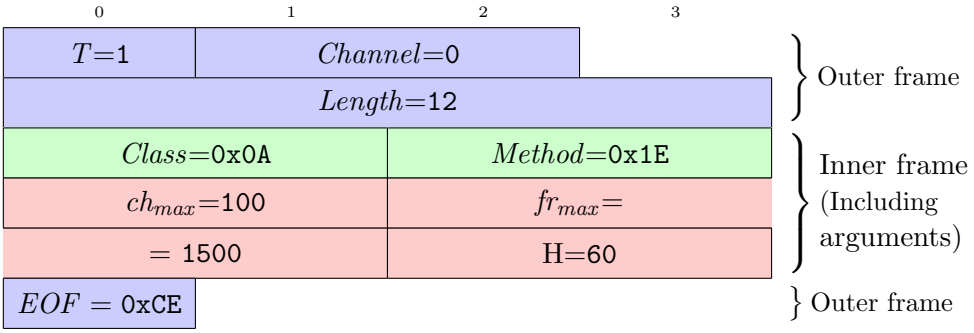


Figure 2.16: Example of a complete method frame

The parameters are explained below:

1. T is the frame type (1 = Method)
2. *Channel* is `0`, as demanded by the specification for this particular class/method
3. *Length* is the length of the inner frame
4. *Class* is `0x0a = Connection`
5. *Method* is `0x1e = Tune`
6. 1st argument ch_{max} is the proposed maximum allowed number of channels
7. 2nd argument fr_{max} is the proposed maximum frame size
8. 3rd argument H is the desired heartbeat period in seconds

9. EOF - End of Frame is always `0xCE`

The three arguments within the argument list are of types `short-int`, `long-int` and `short-int`.

As to extend on the previous Java analogue, sending this frame to a message broker would roughly correspond to remotely calling a method `Tune()` on an object `Connection`:

```
Connection.Tune(100, 1000, 60);
```

The Java `Tune` signature could look similar to:[8]

```
Tune(int maxAllowedChannels, long proposedFrameSize,
int heartBeatPeriod)
```

2.6 UTF-8

UTF-8 is a widely used character encoding scheme that encodes letters and symbols in different languages [19]. It is backwards compatible with most of the ASCII [20] standard, meaning there is a one-to-one mapping between all English letters, numbers and the most commonly used characters and punctuations.

While ASCII always encodes one character using one byte, UTF-8 may use up to four bytes in order to increase the set of all possible characters. UTF-8 always encodes data in bit sizes divisible by 8, i.e. exactly 1, 2, 3 or 4 bytes long.

Contrary to ASCII, this has the side effect that the number of characters in a UTF-8 string may not equal the number of bytes contained within the string.

AMQP denotes lengths of data types (such as strings) in bytes, while it at the same time uses UTF-8 encoding[5, pp. 35].

2.7 AMQP libraries

In this section, a brief description and background with regards to the five tested AMQP libraries will be given. The criteria for selecting these libraries

[8]Note that Java by default handles integers as signed, while AMQP handles them as unsigned. Java 8 and later does include functionality within the `Integer` class to threat integers as unsigned.

were such that two or more should not be written in the same programming language. They should also be at least somewhat popular and be in active development.

2.7.1 PHP-amqplib

Php-amqplib is an AMQP library written in PHP. It is endorsed by RabbitMQ and used in their tutorials. According to their Github page [21], the implementation supports AMQP 0-9-1 and has been tested against RabbitMQ.

The library also supports RabbitMQ extensions to the AMQP protocol, such as binding an exchange directly to another exchange. However, as these extensions are not officially supported in AMQP 0-9-1, they are not relevant for this thesis.

In this thesis, version 2.7.2 was tested.

2.7.2 AMQP.Node

Amqp.node [22] is an AMQP library written in Javascript. The API is mostly asynchronous and uses either Javascript *promises* or callbacks in order to deliver data from the broker.

In this thesis, version 0.5.5 was tested.

2.7.3 Py-AMQP

Py-amqp [23] is an AMQP library written in the Python programming language. The project was originally forked from another AMQP library, and additional support for receiving data on multiple channels and support for timeouts was added (amongst others).

The library also supports multiple RabbitMQ extensions to the AMQP protocol. In this thesis, version 2.3.2 was tested.

2.7.4 Rabbitmq-C

Rabbitmq-c [24] is an AMQP library supported by RabbitMQ written in C. The author specifies that the library is "for use with RabbitMQ", but also

mentions in the documentation that other AMQP brokers should work as well.

Rabbitmq-c implements an event-driven API with callbacks. In this thesis, version 0.9.0 was compiled and tested on a vanilla Linux Ubuntu 16.04 system.

2.7.5 RabbitMQ Java Client

The RabbitMQ Java Client [25] is an AMQP client written in Java by the same team that maintains RabbitMQ. As such, they provide code examples and API references on the RabbitMQ website.

The API is event-driven and extensively supports the AMQP 0-9-1 protocol. Version 6.0.0.M2 was tested in this thesis.

2.8 Summary

AMQP is a widely used pub-sub and decoupled middleware solution. The AMQP protocol and AMQ model is published as an open standard, allowing anyone to implement their own client or server. Today, there exists a plethora of different implementations in many different programming languages.

AMQP is generally used over TCP/IP. The protocol itself has a specific frame format formally defined in its specification. Each frame is associated with a specific frame type depending on the intent of the frame. For example, body frames only carry user data, while method frames indicates to the receiving peer that it should execute some action on its internal state. Frames may contain multiple levels of nested data structures, indicating various arguments and options. When receiving a frame, the AMQ model describes what actions the receiving peer should perform.

The research question will be answered by testing five different AMQP implementations with regards to how they handle different aspects of the AMQP protocol and AMQ model.

Chapter 3

Methodology

In this chapter, the used methodology for testing the various AMQP implementations is presented. First, the AMQPTester software tester is presented, along with its architecture and a detailed description of the implementation.

Then, a detailed description of each test case is presented, along with a motivation and references to both the formal protocol definition and AMQ model. Each test case includes a list of requirements that must be met for the testing oracle to consider the client-under-test to be compliant. In addition, the method used to conclude a verdict will be presented for each test case.

3.1 AMQPTester

In order to test different AMQP implementations, an open source software tester *AMQPTester* [6] was implemented in the Java programming as part of this thesis.

The tester uses the Java NIO API in order to implement its event based architecture. The Java NIO API was introduced with the release of J2SE 1.4 and is included in Java by default [26, pp. 264], making AMQPTester independent of any external libraries.

AMQPTester implements AMQP from scratch. The tester acts much like a simplified AMQP broker; it accepts connections from AMQP clients and performs tests by sending and receiving data while observing the responses of each connected peer.

The tester does not implement all of AMQP nor even most of the AMQ

model. However, for a client connecting in order to do a specific set of tests, AMQPTester will appear as any other AMQP broker.

The tester was designed from the ground up to be flexible in how it implements AMQP; it supports extensive protocol fuzzing and allows for most parameters and frame arguments to be arbitrarily set.

3.1.1 Architecture overview

A simplified overview of the AMQPTester architecture is provided in Figure 3.1. The *Server* Java class contains the main method which is the code entry point. Once AMQPTester starts, it sets up a Java *NIO Selector*, which is responsible for creating and listening to a TCP socket. AMQPTester then calls the blocking `select()` method which only returns upon some predefined event, making AMQPTester event-driven.

Generally, `select()` returns when some data has been received or delivered to the other peer. In AMQPTester, it also returns periodically in order to allow for the test case to execute periodical events such as sending messages in regular intervals. Currently, it returns once every second, allowing for test cases to have their respective `periodical()` method called.

Once the `Server` class accepts an incoming client connection, it creates a new `AMQPConnection` object and associated with the connection. The `AMQPConnection` object is responsible for holding incoming and outgoing TCP byte buffers which are used to store incoming and outgoing TCP data. These objects lives for as long as the TCP connection is kept alive.

The `AMQPConnection` object holds a member object of type `AMQPTester` which is instantiated at the same time as `AMQPConnection` is created. The `AMQPTester` class is, in turn, extended by other specific test cases. Which of these is being instantiated inside `AMQPConnection` depends on the command line arguments passed to AMQPTester when started.

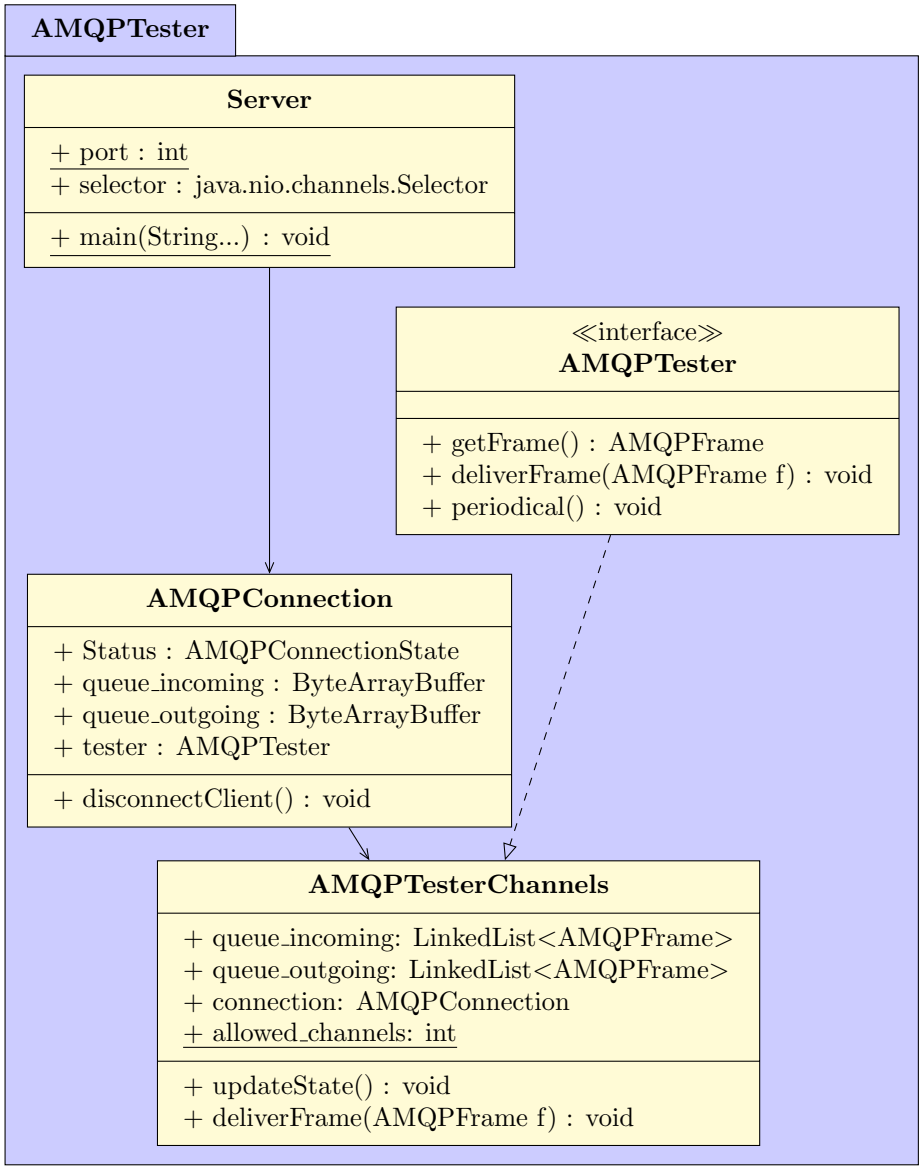


Figure 3.1: Simplified AMQPTester architecture overview

3.1.2 Sending and receiving frames

AMQPTester automatically handles encoding and decoding of AMQP wire-level frames. Hence, a programmer implementing a test case does not need to understand any of the wire-level protocol in order to write a new test case, but rather only needs to understand the AMQ model.

