AMQP

A General-Purpose Middleware Standard

AMQP: A General-Purpose Middleware Standard

Copyright Notice

License

Status of this Document

Table of Contents

Credits xi
1. Technical Contributors xi
2. Reviewers xi
I. Concepts 1
1. Overview 4
1.1. Goals of This Document 4
1.2. Patents 4
1.3. Summary 4
1.3.1. What is AMQP? 4
1.3.2. Why AMQP? 4
1.3.3. Scope of AMQP 4
1.3.4. The Advanced Message Queuing Protocol 5
1.3.5. Functional Scope 7
1.4. Organization of This Document 7
1.5. Conventions 7
1.5.1. Definitions 7
1.5.2. Version Numbering 8
1.5.3. Technical Terminology 8
2. The AMQP Model 11
2.1. Introduction to The AMQP Model 11
2.1.1. The Message Queue 11
2.1.2. The Exchange 12
2.1.3. The Routing Key 12
2.1.4. Analogy to Email 12
2.1.5. Message Flow 13
2.2. Virtual Hosts 15
2.3. Exchanges 15
2.3.1. Types of Exchange 16
2.3.2. Exchange Life-cycle 18
2.4. Message Queues 18
2.4.1. Message Queue Properties 19
2.4.2. Queue Life-cycles 19
2.5. Bindings 19
2.5.1. Constructing a Shared Queue 20
2.5.2. Constructing a Reply Queue 20
2.5.3. Constructing a Pub-Sub Subscription Queue 21
2.6. Messages 22
2.6.1. Flow Control 22
2.6.2. Transfer of Responsibility 22
2.7. Subscriptions 22
2.8. Transactions 22
2.9. Distributed Transactions 23
2.9.1. Distributed Transaction Scenario 24
3. Sessions 25
3.1. Session Definition 25
3.1.1. Session Lifetime 25
3.1.2. A Transport For Commands 25
3.1.3. Session as a Layer 25
3.2. Session Functionality 26
3.2.1. Sequential Identification 26
3.2.2. Confirmation 26

3.2.3. Completion 26
3.2.4. Replay and Recovery 27
3.3. Transport requirements 27
3.4. Commands and Controls 27
3.4.1. Commands 27
3.4.2. Controls 28
3.5. Session Lifecycle 28
3.5.1. Attachment 29
3.5.2. Session layer state 29
3.5.3. Reliability 29
3.5.4. Replay 29
3.6. Using Session Controls 30
3.6.1. Attaching to a "new" session 30
3.6.2. Attempting to re-attach to an existing session 31
3.6.3. Detaching cleanly 32
3.6.4. Closing 33
II. Specification 34
4. Transport 40
4.1. IANA Port Number 40
4.2. Protocol Header 40
4.3. Version Negotiation 40
4.4. Framing 41
4.4.1. Assemblies, Segments, and Frames 41
4.4.2. Channels and Tracks 42
4.4.3. Frame Format 43
4.5. SCTP 44
5. Formal Notation 45
5.1. Docs and Rules 45
5.2. Types 46
5.3. Structs 47
5.4. Domains 50
5.4.1. Enums 51
5.5. Constants 51
5.6. Classes 52
5.6.1. Roles 52
5.7. Controls 53
5.7.1. Responses 54
5.8. Commands 54
5.8.1. Results 55
5.8.2. Exceptions 55
5.9. Segments 56
5.9.1. Header Segment 56
5.9.2. Body Segment 57
6. Constants 58
7. Types 59
7.1. Fixed width types 59
7.1.1. bin8 59
7.1.2. int8 60
7.1.3. uint8 61
7.1.4. char 62
7.1.5. boolean 63
7.1.6. bin16 64
7.1.7. int16 65
7.1.8. uint16 66