The instantiated test case is responsible for sending and receiving AMQP

frames and implementing the logic for doing so. The default implementation of the `AMQPTester` class holds two queues which handles incoming and outgoing AMQP frames.

When a frame is received, it is automatically decoded by the `AMQPConnection` class and put into the incoming queue, represented as a Java object rather than the actual wire-level AMQP frame. When this occurs, a callback is made to the test case, notifying it of the newly received frame.

The queued Java object representing the incoming AMQP frame is an instance of the `AMQPFrame` class. From this class, all properties related to the received AMQP frame can be extracted from its various members. This allows the test case to easily extract only the information it is interested in, such as the frame class/method ID, channel, inner frame or similar data.

If needed, the frame queues can be disabled or altered by overriding the `getFrame()` and `deliverFrame(AMQPFrame)` methods in the test case.

Sending frames works in a similar manner; the test case programmatically calls a static method `build()` on any of the `AMQPFrame` inherited classes in order to create a frame and populate it with data and arguments. For example, in order to create a Java object representing an AMQP body frame, `AMQPBodyFrame.build(channel, payload)` would be called, returning an instance of an `AMQPBodyFrame` object.

The same design logic applies for all other AMQP frame types. To create nested frame types (such as method frames), it is necessary to separately create the inner and outer frame, and then join the inner frame with the outer frame. This also applies to inner frame arguments and similar data structures.

Each Java class representing an AMQP frame is a subclass of `AMQPFrame`. The complexity of each of these classes varies depending on the complexity of their corresponding AMQP frame structure. `AMQPMethodFrame` is by far the most complex class, while `AMQPBodyFrame` is fairly simple. The class structure is presented in Figure 3.2.

Once the Java frame object has been programmatically created, it is put in the outgoing queue, after which it will be de-queued, encoded into a byte-level frame and sent over the TCP connection by the `AMQPConnection` and `Server` classes. This happens automatically.

Both the sending and receiving queues are strictly FIFO (First In, First Out) and hence maintain the correct order of frame objects to be sent, which is mandatory for AMQP to function correctly.

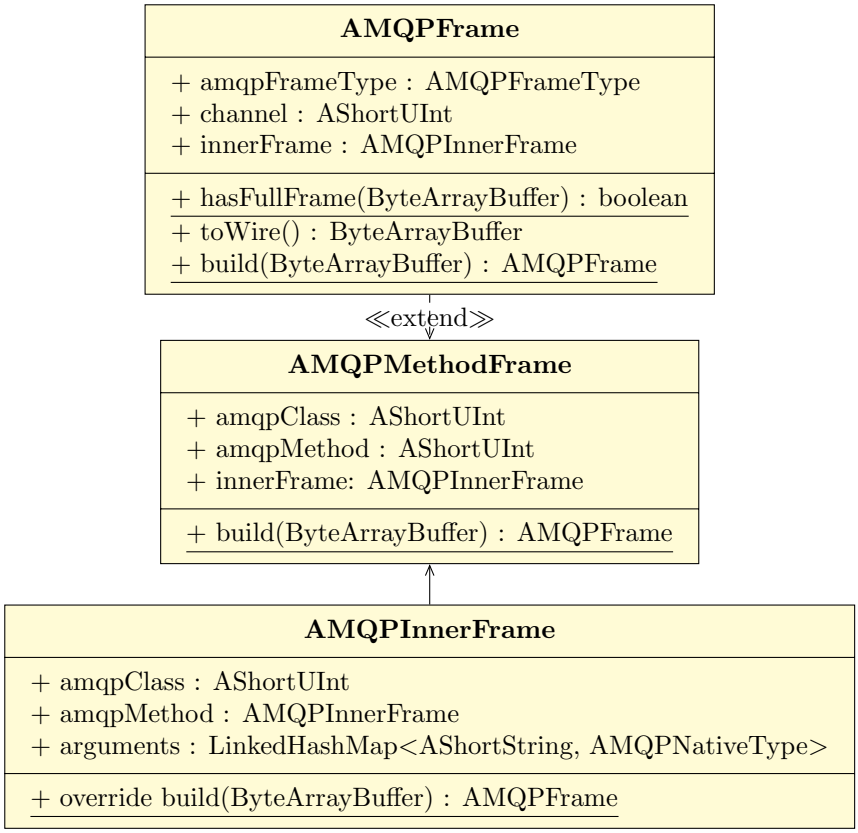


Figure 3.2: Simplified view of the `AMQPMethodFrame` class hierarchy

3.1.3 Recursive frame encoding

The four different `AMQPFrame` subclasses contains different sets of member variables used to hold all relevant data for every respective frame type.

For example, `AMQPBodyFrame` contains an `ALongString`, which is a specific data type used in AMQPTester to represent a long string. This object is used to store the payload of the body frame.

In turn, the Java class representing a method frame contains an inner frame object, which itself contains a list of arguments. This list of arguments is made up by the superclass `AMQPNativeType`, from which all other AMQP data types are inherited.

All of these AMQP data type classes (including the frame classes) implements a `toWire()` method which will recursively encode it and any held member objects into data suitable for transmitting over the TCP connection.

For example, consider an object `AMQPMethodFrame`. This object will hold a member representing the inner frame, which itself will hold other members representing its class/method ID and argument list. The argument list, in turn, will hold members representing each argument.

When calling `toWire()` on the `AMQPMethodFrame` object, a recursive call is performed. Each member of the `AMQPMethodFrame` will have their respective `toWire()` called. The resulting wire-level protocol data will be put in a buffer.

This creates a "tree of recursion" where any nested AMQP data types such as frames, field tables or arrays will create a branch, while a non-nested data types are seen as leaf nodes.

Once all the leaf nodes in the recursion tree has been called, the final buffer containing the complete method frame is returned as a byte array buffer to the callee.[1]

It should be noted that the order of which the `toWire()` calls are performed are of imperative importance. Should the calls be made in the wrong order, the resulting byte buffer will not be a valid AMQP data unit.

An example of the recursive encoding process is depicted in figure 3.3. The resulting byte buffer would be a complete wire-level AMQP frame string, suitable for transmitting over the TCP connection.

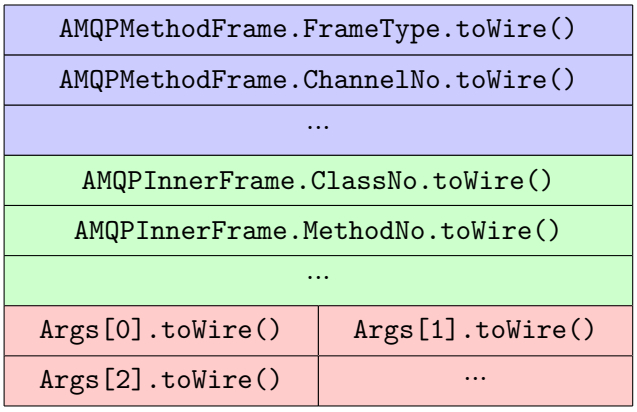


Figure 3.3: Recursive encoding in AMQPTester

[1] A `ByteArrayBuffer` is an AMQPTester specific data type, but it may be viewed as an ordinary Java `byte[]` for the sake of simplicity.

3.1.4 Recursive frame decoding

The process of receiving and decoding frame objects works much the same, but in reverse order. Each frame class and data type class has a constructor that accepts a byte array buffer from which it will populate its own members recursively.

The byte array buffer may contain more data than the individual data type, in which case the relevant data is popped from the beginning of the buffer, leaving any following data untouched.

Since the frame byte buffer is passed by reference, the same buffer can conveniently be passed from constructor to constructor, which in the end will render a complete AMQP object together with all its members correctly populated.

In AMQPTester, TCP data is read into a `ByteArrayBuffer`. Once this buffer has enough data to fully decode one or more frames, it is passed into a static polymorphic method `AMQPFrame.build(ByteArrayBuffer)`. This method determines which frame type is held within the `ByteArrayBuffer` and calls its respective constructur, returning a complete frame object which is put into the test case queue.

In order to determine if a `ByteArrayBuffer` has enough data to decode a frame, the static method `AMQPFrame.hasFullFrame(ByteArrayBuffer)` is used.

Should there be any any error when decoding a frame, a Java exception is thrown. These errors may include invalid data (such as an invalid frame type), invalid data lengths and similar. These checks are performed by every class representing an AMQP data type, such as integers, strings, frames, inner frames, booleans, etc. Each class is responsible for decoding its own data from the `ByteArrayBuffer` and throwing an error if it is unable to do so.

Test case implementations do not have to anticipate limits such as filling up the operative systems TCP buffer. AMQPTester guarantees that a queued outgoing frame will indeed be transmitted over the TCP connection, given that the connection itself is not closed.

Low level TCP access is made available to the test case, in the unlikely event that special data needs to be transmitted such that AMQPTester cannot generate them. This also allows for out-of-band data to be transmitted.[2]

[2]Out-of-band data: Data transmitted outside the defined AMQP protocol framing format, either within the same TCP connection or by other means. Both peers will have to agree on the structure of this data for it in order to be successfully transmitted.