7.1.9. bin32 67
7.1.10. int32 68
7.1.11. uint32 69
7.1.12. float 70
7.1.13. char-utf32 71
7.1.14. sequence-no 72
7.1.15. bin64 73
7.1.16. int64 74
7.1.17. uint64 75
7.1.18. double 76
7.1.19. datetime 77
7.1.20. bin128 78
7.1.21. uuid 79
7.1.22. bin256 80
7.1.23. bin512 81
7.1.24. bin1024 82
7.1.25. bin40 83
7.1.26. dec32 84
7.1.27. bin72 85
7.1.28. dec64 86
7.1.29. void 87
7.1.30. bit 88
7.2. Variable width types 90
7.2.1. vbin8 90
7.2.2. str8-latin 91
7.2.3. str8 92
7.2.4. str8-utf16 93
7.2.5. vbin16 94
7.2.6. str16-latin 95
7.2.7. str16 96
7.2.8. str16-utf16 97
7.2.9. byte-ranges 98
7.2.10. sequence-set 99
7.2.11. vbin32 100
7.2.12. map 101
7.2.13. list 102
7.2.14. array 103
7.2.15. struct32 104
7.3. Mandatory Types 106
8. Domains 107
8.1. segment-type 107
8.2. track 107
8.3. str16-array 108
9. Control Classes 110
9.1. connection 110
9.1.1. connection.close-code 111
9.1.2. connection.amqp-host-url 112
9.1.3. connection.amqp-host-array 113
9.1.4. connection.start 114
9.1.5. connection.start-ok 116
9.1.6. connection.secure 117
9.1.7. connection.secure-ok 118
9.1.8. connection.tune 119
9.1.9. connection.tune-ok 120

9.1.10. connection.open 122
9.1.11. connection.open-ok 123
9.1.12. connection.redirect 124
9.1.13. connection.heartbeat 125
9.1.14. connection.close 126
9.1.15. connection.close-ok 127
9.2. session 129
9.2.1. Rules 130
9.2.2. session.header 130
9.2.3. session.command-fragment 131
9.2.4. session.name 132
9.2.5. session.detach-code 133
9.2.6. session.commands 134
9.2.7. session.command-fragments 135
9.2.8. session.attach 136
9.2.9. session.attached 137
9.2.10. session.detach 138
9.2.11. session.detached 139
9.2.12. session.request-timeout 140
9.2.13. session.timeout 141
9.2.14. session.command-point 142
9.2.15. session.expected 143
9.2.16. session.confirmed 144
9.2.17. session.completed 145
9.2.18. session.known-completed 146
9.2.19. session.flush 147
9.2.20. session.gap 148
10. Command Classes 150
10.1. execution 150
10.1.1. execution.error-code 150
10.1.2. execution.sync 151
10.1.3. execution.result 152
10.1.4. execution.exception 153
10.2. message 155
10.2.1. Rules 157
10.2.2. message.delivery-properties 158
10.2.3. message.fragment-properties 160
10.2.4. message.reply-to 161
10.2.5. message.message-properties 162
10.2.6. message.destination 164
10.2.7. message.accept-mode 165
10.2.8. message.acquire-mode 166
10.2.9. message.reject-code 167
10.2.10. message.resume-id 168
10.2.11. message.delivery-mode 169
10.2.12. message.delivery-priority 170
10.2.13. message.flow-mode 171
10.2.14. message.credit-unit 172
10.2.15. message.transfer 173
10.2.16. message.accept 175
10.2.17. message.reject 176
10.2.18. message.release 177
10.2.19. message.acquire 178
10.2.20. message.resume 179

10.2.21. message.subscribe 180
10.2.22. message.cancel 182
10.2.23. message.set-flow-mode 183
10.2.24. message.flow 184
10.2.25. message.flush 185
10.2.26. message.stop 186
10.3. tx 188
10.3.1. Rules 188
10.3.2. tx.select 188
10.3.3. tx.commit 189
10.3.4. tx.rollback 190
10.4. dtx 192
10.4.1. Rules 193
10.4.2. dtx.xa-result 193
10.4.3. dtx.xid 194
10.4.4. dtx.xa-status 195
10.4.5. dtx.select 196
10.4.6. dtx.start 197
10.4.7. dtx.end 199
10.4.8. dtx.commit 201
10.4.9. dtx.forget 203
10.4.10. dtx.get-timeout 204
10.4.11. dtx.prepare 205
10.4.12. dtx.recover 207
10.4.13. dtx.rollback 208
10.4.14. dtx.set-timeout 210
10.5. exchange 212
10.5.1. Rules 212
10.5.2. exchange.name 213
10.5.3. exchange.declare 214
10.5.4. exchange.delete 217
10.5.5. exchange.query 218
10.5.6. exchange.bind 219
10.5.7. exchange.unbind 222
10.5.8. exchange.bound 223
10.6. queue 226
10.6.1. Rules 226
10.6.2. queue.name 226
10.6.3. queue.declare 227
10.6.4. queue.delete 230
10.6.5. queue.purge 231
10.6.6. queue.query 232
10.7. file 234
10.7.1. Rules 235
10.7.2. file.file-properties 235
10.7.3. file.return-code 236
10.7.4. file.qos 237
10.7.5. file.qos-ok 238
10.7.6. file.consume 239
10.7.7. file.consume-ok 241
10.7.8. file.cancel 242
10.7.9. file.open 243
10.7.10. file.open-ok 244
10.7.11. file.stage 245