3.1.5 Test case categories

In order to test the conformance of AMQP clients, a set of tests were implemented. These are provided by default in the AMQPTester code. They are categorized as one of two classes:

1. Specification adhering tests
2. Specification incompatible tests

The first category of tests will adhere to the standard; all traffic sent to the client under test will be valid according to the AMQP standard. A perfectly adhering client will be able to process and respond to all of these tests without errors.

This category includes tests such as timeout handling and data starvation, where the server sends data, but at an extremely slow pace.

The second category of tests do not adhere to the AMQP specification. These tests may add, modify or delete information from the protocol data which is sent to the client under test.

These types of tests may also include parameters which are out of range according to the standard, but can still technically be set to a higher or lower value. For example, the specification may define that some integer argument must be within a range of 0 – 127, while the data type allows for values between 0 – 255.

Some clients may still accept data of this sort and silently ignore it or interpret it in other ways which is not in adherence with the standard. In such cases, despite being able to continue functioning, these clients are still violating the standard as they are generally supposed to throw an error message to the broker and close the channel or connection.

3.1.6 Limitation of the tester

While the tester was built to be flexible with regards to modifying the AMQP wire-level protocol, there are still some limitations in its design. In this section, the most notable ones will be explained.

AMQPTester model granularity

The aim of the AMQPTester is to create a model of the AMQP protocol by implementing various classes that each represents some minor part of the protocol. Each connected client is an object, each incoming frame is an object, each inner type of a frame is an object, etc.

This is done in order to be able to allow the tester code to easily modify each aspect of how AMQPTester communicates with its connected clients and to abstract away complexity.

The most notable missing abstraction are with regards to channels. Each `AMQPFrame` is sent on a specific channel. In AMQPTester, it is up to each implemented test case to inspect the channel number of a received frame.

An alternative solution could have been to create an "inner channel" class. This approach would however have the consequence that more than one instance of a test case would be running. More specifically, one for each channel.

As each test case generally only performs a minor verification on a very specific part of the protocol, channel abstraction was left out.

Many of the AMQPTester supplied test cases all communicate on channel zero, making channel abstraction unnecessary. Tests that do use different channels have had no need of channel abstraction.

No client support

AMQPTester cannot initialize outgoing connections to another broker, it can only accept incoming connections from other clients.

Implementing client support is out of scope for this thesis.

TLS support

The tester does not implement or use TLS. It should be noted that TLS is running below the application layer, making it unnecessary for an AMQP software tester, as the application data will be the same regardless.[3]

[3]Transport Layer Security (TLS): A protocol used to encrypt data traffic over a network. Encryption is applied transparently from the point of view of the broker/client.

Low-level incoming TCP data

AMQPTester allows for direct TCP socket access to individual test cases, making it possible for test cases to directly write data to the client under test.

However, when the client sends data to AMQPTester, an attempt is always made to directly build a Java frame object of the corresponding received data. Should this data not be decodable, there is no way for the test case to access it without modifying the underlying AMQPTester code.

Single-threaded architecture

AMQPTester is single-threaded. While this design makes it somewhat slower (especially for many concurrent connections), it was still favoured because of its simpler design. In addition, performance is more than adequate using a single thread.

Multi-threaded support can be added relatively easy, as there currently exists no communication in-between multiple connected clients.

3.2 Verifying client conformance

In order to validate the verdict of a specific test case in AMQPTester, two methods will be used. Either AMQPTester acts as a testing oracle that immediately determines the verdict, or it will provide logging facilities which will be used to manually conclude a verdict together with other (possibly manually collected) information.

The latter method will be used in cases where it is not possible to automatically generate a verdict. Such cases may include inspecting the state of the client under test, verifying conformity by receiving test data via its API or dumping and inspecting protocol data using Wireshark or other network inspection tools.

The need for manual inspection arises for the reason that the client under test may act the same way with respect to AMQPTester, regardless if it passes or fails a test case.

In most AMQPTester included test cases, a verdict is automatically reached.

3.3 Test case implementations

This section gives a technical explanation of the implemented test cases in AMQPTester, including a motivation and the formal requirements for a passing verdict in each test case. The corresponding Java source code can be found in [6].

Each section below also explains how test data is generated.

3.3.1 Channel tester

The channel test case evaluates that the client under test adheres to restrictions put upon the maximum number of allowed channels per TCP connection.

During the connection phase, the AMQP broker and the client negotiates the maximum number of allowed open channels. Once this number has been negotiated, the client should never attempt to open up more channels.

The AMQ model describes how the maximum number of channels are negotiated over the wire. The most important parts of this is depicted in Figure 3.4. All of these frames are synchronized, indicating that they require a response before any other (synchronized) frames are transmitted. It should be noted that some irrelevant arguments and frames have been left out for simplicity in Figure 3.4.

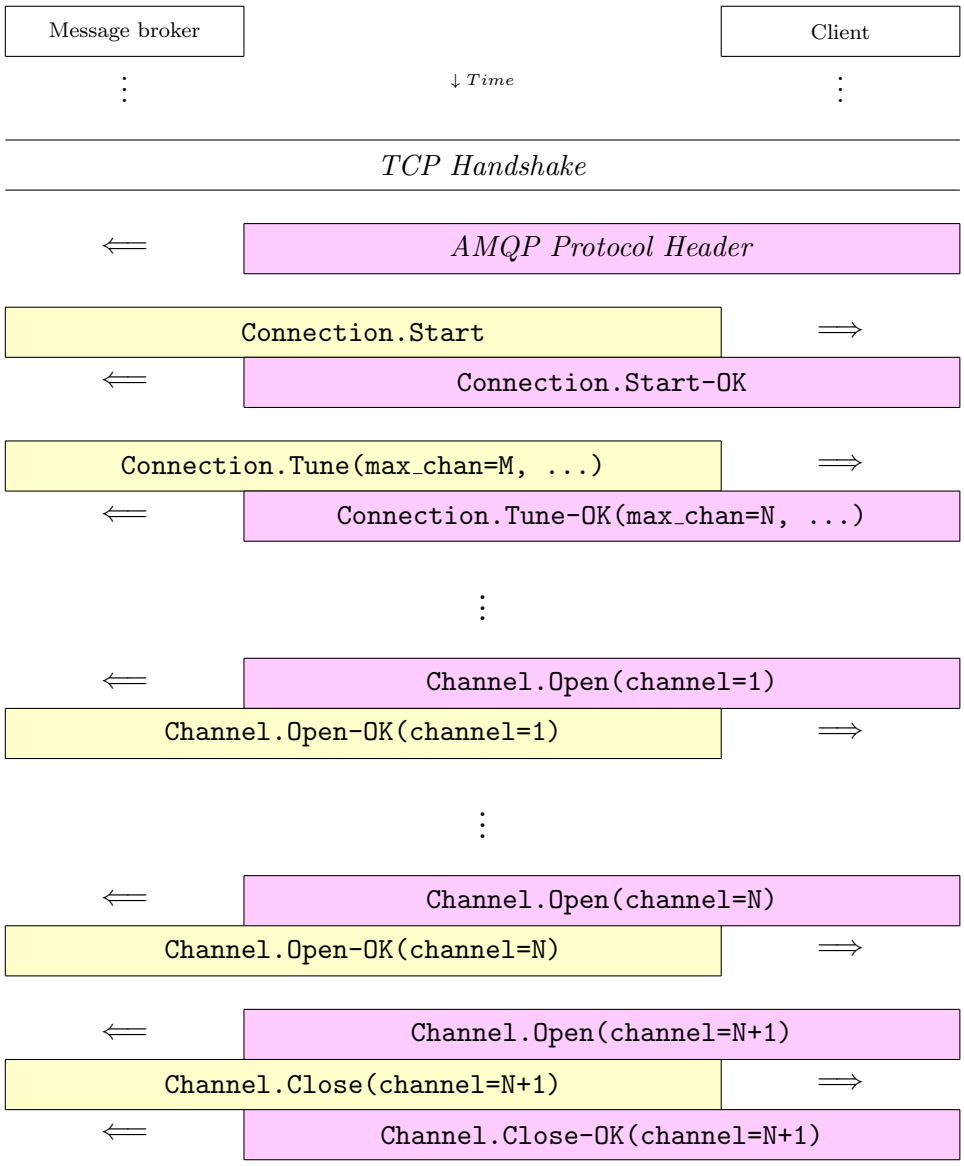


Figure 3.4: Channels test flow

In the `Connection.Tune` frame, the broker specifies the maximum number of channels M, which is the highest possible numbers of channels it will ever allow. If $M = 0$, the broker has no restriction in the number of possible channels.[4]

The client replies with `Connection.Tune-OK`, including a number N of max-

[4]It should be noted that the channel number is specified by a 16-bit integer, hence the maximum possible number of channels over one connection will be limited to $2^{16} - 1 = 65535$.

imum allowed channels such that $1 \leq N \leq M$ (given that $M \neq 0$) and includes N as an argument to the server. Should $M = 0$, the client is free to pick any N it desires given the 16 bit integer bound restriction.

After this negotiation, N is the maximum number of channels agreed upon by both the broker and client.

This test case verifies the following aspects of the client under test:

1. A client should never request more than N channels.
2. A client should never reply with an $N > M$ given $M \neq 0$.
3. Given $M = 0$, the client should never reply with $N = 0$.
4. Given $M = 0$, the number of maximum channels chosen by the client will be observed.[5]

Regarding point 3, the AMQP specification makes no mention that a client choosing $N = 0$ would suggest to the broker that any number of channels are allowed.

The specification does however not forbid a client from sending $N = 0$ so long as the broker previously sent $M = 0$. This would be a problem, as the negotiated maximum number of channels would be zero, making it impossible to open any new channels and transmitting any application data.

Therefore, the client would be fundamentally broken in such cases. It is hence a relevant scenario to test for, even if it is formally allowed by the specification. It should be noted that this is a corner case, and its sole purpose is to detect whether a client AMQP implementation simply copies the M data directly to N without interpreting it.

This test case automatically reaches a verdict. This is done by having the client under test automatically opening a large number of channels within a for-loop. AMQPTester will negotiate a lower number of channels than the client under test will attempt to open. A compliant client should throw an error via its API (or similar method), indicating that it is not allowed to open any further channels.

3.3.2 Data type compliance

The AMQP specification defines [5, p. 22] many different data types and how to encode them over the wire, amongst others:

[5]This is useful because different client libraries may request a different amount of maximum channels, potentially introducing bugs into the larger application.

1. Integers, 1–8 bytes long.
2. Bits/booleans, encoded into bytes.
3. Short and long strings.
4. Field tables, containing key-pair associative values.

The AMQP specification is ambiguous in defining which data types may and may not be transmitted over the wire. In the beginning of the specification [5, p. 22], only the above data types are formally allowed to be sent within method frames.

However, as defined in [5, p. 31], the formal protocol grammar allows for method frames to contain other data types as well. Following this grammar, the below method frame would be a valid AMQP sentence, yet it would contain a `field-array` data type which is not included in the allowed data types as mentioned in the previous paragraph:

```
method-frame =
= 0x01 frame-properties method-payload frame-end =
= 0x01 frame-properties class-id method-id *amqp-field =
= 0x01 frame-properties class-id method-id field-table =
= 0x01 frame-properties class-id method-id 'A' field-array
```

The same ambiguity holds true for other data types as well, such as the "null data type" (as specified by the upper case 'V' over the wire), which contains no data at all.

Another protocol ambiguity, which also mentioned in the RabbitMQ Errata [27], is the definition of the `field-array` data type. In all of the AMQP specification, it is only defined within the formal grammar as such:

```
field-array = long-int *field-value
```

While `*field-value` is well-defined, it is unclear whether the `long-int` specifies the number of bytes, the number of elements in `*field-value`, or something else.

As such, different AMQP implementations may incorrectly interpret this data type and enter into a state of protocol miss-alignment, causing the broker/client state machines to behave in an undefined manner.

It should be noted that the RabbitMQ broker assumes that the `long-int` is the total length in bytes of the `*field-value` data. This arguably makes the most sense, as it is how all other data type lengths are implemented in AMQP.

As such, AMQPTester implements `field-array` in the same way as RabbitMQ.

The goal of the data type compliance test case are as follows:

1. Validate the decoding process of well defined data types.
2. Compare ambiguous data types behaviour in different client implementations, such as `field-array`.
3. Verify that nesting works to a sufficient large depth without getting any stack overflows or similar client errors.[6]
4. Validate that the client under test supports reading multi-byte UTF-8 characters.[7]
5. Validate that the client under test does throw an error if it receives a data type which is not defined within AMQP.

The test case works by transmitting a multitude of different data types to the client under test. Should the client be unable to decode one or more of these data types, a protocol alignment error will occur and the client will not be able to continue operating. Should this happen, the test will be re-run with a subset of all possible data types, narrowing down which data types the client under test are unable to decode.

[6] Application data, i.e. the data that different middleware functions are processing, is never transmitted nested within `field-array`s. An external adversary will therefore have no way of inducing such data within method frames. However, it may still be relevant to know the nesting limit as application developers could accidentally create very large nested data structures.

[7] If this is not the case, a client library could be popping too much data from the buffer, ending up with an alignment error.

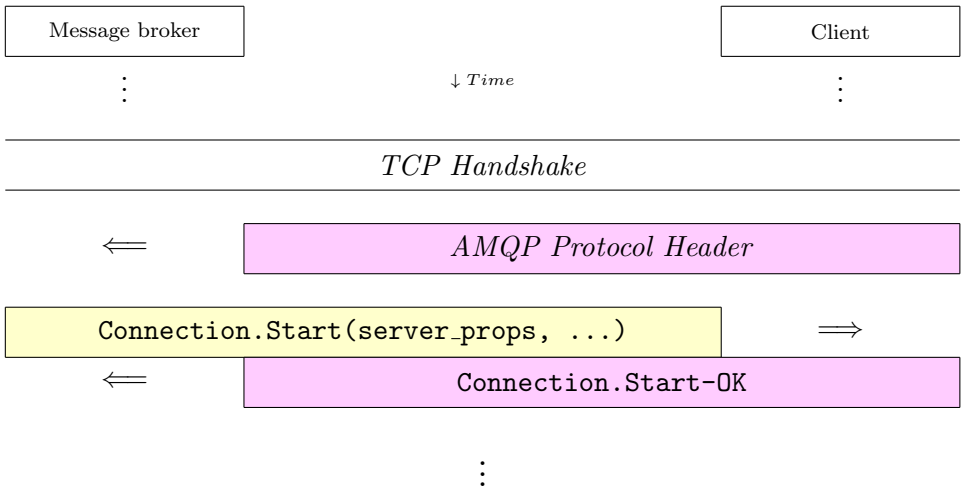


Figure 3.5: AMQP handshake with specially encoded *server_props*

During the connection initialization, as depicted in Figure 3.5, the server transmits a method frame with the class/method name `Connection.Start`. Within this frame, there exists a field-array argument `Server-Properties`. Its content is defined within the XML protocol definitions [28]:

"The properties SHOULD contain ***at least*** *these fields: 'host', specifying the server host name or address, 'product', giving the name of the server product, 'version', giving the name of the server version, 'platform', giving the name of the operating system, 'copyright', if appropriate, and 'information', giving other general information."*

The `Server-Properties` argument can hence be populated with more data than mentioned above. This is the method used by this test case in order to send arbitrary data to the client under test. Should the client be unable to decode any of these fields, it will be unable to continue processing further frames from AMQPTester.

Therefore, it is trivial to automatically determine a verdict, as the test case only needs to transmit and acknowledge any synchronous frame after transmitting the `Connection.Start` frame, to make sure that the client under test is still aligned with the protocol. More specifically, this test case transmits `Connection.Tune` and will give a passing verdict if it receives a `Connection.Tune-OK` frame.

This implies a passing verdict if the client under test passes these criteria:

1. The client completes the handshake and properly decodes all data types, and

2. No error occurs when decoding deeply nested `field-array`s, and
3. The client correctly decodes multi-byte UTF-8 characters, and
4. The client does throw an error when invalid datatypes are being transmitted by the broker (For non-adhering tests only).

In such cases where a client under test fails to decode one or more data types, a manual inspection of the AMQP protocol traffic will be conducted via Wireshark. This is to guarantee that AMQPTester does indeed encode data correctly.

3.3.3 Heartbeat compliance

Heartbeats are an AMQP feature intended to detect stale connections. While many layer 4 protocols (e.g. TCP) already supports similar features, the timeout tends to be very long (in the order of hours or days) before a broken connection is torn down by the TCP stack.

To address this issue, the AMQ model defines how brokers and clients should transmit heartbeats in order to make sure the other peer is still alive and responsive. Not only does AMQP heartbeats solve the problem of long lived unresponsive TCP connections, but it could theoretically also detect logical errors in the other peer such as if it has reached an infinite loop.

In such cases, the TCP connection would still be considered alive as TCP heartbeats would be handled by the operative system which is not affected by such a user-space logical error. An AMQP heartbeat would, in this case, most likely not be replied to since the code is stuck in a state it cannot recover from.

Heartbeat behaviour is negotiated between the broker and client during the connection initialization within the $Connection.Tune[-OK]$ frames. Both the broker and the client sends an argument containing a heartbeat period, as depicted in Figure 3.6.

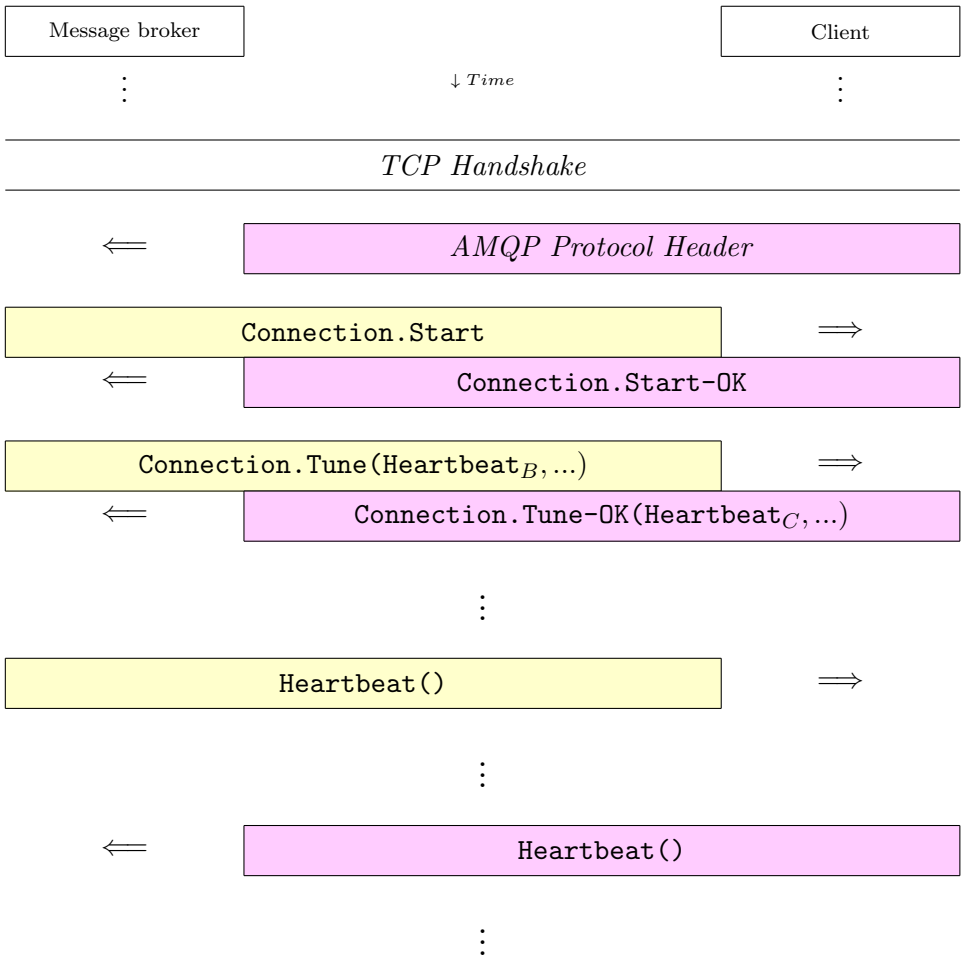


Figure 3.6: Heartbeat negotiation and transmission

The two heartbeat periods $Heartbeat_B$ and $Heartbeat_C$ indicates to the other peer that they should send a heartbeat frame as often as requested. Both of these values are defined in seconds.