10.7.12. file.publish 246
10.7.13. file.return 248
10.7.14. file.deliver 249
10.7.15. file.ack 250
10.7.16. file.reject 251
10.8. stream 253
10.8.1. Rules 254
10.8.2. stream.stream-properties 254
10.8.3. stream.return-code 255
10.8.4. stream.qos 256
10.8.5. stream.qos-ok 257
10.8.6. stream.consume 258
10.8.7. stream.consume-ok 260
10.8.8. stream.cancel 261
10.8.9. stream.publish 262
10.8.10. stream.return 264
10.8.11. stream.deliver 265
11. The Model 267
11.1. Exchanges 267
11.1.1. Mandatory Exchange Types 267
11.1.2. Optional Exchange Types 268
11.1.3. System Exchanges 270
11.1.4. Implementation-defined Exchange Types 271
11.1.5. Exchange Naming 271
11.2. Queues 271
11.2.1. queue_naming 271
12. Protocol Grammar 272
12.1. Augmented BNF Rules 272
12.2. Grammar 272
A. Conformance Tests 278
A.1. Introduction 278
B. Implementation Guide 279
B.1. AMQP Client Architecture 279

Credits

1. Technical Contributors

Sanjay Aiyagari	*Cisco Systems*	Matthew Arrott	*Twist Process Innovations*
Rajith Attapattu	*Red Hat*	Mark Atwell	*JPMorgan Chase*
Jason Brome	*Envoy Technologies*	Alan Conway	*Red Hat*
Tejeswar Das	*IONA Technologies*	Tony Garnock-Jones	*Rabbit Technologies*
Robert Godfrey	*JPMorgan Chase*	Robert Greig	*JPMorgan Chase*
Pieter Hintjens	*iMatix Corporation*	John O'Hara	*JPMorgan Chase*
Navin Kamath	*IONA Technologies*	Hsuan-Chung Lee	*Cisco Systems*
Matthias Radestock	*Rabbit Technologies*	Alexis Richardson	*Rabbit Technologies*
Martin Ritchie	*JPMorgan Chase*	Shahrokh Sadjadi	*Cisco Systems*
Rafael Schloming	*Red Hat*	Steven Shaw	*JPMorgan Chase*
Gordon Sim	*Red Hat*	Arnaud Simon	*Red Hat*
Martin Sustrik	*iMatix Corporation*	Carl Trieloff	*Red Hat*
Kim van der Riet	*Red Hat*	Steve Vinoski	*IONA Technologies*

We also wish to acknowledge the technical contributions of a number of individuals from Credit Suisse.

2. Reviewers

Kayshav Dattatri	*Cisco Systems*	Aidan Skinner	*JPMorgan Chase*
Rupert Smith	*JPMorgan Chase*	Subbu Srinivasan	*Cisco Systems*
Andrew Stitcher	*Red Hat*		

Part I. Concepts

Table of Contents

1. Overview 4
    1.1. Goals of This Document 4
    1.2. Patents 4
    1.3. Summary 4
        1.3.1. What is AMQP? 4
        1.3.2. Why AMQP? 4
        1.3.3. Scope of AMQP 4
        1.3.4. The Advanced Message Queuing Protocol 5
        1.3.5. Functional Scope 7
    1.4. Organization of This Document 7
    1.5. Conventions 7
        1.5.1. Definitions 7
        1.5.2. Version Numbering 8
        1.5.3. Technical Terminology 8
2. The AMQP Model 11
    2.1. Introduction to The AMQP Model 11
        2.1.1. The Message Queue 11
        2.1.2. The Exchange 12
        2.1.3. The Routing Key 12
        2.1.4. Analogy to Email 12
        2.1.5. Message Flow 13
    2.2. Virtual Hosts 15
    2.3. Exchanges 15
        2.3.1. Types of Exchange 16
        2.3.2. Exchange Life-cycle 18
    2.4. Message Queues 18
        2.4.1. Message Queue Properties 19
        2.4.2. Queue Life-cycles 19
    2.5. Bindings 19
        2.5.1. Constructing a Shared Queue 20
        2.5.2. Constructing a Reply Queue 20
        2.5.3. Constructing a Pub-Sub Subscription Queue 21
    2.6. Messages 22
        2.6.1. Flow Control 22
        2.6.2. Transfer of Responsibility 22
    2.7. Subscriptions 22
    2.8. Transactions 22
    2.9. Distributed Transactions 23
        2.9.1. Distributed Transaction Scenario 24
3. Sessions 25
    3.1. Session Definition 25
        3.1.1. Session Lifetime 25
        3.1.2. A Transport For Commands 25
        3.1.3. Session as a Layer 25
    3.2. Session Functionality 26
        3.2.1. Sequential Identification 26
        3.2.2. Confirmation 26
        3.2.3. Completion 26
        3.2.4. Replay and Recovery 27
    3.3. Transport requirements 27
    3.4. Commands and Controls 27