For example, if the broker requests $Heartbeat_B = 5$, the client is supposed to send a heartbeat frame every 5 seconds. Vice versa, if $Heartbeat_C = 10$, the broker is supposed to send a heartbeat frame every 10 seconds.

If the requested heartbeat period is zero, the sending peer indicates that it does not require heartbeats to be transmitted. In such case, the AMQ model does however allow the receiver of such a heartbeat period to send heartbeats anyway.

Heartbeat frames are a special type of frame in AMQP and are not embedded within method frames. They have their own frame type ID just like any other

AMQP frame and are arguably the simplest frame type within AMQP. A typical heartbeat frame is depicted in Figure 3.7. The frame `type` value is denoted {4,8} because the specification is ambiguous as to which value should be used [5, p. 33 and p. 34].

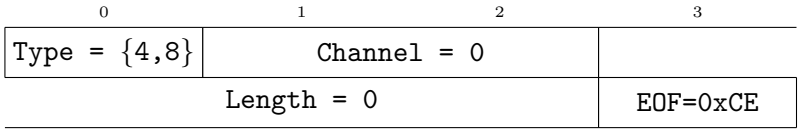


Figure 3.7: Typical heartbeat frame

In addition, the protocol grammar definition does not follow the general AMQP frame format, but rather suggests using a format [5, p. 32] containing only 1 byte per frame field, as depicted in Figure 3.8.

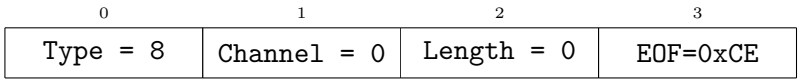


Figure 3.8: AMQP protocol grammar definition of heartbeat frames

RabbitMQ and other popular brokers does [27] use the format as presented in figure 3.7. AMQPTester natively implements heartbeat frames in this format as well.

In addition to the dedicated heartbeat frame, any received frame counts as a heartbeat substitute. This design allows for the protocol to be less chatty and more compact, as any received frame indicates that the other peer is still alive.

The AMQ model defines heartbeat implementation guidelines in [5, p. 36]. For this specific test case, the relevant guidelines are as follows:

1. Heartbeats must be sent on channel 0. If another channel number is used, the receiving peer has to raise a `Connection` exception with error code 501 and then terminate the connection.
2. If a peer does not implement heartbeat functionality, it should silently ignore any received heartbeats.
3. Peers should make best efforts in order to transmit heartbeats at the requested intervals, but both peers are allowed to transmit them at any time.
4. If two or more heartbeats have been missed, the TCP connection should immediately be closed, without sending any further frames.

A passing verdict will be given if the client under test it fulfills all these criteria:

1. Heartbeat support is implemented.
2. Heartbeats are sent periodically such that the broker will not miss more than 2 heartbeats.
3. Heartbeats received on any channel other than zero raise an 501 exception and disconnect from AMQPTester.

In addition, it will be noted if any client follows a heartbeat syntax other than suggested by RabbitMQ, as this may have significant impact on compatibility between different implementations.

3.3.4 Message delivery fuzzing

This test case validates multiple aspects of transmitting application data (i.e. actual user data) the same way a broker would deliver data to an AMQP client. Common to all aspects is that the client under test will connect to AMQPTester and request two additional channels from which it will receive application data. Using these two channels, a multitude of different tests will be performed.

The initial connection procedure is depicted in Figure 3.9. Some irrelevant frames have been left out from the figure for simplicity, such as the frames needed to declare the queues before they can be consumed. In addition, all channel numbers are transmitted in the frame channel field, although they are depicted as arguments.

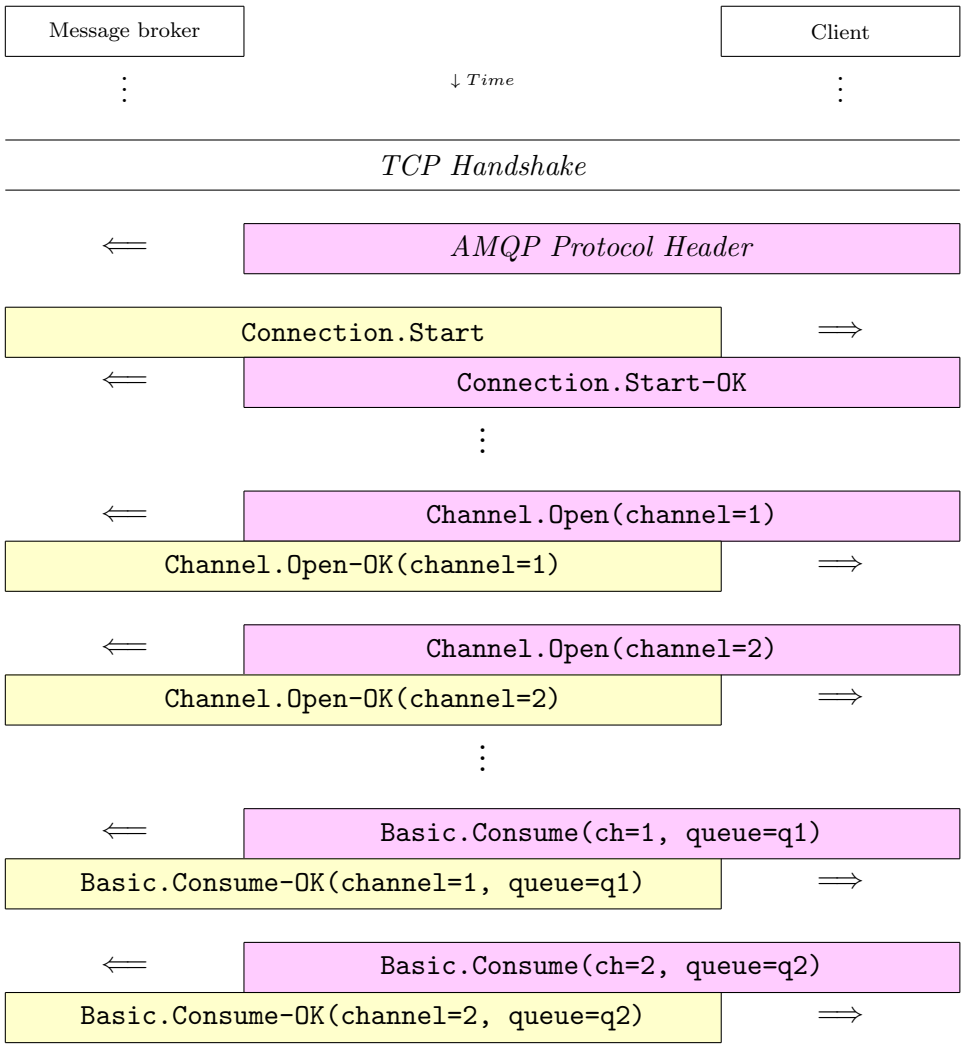


Figure 3.9: Opening two channels for data transfer

The `Basic-Consume` method frame indicates to the broker that the client under test is subscribing to receive application data from the queues *q1* and *q2* on channels 1 and 2 respectively. After these frames have been acknowledged, the client will be listening for incoming application data.

Once the broker has some application data available to be delivered to the client, the broker notifies it using a method frame `Basic-Deliver`. This method frame contains (amongst others) information about which queue data is about to be delivered from, if the message is being re-transmitted because of previous failure and from which exchange the message was sent.

After sending the `Basic-Deliver` frame, the broker immediately sends a

header frame. The header frame contains information about the application data that is about to be transmitted, such as the length and content type.

Each header frame contains a class ID corresponding to the class ID used in the previously sent method frame. The AMQP specification demands that both these class IDs are set to the same value. For example, if transmitting a method frame `Basic-Deliver`, the following header frame class ID must be `Basic`.

The header frame encodes property flags together with a property list, which may encode additional data. In this test case, these property flags are zeroed out as they are irrelevant for the test itself.

After transmitting the header frame, the broker transmits the application data within one or more body frames. These frames may vary in size, but may not exceed the size as negotiated by `Connection-Tune` during connection initialization. As body frames does not contain any sequence number, these must delivered in the correct order.

This test case validates multiple aspects of transmitting message payloads to the client. The different methods used to determine the verdict of a client under test will be discussed each in their own subsection below.

Aborted messages

When the broker is transmitting body frames, sending any other frame type on the same channel aborts the transmission of the application data completely [5, p. 35]. This allows the broker to cancel an in-transit message from being delivered.

This part of this test case validates that the client under test supports aborted messages. In order to pass this part of the test, the client under test must accept an aborted message transfer without occurring a protocol alignment error and without any other type of error messages. Throwing a specific aborted message error via its API is acceptable, given that this behaviour is documented.

Message multiplexing

Multiple messages can be transmitted over two or more channels at the same time within the same TCP connection. This is depicted in Figure 3.10, where AMQPTester first indicates a message delivery on channel 1, then directly after on channel 2. Thereafter, the messages are multiplexed over the two

channels at the same time. In Figure 3.10, frames are coloured based on the channel they are being transmitted for viewing convenience.

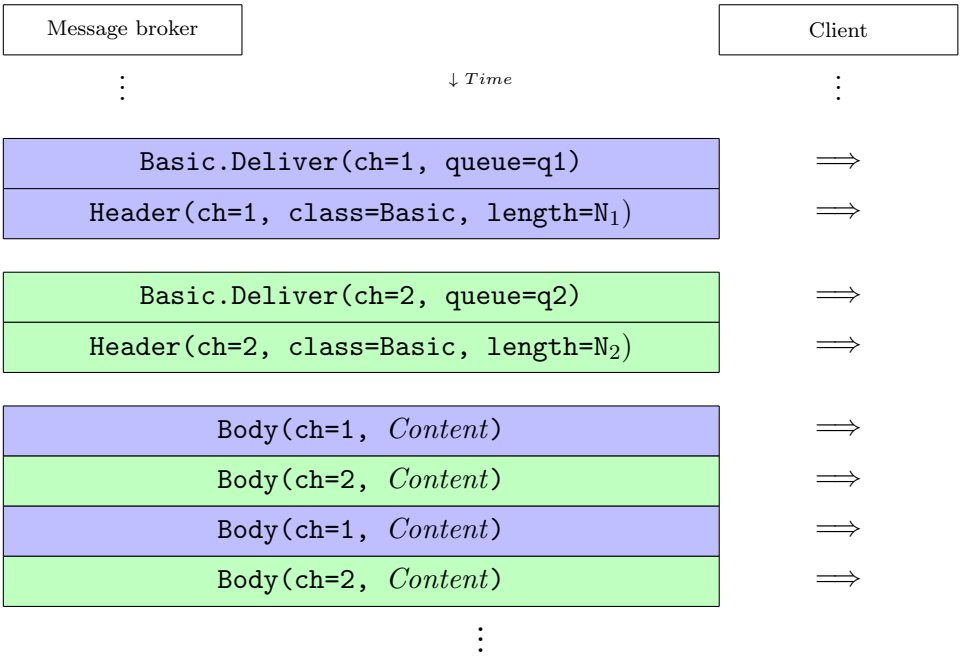


Figure 3.10: Multiplexing data over two channels on the same TCP connection

RabbitMQ does not multiplex application data like so, but rather transmits one message at a time [7]. While this behaviour does not violate the AMQP protocol nor the AMQ model, this may have the side effect that some client libraries have not been properly tested for multiplexing, given that they were developed and tested against RabbitMQ only.

The transmission technique used by RabbitMQ does comply with the AMQP specification. It may be more memory efficient, as it allows the client to only cache one message at a time in memory. Possible negative effect with the approach could be that smaller messages have to wait a longer time before they reach an application function, if there is a larger message in-transit which is blocking the connection.

This part of the test case validates that the client under test does indeed support data multiplexing. A passing verdict is given if the application data messages are fully delivered and correctly consumed regardless of which multiplexing method is being used.

The verdict will be given manually by writing a small program which opens

two channels as described above. Compliance cannot be automatically validated by AMQPTester, as it has no way of knowing whether the application data was properly delivered or not.

Slow delivery

This part of the test case will combine the above multiplexing technique with slow frame delivery. More specifically, AMQPTester will deliver frames in a slow manner to the client under test. 10 bytes will be written to the TCP socket every 100 millisecond.

This will cause the brokers TCP stack to send a single AMQP frame over multiple TCP packets, having the effect of the clients TCP queue to contain partially complete frames. The reason for doing so is to validate the client under tests ability to wait for (at least) one complete frame to be received before decoding is attempted.

The verdict will be given manually by observing the behaviour of the client under test. A pass will be given if it consumes messages and delivers them in the same way as with without slow TCP delivery.

The TCP connection will be manually inspected in Wireshark to confirm that AMQP frames are indeed sent over two or more TCP packets.

Invalid body size

In this part of the test case, the client under test will be sent a method frame `Basic-Deliver` along with a header frame indicating that it will receive N bytes of data. Instead of transmitting N bytes of application data, $N + i$ bytes will be delivered for some integer $i > 1$.

According to the protocol guidelines [5, p. 36], a client receiving an incomplete or badly formatted frame must raise a connection exception with the code 505 and close the connection.

This part of the test case combines the above described slow delivery as well as regular delivery. As each TCP packet may contain more than one AMQP body frame, this has the effect that the client under test may not reach its socket `select()` function during fast delivery, because multiple AMQP body frames are delivered atomically. This, in turn, may trigger different results.

This part of the test is given a manual verdict by inspecting the messages delivered via the clients API. A pass will be given if (and only if) the client

under test reports some kind of error, such as a programmatic exception.

Delivery on invalid channel

AMQPTester will attempt to deliver content on invalid channel numbers. More specifically, channel zero (which should never have any payload sent over it) and channels which have not been opened by the client will receive message payloads. In both cases, data will be transmitted by AMQPTester after first having the client under test opening and subscribing on two channels.

In both cases, the client under test should trigger a connection exception and disconnect from AMQPTester [5, p. 36]. This aspect of the test case is automatically given a verdict by AMQPTester. A passing verdict is given if a connection exception is raised and the connection is closed.

Badly formatted header frames

AMQPTester will transmit header frames containing wrongly selected class IDs and multiple chained property flags.

The chained property flags will consist of multiple property flags, all zeroed out, indicating no set options at all. Each LSB will be set except for the last one, indicating that more flag indicators are to follow.

As header frames do not contain more than 15 different flags, there is no logical reason to ever transmit more than the 2 bytes of flag indicators. As such, some AMQP implementations may have hard-coded the flag length and will therefore not adhere to the AMQP protocol.

A test verdict will automatically be given. In the case of chained property flags, a passing verdict will be given if the client under test does not run into any protocol alignment issues. This will be verified by transmitting a synchronous frame after sending a chained property header frame. Receiving an acknowledgement will be considered a pass.

In the case of transmitting a wrongly selected header frame ID, a pass will be given to the clients that closes the connection.

3.3.5 Mandatory routing

Sending application data to a broker works in a similar manner as to receiving data. The client opens a channel and sends a `Basic-Publish` method frame to the broker. Directly following this method frame, it sends a header and one (or more) body frames, much similar to the way a broker delivers messages to a client. The header and body frame(s) follow the same format as when application data is being received. This is depicted in Figure 3.11.

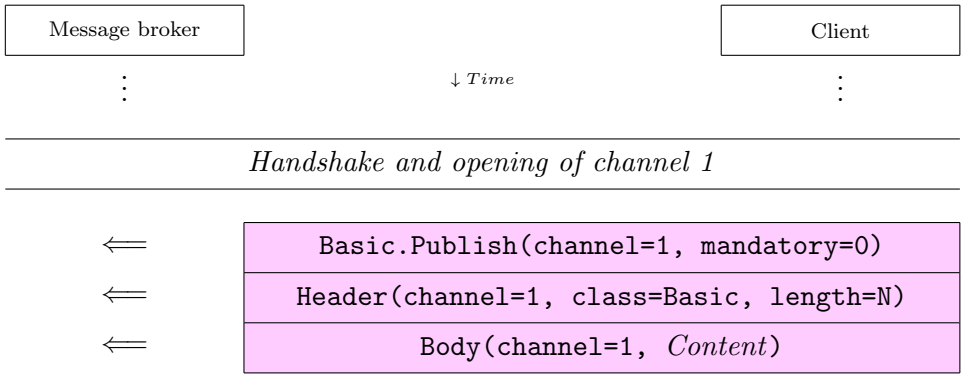


Figure 3.11: A client publishing data to the broker

The *mandatory* boolean option (which is a property flag) indicates whether the broker should report delivery errors to the client.[8] Is the flag zeroed out, any failed messages will be silently dropped by the broker.

When the *mandatory* flag is set and a message delivery occurs, the broker will reply with a `Basic-Return` frame directly followed by the header and body frame(s) of the failed message. This is depicted in Figure 3.12.

[8]Formally, a delivery error occurs when the exchange has failed to put the message into any queues immediately. Without the flag being set, the message is queued until a suitable exchange is found in the future.

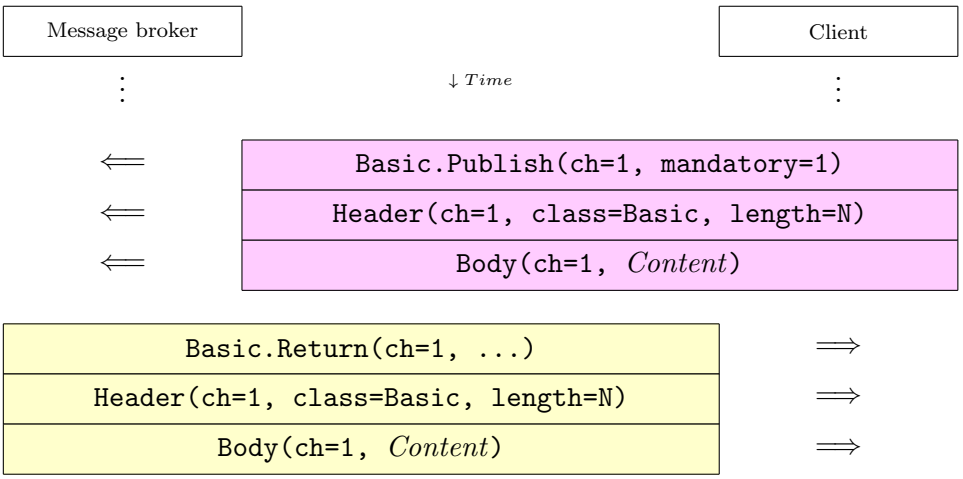


Figure 3.12: Message delivery failure

The `Basic-Return` frame contains an error code describing the failure, along with a human-readable message which can be used for debugging and logging purposes.

The purpose of this test case is to verify that the client under test does indeed handle failed message deliveries. According to [7], this is a common problem in many AMQP implementations.

The client under test will attempt to transmit a message with the mandatory flag set. AMQPTester will decode its three associated frames and will later return them using the `Basic-Return` method.

A verdict will be manually given, as AMQPTester has no reliable way of detecting whether a rejected message has been accepted back by the client under test. As such, a pass will be given if the client under test supports setting the *mandatory* flag and correctly returns a rejected message via its API or similar well-documented method.

3.4 Summary

In order to put five different clients under various tests with regard to the AMQP protocol and the AMQ model, a software tester AMQPTester was implemented. The tester acts as a regular AMQP broker, allowing clients to connect to it. A test case is implemented by extending a Java class. When a client connects to the software tester, an instance of the Java class is created. When data is being received from the client, it is automatically

decoded and fed to the Java test case class as a Java object.

Sending AMQP traffic works much the same but in reverse. The test case Java class programatically creates a Java object and populates it with the necessary parameters, after which AMQPTester will encode it and transfer it to the client under test.

In this chapter, five different test cases are presented, along with a technical description and motivation as to why they were implemented. In addition, each test case has a formal list of criteria which needs to be met in order for a client under test to be considered compliant.

The five different test cases will be used against the five previously presented AMQP libraries in order to answer the research question.

Chapter 4

Results

In this chapter, an overview of the results is presented, followed by a detailed description of the behaviour of each AMQP library under each test case. In addition, any other discovered non-standard behavior is also presented in Section 4.2.

A discovered arbitrary data injection attack is also presented in Section 4.3.

4.1 Result overview

An overview of the results is presented in Table 4.1. A checkmark (✓) denotes a passing verdict, meaning that the client under test passed all criteria listed in Section 3.3.

A checkmark with an exclamation mark ($\checkmark^{!}$) denotes a formally failed verdict. However, the failure were such that only some minor parts of the criteria listed in Section 3.3 were violated, not in such a way that it will have severe impact of the functionality or security of the library. A detailed description of these minor violations are presented in Section 4.2.