3.4.1. Commands 27
3.4.2. Controls 28
3.5. Session Lifecycle 28
3.5.1. Attachment 29
3.5.2. Session layer state 29
3.5.3. Reliability 29
3.5.4. Replay 29
3.6. Using Session Controls 30
3.6.1. Attaching to a "new" session 30
3.6.2. Attempting to re-attach to an existing session 31
3.6.3. Detaching cleanly 32
3.6.4. Closing 33

1. Overview

1.1. Goals of This Document

This document defines a networking protocol, the Advanced Message Queuing Protocol (AMQP), which enables conforming client applications to communicate with conforming messaging middleware services. To fully achieve this interoperability we also define the normative behavior of the messaging middleware service.

We address a technical audience with some experience in the domain, and we provide sufficient specifications and guidelines that a suitably skilled engineer can construct conforming solutions in any modern programming language or hardware platform.

1.2. Patents

A conscious design objective of AMQP was to base it on concepts taken from existing, unencumbered, widely implemented standards such those published by the Internet Engineering Task Force (IETF) or the World Wide Web Consortium (W3C).

Consequently, we believe it is possible to create AMQP implementations using only well known techniques such as those found in existing Open Source networking and email routing software or which are otherwise well-known to technology experts.

1.3. Summary

1.3.1. What is AMQP?

The Advanced Message Queuing Protocol (AMQP) enables full functional interoperability between conforming clients and messaging middleware servers (also called "brokers").

1.3.2. Why AMQP?

Our goal is to enable the development and industry-wide use of standardized messaging middleware technology that will lower the cost of enterprise and systems integration and provide industrial-grade integration services to a broad audience.

It is our aim that, through AMQP, messaging middleware capabilities may ultimately be driven into the network itself, and that through the pervasive availability of messaging middleware, new kinds of useful applications may be developed.

1.3.3. Scope of AMQP

To enable complete interoperability for messaging middleware requires that both the networking protocol and the semantics of the broker services are sufficiently specified.

AMQP, therefore, defines both the network protocol and the broker services through:

1. A *defined set of messaging capabilities* called the "Advanced Message Queuing Protocol Model" (AMQP Model). The AMQP Model consists of a set of components that route and store messages within the broker, plus a set of rules for wiring these components together.

2. A *network wire-level protocol*, AMQP, that lets client applications talk to the broker and interact with the AMQP Model it implements.

One can partially imply the semantics of the server from the AMQP protocol specifications but we believe that an explicit description of these semantics helps the understanding of the protocol.

1.3.4. The Advanced Message Queuing Protocol

1.3.4.1. The AMQP Model

We define the server's semantics explicitly, since interoperability demands that these semantics be the same in any given server implementation.

The AMQP Model thus specifies a modular set of components and standard rules for connecting these.

There are three main types of component, which are connected into processing chains in the server to create the desired functionality:

1. The "*exchange*" receives messages from publisher applications and routes these to "message queues", based on arbitrary criteria, usually message properties or content.

2. The "*message queue*" stores messages until they can be safely processed by a consuming client application (or multiple applications).

3. The "*binding*" defines the relationship between a message queue and an exchange and provides the message routing criteria.

Using this model we can emulate the classic middleware concepts of store-and-forward queues and topic subscriptions trivially.

In very gross terms, an AMQP server is analogous to an email server, with each exchange acting as a message transfer agent, and each message queue as a mailbox. The bindings define the routing tables in each transfer agent. Publishers send messages to individual transfer agents, which then route the messages into mailboxes. Consumers take messages from mailboxes.

In many pre-AMQP middleware systems, by contrast, publishers send messages directly to individual mailboxes (in the case of store-and-forward queues), or to mailing lists (in the case of topic subscriptions).