A crossmark (✗) denotes a failed verdict, such that the client under test is either not adhering to the AMQP standard or will be unable to continue communicating with the broker due to a severe protocol error. In addition, this grade is given to implementations that lack the tested functionality (such as heartbeats).

	PHP-amqplib	AMQP.Node	Py-AMQP	Rabbitmq-C	RabbitMQ-Java
Channels	✓	✓!	✗	✗	✓!
Data Types	✓!	✓!	✓	✓!	✓!
Heartbeat	✗	✓!	✗	✗	✓
Message Delivery Fuzzing	✓!	✓!	✓!	✗	✓!
Mandatory Routing	✓	✓!	✗	✓	✓

Table 4.1: Test case result overview

4.2 Detailed results per library

In this section, the observed results on a per-library basis are presented.

All libraries correctly decoded multibyte UTF-8 characters. The length indicator of `field-array`s were all interpreted as the full length of the array (i.e. following the RabbitMQ convention), not the number of elements. These subjects will therefore not be mentioned individually.

Any aspect where the client under test adheres to the specification will not be mentioned.

4.2.1 PHP-amqplib

Channel negotiation and limitation

The library fully supports negotiating and restricting its number of concurrent open channels. It is possible to manually open a specific channel number via the API, but the library will never allow more than the negotiated number of channels to be open at the same time.

If the server has no upper channel limit, PHP-amqplib negotiates the maximum number of channels possible ($2^{16} - 1 = 65535$).

Data type compliance

Deep nesting is fully supported. All tested data types were supported, excluding `long uint` and `short-string`.

Heartbeat compliance

Heartbeats are not supported and not negotiated. Heartbeat frames sent by the server using the RabbitMQ format are silently dropped, regardless of which channel they are delivered on.

Message delivery fuzzing

The library supports multiplexing of application data over multiple channels simultaneously. Slow delivery of frames functions adequately.

It does not support aborted transfers, but rather throws a programmatic exception and then immediately disconnects from the broker.

The library was observed to accept more than the declared number of bytes of application data. If a header frame was sent specifying that N bytes of data is to be delivered, it was possible to send $N+1$ bytes without any error being reported.

This only holds true when the last body frame delivers the overflow of data. If $\geq N$ bytes are delivered and an additional body frame is delivered, an error is thrown.

Mandatory routing

Full support for mandatory routing and rejecting messages were observed. Rejected messages are delivered via an API callback.

4.2.2 AMQP.Node

Channel negotiation and limitation

The library does correctly limit the number of negotiated channels. However, if the broker sets the maximum channels parameter to zero, AMQP.Node also responds with zero. As explained in 3.3.1, this behaviour would negotiate the maximum number of allowed open channels to zero.

An AMQP 0-9-1 compliant broker would therefore disconnect the client when it attempts to open a channel for application data transport, making it impossible to transfer any application data.

Data type compliance

Deep nesting functions as expected. The most common data types are implemented, with the notable exception of `short-string`, `short-uint` and `long-uint`. This applies to both `field-array`s and `field-table`s, as the same code is used to decode both.

Heartbeat compliance

Heartbeats are implemented using the RabbitMQ frame format. Support for periodical heartbeats are implemented and functional, including disconnecting after 2 missed heartbeats. Support for heartbeat substitution using other frames is functional as well.

The library does accept heartbeats on channels other than zero, which is not compliant to the formal specification.

Message delivery fuzzing

Full multiplexing support was observed, both using normal and slow delivery. The library does validate application data sizes against the value specified in the header frame.

Transmitting application data on undefined channels (including channel zero) correctly causes a programmatic exception to be thrown.

Support for aborted transfer is not implemented, but rather throws an exception when the broker attempts to aborts a transfer.[1]

Mandatory routing

Mandatory routing rejection is supported by the library. However, it was observed that option `14` is set in the `Basic-Publish` frame. This option is defined as a reserved `short-string` for future use by the specification.

[1]The exception indicates that the received frame type is of the wrong type, as the library expected a body frame whilst a method frame was received.

AMQP.Node assigns a value `0x00000000` to this option, indicating an empty `long-string` rather than a `short-string`. This caused an alignment issue in AMQPTester, but a work-around was implemented in order to evaluate the test case.

It should be noted that AMQP.Node is the only library that was seen to set this option.

4.2.3 Py-AMQP

Note: Py-AMQP was observed to send a non-standard protocol string during connection initalization. Instead of the well-defined `'AMQP' 0x00 0x00 0x09 0x01` protocol identification string, Py-AMQP sends `'AMQP' 0x01 0x01 0x09 0x09`. RabbitMQ implements support for this string. A strictly adhering broker would not be able to communicate at all with Py-AMQP. A workaround was added in AMQPTester.

Channel negotiation and limitation

During initialization, Py-AMQP allocates a bucket with all possible 65535 channels. When opening and closing channels during execution time, channels are taken and returned from this bucket. The library does not honor the channel limit as negotiated, but rather keeps sending `Channel-Open`.

As the bucket always contains the maximum number of possible channels, there is a hard limit of 65535 channels before the API refuses to attempt to open more channels.

If the broker sets the maximum number of allowed channels to zero during the initial handshake, Py-AMQP negotiates the maximum number of possible channels (65535).

Data type compliance

Full support for all tested data types were observed. Py-AMQP correctly decodes `short-string`s in `field-array`s and `field-table`s.

Heartbeat compliance

No automatic heartbeat support was observed. The library does correctly ignore heartbeats sent by the broker.

Partial heartbeat support is implemented, but requires the programmer to programmatically send these via the API.

Message delivery fuzzing

Multiplexing over two channels works as expected. Slow TCP delivery and partial frame delivery correctly yielded the same result as with normal delivery. Aborted transfers were not observed to be functional.

The library does not validate the body length of application data in the last frame, but rather delivers the longer body data via the API similar to 4.2.1.

Mandatory routing

While the library does support setting the `mandatory` flag, no callback for programmatically receiving rejected messages is available. Instead, the library throws a `struct.error` as it cannot decode the rejected message correctly.

4.2.4 Rabbitmq-C

Channel negotiation and limitation

The library does not honor the negotiated number of maximum channels. The API allows the programmer to select the channel number when opening a new channel, but the library does not do any boundary checks on the specified channel. Hence, it is possible to send `Channel-Open` on channel zero.

When having 65535 channels open and attempting to open another channel, the channel counter overflows and sends `Channel-Open` on channel zero.

The library negotiates the maximum number of channels (65535) if the broker declares no upper limit during the initial handshake.

Data type compliance

All tested data types were implemented, except `short-string`. Deep nesting is supported.

Heartbeat compliance

Heartbeats are not implemented, but the library correctly ignores any heartbeat frames sent by the broker.

Message delivery fuzzing

Multiplexing was observed to not be fully supported. When multiple body frames were delivered in the same TCP frame, only the first AMQP frame was correctly decoded. When using slow delivery (causing each TCP frame to contain at most one AMQP frame), multiplexing was observed to be working as expected.

Aborted transfers are not supported. When delivering body frames out-of-order, the library delivered an empty envelope via its API instead of triggering an error and disconnecting.[2]

Mandatory routing

Full support for the mandatory flag and message rejection was observed.

4.2.5 RabbitMQ Java Client

Channel negotiation and limitation

The maximum number of negotiated channels are honored. It should however be noted that if N is the negotiated maximum number of channels, the library only allows for channel numbers $\leq N$ to be opened.

While this does not violate the AMQP specification, the client *is* allowed to open channels $> N$, so long as the total number of open channels is $\leq N$.

If the broker indicates zero as its maximum number of allowed channels during the initial handshake, the library responds with 2047, which will then be the negotiated maximum number of channels. It should be noted that the RabbitMQ Java Client is the only library that sets this value to something else other than zero or maximum.

[2] In this case, body frames were delivered without being preceded by the mandatory method + header frames.

Data type compliance

Deep nesting is supported. All tested data types were implemented, with the notable exception of `short-string`, `short-uint` and `long-uint`.

Heartbeat compliance

The library correctly implements heartbeat support using the RabbitMQ frame format. Substitution frames is also implemented as described in the AMQP specification.

Message delivery fuzzing

Multiplexing works as expected, as does slow delivery. Aborted transfers are not implemented, but incorrectly triggers a frame type exception.[3]

The library correctly verifies the number of bytes transmitted in the application data against the value in the header frame. When transmitting body frames on an unopened channel or channel zero, the library correctly triggers an exception.

Mandatory routing

Full support for both the mandatory flag and rejected messages were observed. Rejected messages are delivered via an API callback.

4.3 Arbitrary data injection

Due to the ambiguity between RabbitMQ and the AMQP 0-9-1 specification, a data injection attack was discovered. The attack has the potential to inject arbitrary data into both `field-table`s and `field-array`s.

The attack is made possible if all the following prerequisites are met:

1. A `field-table` or `field-array` contains at least one member V of type `short-string`, as shown in Figure 4.1
2. An adversary has control over the content contained within V

[3]Because another body frame was expected, when receiving a method frame.

3. The broker follows the formal AMQP 0-9-1 protocol grammar
4. The client follows the RabbitMQ protocol grammar

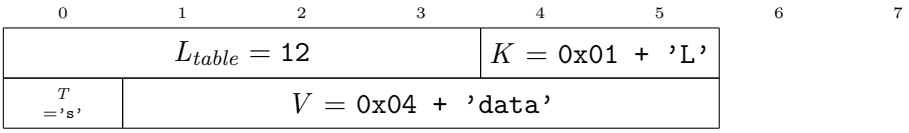


Figure 4.1: AMQP field table containing a `short-string`

The attack is made possible by ambiguity between the data type definitions within `field-table`s and `field-array`s. In the RabbitMQ convention, the `short-string` data type (T in Figure 4.1) will be interpreted as a signed 16 bit integer.

As such, an adversary controlling a `short-string` on the broker side could encode it in such a way that arbitrary key/value-pairs (just values in the case of `field-array`s) can be injected.

Consider a specially crafted `short-string` (including its length indicator, `0x14`) V_{inject} =
`0x14 + 0xff + 7 + 'inj-str' + 'S' + 0x00000006 + 'inject'`

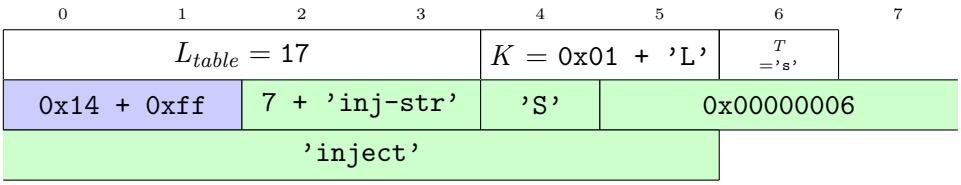


Figure 4.2: AMQP field table data type encoding

As shown in Figure 4.2, an AMQP library following the RabbitMQ convention would decode the first two (in blue) bytes of V_{inject} as a signed 16-bit integer with the value `0x14ff`. The rest of the string (in green) would be considered more elements within the `field-array`. Here, V_{inject} encodes an additional `long-string` named *inj-str* with the value `inject`.

The first byte in V_{inject} cannot be freely set, as it needs to correspond to the length of the injected string in order to comply with the formal protocol grammar. The second byte can be freely set and will correspond to the lower part of the 16 bit signed integer.

The rest of the string can be freely set, allowing for a maximum of $2^8 - 1 - 2 = 253$ bytes to be injected.

This attack is implemented in AMQPTester and was shown to successfully inject arbitrary data into all libraries in Section 2.7 except Py-AMQP, which correctly handles the data type definitions as defined by the AMQP specification.

Chapter 5

Discussion

The result of this thesis has shown that different AMQP implementations does indeed have differences as to how the wire-level protocol is handled. It has also been shown that the AMQ model is being interpreted differently between implementations.

While AMQP may not be as mature and widely used as other protocols such as HTTP or TLS, there still exists high compatibility between implementations, especially for the most commonly used features such as message subscription and publishing.

As many libraries differs from the AMQP specification in the same way (such as how they interpret data type encoding), it would suggest that most have been developed and tested against RabbitMQ rather than against the official specification.

During the development of AMQPTester, multiple AMQP libraries were tested for compatibility. At the same time, the libraries source code were inspected and analysed for possible security or protocol flaws. Attempts were made to find common security errors such as buffer overflows and protocol elevation. This was also how the injection attack in Section 4.3 was discovered.

No other security issues were found during the code analysis. It should be noted that the code analysis was not extensive due to lack of time. Only the code responsible for encoding and decoding data within the wire-level protocol was inspected in each library, respectively.

5.1 Threats to validity

While the utmost care was taken to ensure that AMQPTester does indeed follow the formal AMQP specification, there is always a risk of introducing bugs into any tool used for software testing.

In the case of this thesis and AMQPTester, all failed tests were manually inspected using the popular network analyzer program Wireshark. From Wireshark, the wire-level frames were extracted and manually decoded with pen and paper to ensure that AMQPTester was not the cause of failure.

This strategy worked, as AMQPTester did indeed fail to correctly implement the AMQP protocol in some cases. One such case was found when processing the library mentioned in Section 4.2.2. The library set a frame option which AMQPTester did not know about, causing AMQPTester to enter into a state of failed protocol alignment.

5.2 Future work

While the set goals for this thesis have been achieved, there still exists much room for further research into the AMQP protocol. As AMQPTester makes it easy to implement AMQP in both the server and client side, future research within the field should be made easier.

The AMQPTester implementation could be extended to cover more of the AMQP protocol. Currently, only the frame types and class/method IDs relevant to this thesis are implemented as they were added alongside the test case implementations.

Implementing client support in AMQPTester in order to test AMQP brokers would be another possible future improvement. This would also allow for testing of broker-specific domains, such as Quality of Service queuing, which is not handled at all by the client side.

While AMQPTester is a suitable tool for testing AMQP, it can generally not be used to test similar protocols such as MQTT. This is due to severe differences in the protocol design and the largely different set of features specific to each protocol.

A "reverse" attack similar to the discovered data injection attack may be relevant for future research. In this thesis, the attack was made possible if the broker follows the formal protocol specification and the client follows the RabbitMQ protocol grammar. Similar types of attacks might be achievable

if the protocol grammar is flipped, where the broker follows the RabbitMQ protocol grammar and the client follows the formal specification.

Due to lack of time and the added complexity of such an attack, it was left out from this thesis. In addition, it may be relevant to conduct a study as to how many applications utilizing AMQP are vulnerable, as the prerequisite requirements to inject data are quite strict to achieve.

Chapter 6

Conclusions

As the need to scale large-scale applications grows ever more urgent, middleware solutions like AMQP may hold the key for companies and organizations to easily and linearly scale their applications.

This thesis set out to investigate the AMQP protocol for stability and adherence between different software implementations. While multiple differences were found, it was also found that all of the libraries generally work very well with RabbitMQ, which is the de-facto standard broker used in production today.

While RabbitMQ does not fully adhere to the standard, it would seem that most libraries have been developed with RabbitMQ as a test bench, making most of them both stable and compliant towards the RabbitMQ take of the AMQP specification. In practice, although for the wrong reasons, this makes the current AMQP ecosystem very stable with regards to stability, determinism and security.

The subtle differences between the formal specification and the actual implementations may have implications in the future with regards to both security and stability if another (more standard-compliant) broker were to become popular, as was shown with AMQPTester which was solely developed from the formal specification.

6.1 Updating the AMQP specification

One possible solution to the differences between the standard and various implementations might be to publish a new version of the AMQP 0-9-* branch, updating it in such a way that the RabbitMQ take on the protocol

becomes the new standard. This would not be a huge feat, as the differences between the current standard and the RabbitMQ implementation are somewhat subtle. As most implementations have been shown to adhere more to the RabbitMQ version of the protocol, the effort needed to modify existing implementations would be minimal.

Such an update could also benefit from reducing the set of possible frame types, frame methods and such, as there exists many definitions in the formal specification that are never used. In addition, some parts of the protocol are completely unnecessary, such as transmitting `0xCE` at the end of each frame (because the frame length is already know from the frame header). Many reserved fields and flags with always carry the same content could also be removed, making the protocol more compact and efficient.

6.2 AMQPTester

As no other comparable tools existed during the development of this thesis, AMQPTester was developed. Other AMQP libraries and frameworks were found to not be flexible enough by not allowing low-level protocol access or the ability to easily compose or decode frames.

AMQPTester will most likely make future research within the AMQP protocol simpler and more convenient, as the research community now has an open source reference implementation that, by default, implements AMQP 0-9-1 as per the formal specification.

Much of the AMQPTester code may be re-used for many other purposes as well, as it can be considered a "general purpose" AMQP implementation. One such use case could be implementing an AMQP proxy that sits in-between a broker and a client, decoding and relaying frames, making it easy to inject very specific data into the protocol.

Bibliography

[1] Emmerich, W., 2000, May. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 117–129). ACM.

[2] Geihs, K., 2001. *Middleware challenges ahead.* Computer, 34(6), pp.24-31.

[3] Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.M., 2003. *The many faces of publish/subscribe.* ACM computing surveys (CSUR), 35(2), pp.114–131.

[4] O'Hara, J., 2007. *Toward a commodity enterprise middleware.* Queue, 5(4), pp.48-55.

[5] *AMQP version 0-9-1*, OASIS formal specification, http://www.amqp.org/specification/0-9-1/amqp-org-download, fetched 2018-08-27.

[6] *AMQPTester*, https://github.com/petcap/AMQPTester.

[7] Conversation with Carl Hörberg at 84codes, Sveavägen 98, Stockholm, 19:th of February 2018.

[8] Luzuriaga, J.E., Perez, M., Boronat, P., Cano, J.C., Calafate, C. and Manzoni, P., 2015, January. *A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks.* In Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE (pp. 931–936).

[9] Subramoni et al. *Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand.* Workshop on High Performance Computational Finance, 2008.

[10] Walraven, S., Truyen, E. and Joosen, W., 2011, December. *A middleware layer for flexible and cost-efficient multi-tenant applications.* In Pro-

ceedings of the 12th International Middleware Conference (pp. 360–379). International Federation for Information Processing.

[11] Simon N. Foley, Thomas B. Quillinan, Maeve OConnor, Barry P. Mulcahy, and John P. Morrison *A Framework for Heterogeneous Middleware Security.*, 2004.

[12] Jun Yoneyama, Cyrille Artho, Yoshinori Tanabe, Masami Hagiya. *Model-based Network Fault Injection for IoT Protocols*. Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering. SCITEPRESS-Science and Technology Publications, Lda, 2019.

[13] *Modbat*, https://people.kth.se/ artho/modbat/, accessed 17:th dec 2019

[14] Houimli, Manel, Laid Kahloul, and Sihem Benaoun. *Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things*. 2017 International Conference on Mathematics and Information Technology (ICMIT). IEEE, 2017.

[15] David, Alexandre, et al. *Uppaal SMC tutorial*. International Journal on Software Tools for Technology Transfer 17.4 (2015): 397–415.

[16] G. M. Roy, *RabbitMQ in Depth*, Published by Manning Publications, Shelter Island, NY 11964. ISBN: 9781617291005.

[17] B. Braden, ed., *Requirements for Internet Hosts Communication Layers*, RFC 1122, Oct. 1989.

[18] J. Postel. *RFC 791: Internet protocol*, 1981.

[19] IETF RFC 3629: UTF-8, a transformation format of ISO 10646, Yergeau, Francois, 2003.

[20] IETF RFC 20: ASCII format for Network Interchange, Vint Cerf, Oct. 1969.

[21] *php-amqplib*, https://github.com/petcap/AMQPTester.

[22] *AMQP 0-9-1 library and client for Node.JS*, http://www.squaremobius.net/amqp.node/, accessed 5 dec 2019.

[23] *amqp - Python AMQP low-level client library*, https://amqp.readthedocs.io/en/latest/index.html, accessed 5 dec 2019.

[24] *RabbitMQ C AMQP client library*, https://github.com/alanxz/rabbitmq-c, accessed 5 dec 2019.

[25] *Build RabbitMQ Java Client from Source*, https://www.rabbitmq.com/build-java-client.html, accessed 5 dec 2019.

[26] Patterson, Jeremy, Mehran Habibi, and Terry Camerlengo. *BThe Sun Certified Java Developer Exam with J2SE 1.4*. Apress, 2002.

[27] *AMQP 0-9-1 Errata - RabbitMQ*, https://www.rabbitmq.com/amqp-0-9-1-errata.html.

[28] *AMQP Working Group 0-9-1*, XML Specification, http://www.amqp.org/specification/0-9-1/amqp-org-download, accessed 8 dec 2019