The difference is that when the rules connecting message queues to exchanges are under control of the architect (rather than embedded in code), it becomes possible to do interesting things, such as define a rule that says, "place a copy of all messages containing such-and-such a header into this message queue".

The AMQP model was conceived with the following goals:

1. To support the semantics required by the financial services industry.

2. To provide the levels of performance required by the financial services industry.

3. To be easily extended for new kinds of message routing and queuing.

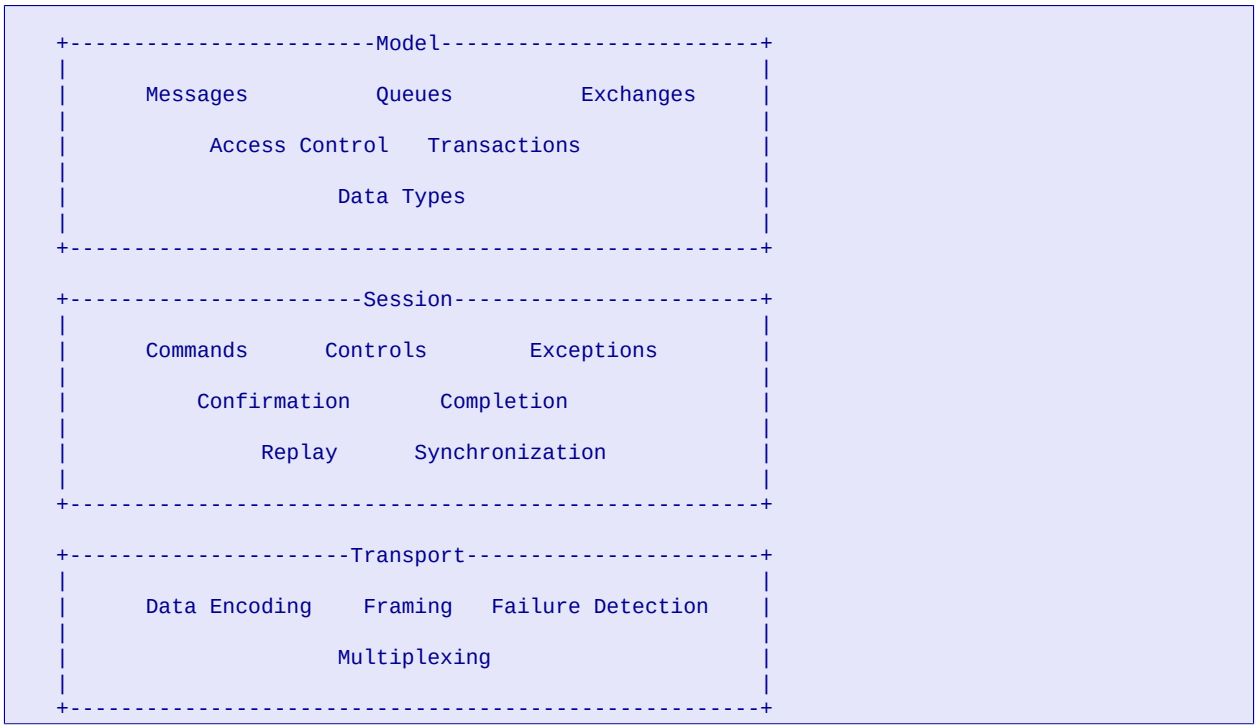
4. To permit the server's specific semantics to be programmed by the application, via the protocol.

5. To be flexible yet simple.

1.3.4.2. The AMQP Protocol

The AMQP protocol is a binary protocol with modern features: it is multi-channel, negotiated, asynchronous, secure, portable, neutral, and efficient.

AMQP is usefully split into three layers:



The model layer defines a set of commands (grouped into logical classes of functionality) that do useful work on behalf of the application.

The session layer provides reliable transport of commands from application to server and back with replay, synchronization, and error handling.

The transport layer provides framing, channel multiplexing, failure detection, and data representation.

One could replace the transport layer with arbitrary transports without changing the application-visible functionality of the protocol. One could also use the same session layer for different high-level protocols.

The design of the AMQP Model was driven by these requirements:

1. To guarantee interoperability between conforming implementations.
2. To provide explicit control over the quality of service.
3. To support any middleware domain: messaging, file transfer, streaming, RPC, etc.
4. To accommodate existing open messaging API standards (for example, Sun's JMS).
5. To be consistent and explicit in naming.
6. To allow complete configuration of server wiring via the protocol.
7. To use a command notation that maps easily into application-level API's.
8. To be clear, so each operation does exactly one thing.

The design of the AMQP transport layer was driven by these main requirements, in no particular order:

1. To be compact, using a binary encoding that packs and unpacks rapidly.
2. To handle messages of any size without significant limit.

3. To permit zero-copy data transfer (e.g. remote DMA).
4. To carry multiple sessions across a single connection.
5. To allow sessions to survive network failure, server failover, and application recovery.
6. To be long-lived, with no significant in-built limitations.
7. To be asynchronous.
8. To be easily extended to handle new and changed needs.
9. To be forward compatible with future versions.
10. To be repairable, using a strong assertion model.
11. To be neutral with respect to programming languages.
12. To fit a code generation process.

1.3.5. Functional Scope

We support a variety of messaging architectures:

1. Store-and-forward with many writers and one reader
2. Transaction distribution with many writers and many readers
3. Publish-subscribe with many writers and many readers
4. Content-based routing with many writers and many readers
5. Queued file transfer with many writers and many readers
6. Point-to-point connection between two peers

1.4. Organization of This Document

The document is divided into two parts:

1. "*Concepts*", which provides and introduction to the concepts in AMQP, a narrative introduction to how AMQP works, and how AMQP may be used.
2. "*Specification*", in which we define precisely the semantics of every part of the AMQP model layer; the session layer; and define a wire format for the transmission of AMQP over a network.

1.5. Conventions

1.5.1. Definitions

1. We use the terms MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY as defined by IETF RFC 2119.
2. We use the term "the server" when discussing the specific behavior required of a conforming AMQP server.
3. We use the term "the client" when discussing the specific behavior required of a conforming AMQP client.

4. We use the term "the peer" to mean "the server or the client".

5. All numeric values are decimal unless otherwise indicated.

6. Protocol constants are shown as upper-case names. AMQP implementations SHOULD use these names when defining and using constants in source code and documentation.

7. Property names, command or control arguments, and frame fields are shown as lower-case names. AMQP implementations SHOULD use these names consistently in source code and documentation.

1.5.2. Version Numbering

The AMQP version is expressed using two numbers – the major number and the minor number. By convention, the version is expressed as the major number followed by a dash, followed by the minor number. (For example, 1-3 is major = 1, minor = 3.)

1. Major and minor numbers may take any value between 0 and 255 inclusive.

2. Minor numbers are incremented with the major version remaining unchanged. When the AMQP working group decides that a major version is appropriate, the major number is incremented, and the minor number is reset to 0. Thus, a possible sequence could be 1-2, 1-3, 1-4, 2-0, 2-1...

3. Once the protocol reaches production (major >= 1), minor numbers greater than 9 would be strongly discouraged. However, prior to production (versions 0-x), this may occur owing to the rapid and frequent revisions of the protocol.

4. Once the protocol reaches production (major >=1), backwards compatibility between minor versions of the same major version must be guaranteed by implementers. Conversely, backwards compatibility between minor versions prior to production is neither guaranteed nor expected.

5. Major version numbers of 99 and above are reserved for internal testing and development purposes.

1.5.3. Technical Terminology

The following terms have special significance within the context of this document:

1. *AMQP Model*: A logical framework representing the key entities and semantics which must be made available by an AMQP compliant server implementation, such that the server can be meaningfully manipulated by AMQP Commands sent from a client in order to achieve the semantics defined in this specification.

2. *Connection*: A network connection, e.g. a TCP/IP socket connection.

3. *Session*: A named dialog between peers. Within the context of a Session, *exactly-once* delivery is guaranteed.

4. *Channel*: An independent bidirectional stream within a multiplexed connection. The physical transport for a connected session.

5. *Client*: The initiator of an AMQP connection or session. AMQP is not symmetrical. Clients produce and consume messages whereas servers queue and route messages.

6. *Server*: The process that accepts client connections and implements the AMQP message queuing and routing functions. Also known as "broker"

7. *Peer*: Either party in an AMQP dialog. An AMQP connection involves exactly two peers (one is the client, one is the server)

8. *Partner*: The term Partner is used as a convenient shorthand for describing the "other" Peer when describing an interaction between two Peers. For example if we have defined Peer A and Peer B as opposite ends of a given interaction, then Peer B is Peer A's Partner and Peer A is Peer B's partner.
9. *Assembly*: An ordered collection of Segments that form a logical unit of work.
10. *Segment*: An ordered collection of Frames that together form a complete syntactic sub-unit of an Assembly.
11. *Frame*: The atomic unit of transmission in AMQP. A Frame is an arbitrary fragment of a Segment.
12. *Control*: A formally defined one-way instruction assumed to be unreliably transported.
13. *Command*: A formally defined and identified instruction requiring acknowledgement. AMQP attempts to reliably transport Commands.
14. *Exception*: A formally defined error condition that may occur during execution of one or more commands.
15. *Class*: A collection of AMQP commands or controls that deal with a specific type of functionality.
16. *Header*: A specific type of Segment that describes properties of message data.
17. *Body*: A specific type of Segment that contains application data. Body segments are entirely opaque - the server does not examine or modify these in any way.
18. *Content*: The message data contained within a body segment.
19. *Exchange*: An entity within the server which receives messages from producer applications and routes these to message queues within the server.
20. *Exchange Type*: The classification of an exchange based on routing semantics.
21. *Message Queue*: A named entity that holds messages until they can be sent to consumer applications.
22. *Binding*: A relationship that defines routing between a Message Queue and an Exchange.
23. *Binding Key*: A name for a binding. Some exchange types may use this as a pattern that defines the routing behavior for the Binding.
24. *Routing Key*: A message header that an Exchange may use to decide how to route a specific message.
25. *Durable*: A server resource that survives a server restart.
26. *Transient*: A server resource that is wiped or reset after a server restart.
27. *Persistent*: A message that the server holds on reliable disk storage and MUST NOT lose after a server restart.
28. *Non-Persistent*: A message that the server holds in memory and MAY lose after a server restart.
29. *Consumer*: A client application that requests messages from a message queue.
30. *Producer*: A client application that publishes messages to an exchange.
31. *Virtual Host*: A collection of exchanges, message queues and associated objects. Virtual hosts are independent server domains that share a common authentication and encryption environment. The client application chooses a virtual host after logging in to the server.

These terms have *no special significance* within the context of AMQP:

1. *Topic*: Usually a means of distributing messages; AMQP implements topics using one or more types of exchange.

2. *Service*: Usually synonymous with server. The AMQP standard uses "server" to conform with IETF standard nomenclature and to clarify the roles of each party in the protocol (both sides may be AMQP services).

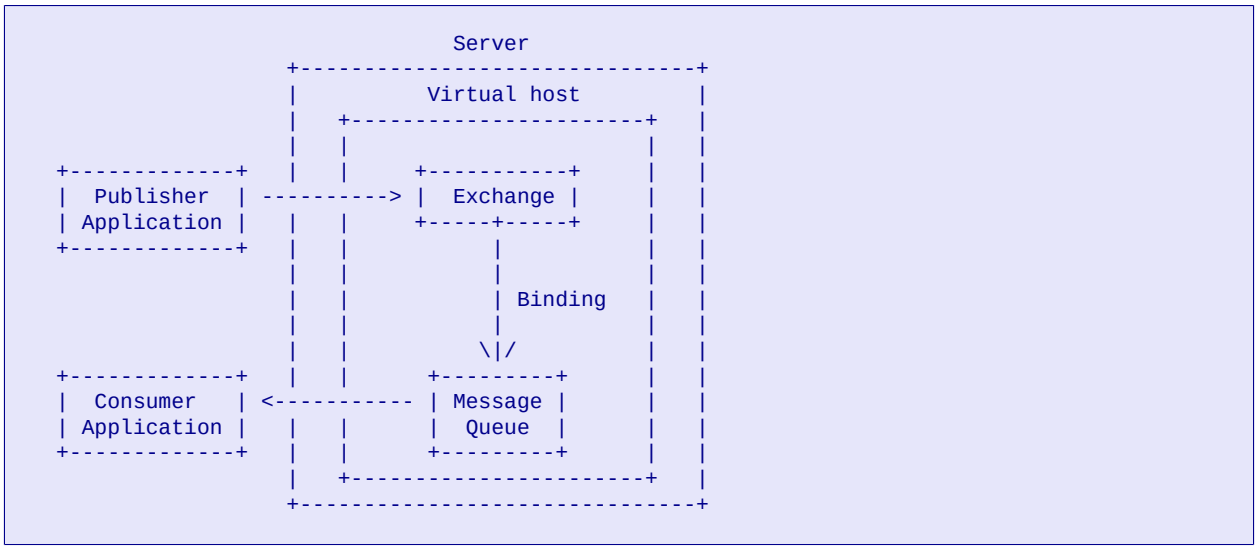
3. *Broker*: synonymous with server. The AMQP standard uses the terms "client" and "server" to conform with IETF standard nomenclature.

2. The AMQP Model

2.1. Introduction to The AMQP Model

This section explains the server semantics that must be standardized in order to guarantee interoperability between AMQP implementations.

This diagram shows the overall AMQP Model:



We can summarize what a middleware server is: it is a data server that accepts messages and does two main things with them; it routes them to different consumers depending on arbitrary criteria, and it buffers them in memory or on disk when consumers are not able to accept them fast enough.

In a pre-AMQP server these tasks are done by monolithic engines that implement specific types of routing and buffering. The AMQP Model takes the approach of smaller, modular pieces that can be combined in more diverse and robust ways. It starts by dividing these tasks into two distinct roles:

1. The exchange, which accepts messages from producers and routes them to message queues.

2. The message queue, which stores messages and forwards them to consumer applications.

There is a clear interface between exchange and message queue, called a "binding", which we will come to later. The usefulness of the AMQP Model comes from three main features:

1. The ability to create arbitrary exchange and message queue types (some are defined in the standard, but others can be added as server extensions).

2. The ability to wire exchanges and message queues together to create any required message-processing system.

3. The ability to control this completely through the protocol.

In fact, AMQP provides runtime-programmable semantics.

2.1.1. The Message Queue

A message queue stores messages in memory or on disk, and delivers these in sequence to one or more consumer applications. Message queues are message storage and distribution entities. Each message queue is entirely independent.

A message queue has various properties: private or shared, durable or transient, permanent or temporary. By selecting the desired properties, we can use a message queue to implement conventional middleware entities such as:

1. A standard **store-and-forward queue**, which holds messages and distributes these between subscribers on a round-robin basis. Store and forward queues are typically durable and shared between multiple subscribers.

2. A **temporary reply queue**, which holds messages and forwards these to a single subscriber. Reply queues are typically temporary, and private to one subscriber.

3. A "**pub-sub**" subscription queue, which holds messages collected from various "subscribed" sources, and forwards these to a single subscriber. Subscription queues are typically temporary, and private to one subscriber.

These categories are not formally defined in AMQP: they are examples of how message queues can be used. It is trivial to create new entities such as durable, shared subscription queues.

2.1.2. The Exchange

An exchange accepts messages from a producer application and routes them to message queues according to pre-arranged criteria. These criteria are called "bindings". Exchanges are matching and routing engines. That is, they inspect messages and using their binding tables, decide how to forward these messages to message queues. Exchanges never store messages.

The term "exchange" is used to mean both a class of algorithm, and the instances of such an algorithm. More properly, we speak of the "exchange type" and the "exchange instance".

AMQP defines a number of standard exchange types, which cover the fundamental types of routing needed to do common message delivery. AMQP servers will provide default instances of these exchanges. Applications that use AMQP can additionally create their own exchange instances. Exchange types are named so that applications which create their own exchanges can tell the server what exchange type to use. Exchange instances are also named so that applications can specify how to bind queues and publish messages.

The exchange concept is intended to define a model for adding extensibile routing behavior to AMQP servers.

2.1.3. The Routing Key

In the general case, an exchange examines a message's properties, its header fields, and its body content, and using this and possibly data from other sources, decides how to route the message.

In the majority of simple cases, the exchange examines a single key field, which we call the "routing key". The routing key is a virtual address that the exchange may use to decide how to route the message.

For **point-to-point** routing, the routing key is the name of a message queue.

For **topic pub-sub** routing, the routing key is the topic hierarchy value.

In more complex cases, routing may be based on message header fields and/or the message body.

2.1.4. Analogy to Email

If we make an analogy with an email system, we see that the AMQP concepts are not radical:

1. An AMQP message is analogous to an email message.

2. A message queue is like a mailbox.

3. A consumer is like a mail client that fetches and deletes email.

4. An exchange is like an MTA (mail transfer agent) that inspects email and decides, on the basis of routing keys and tables, how to send the email to one or more mailboxes.

5. A routing key corresponds to an email To: or Cc: or Bcc: address, without the server information (routing is entirely internal to an AMQP server).

6. Each exchange instance is like a separate MTA process, handling some email sub-domain, or particular type of email traffic.

7. A binding is like an entry in an MTA routing table.

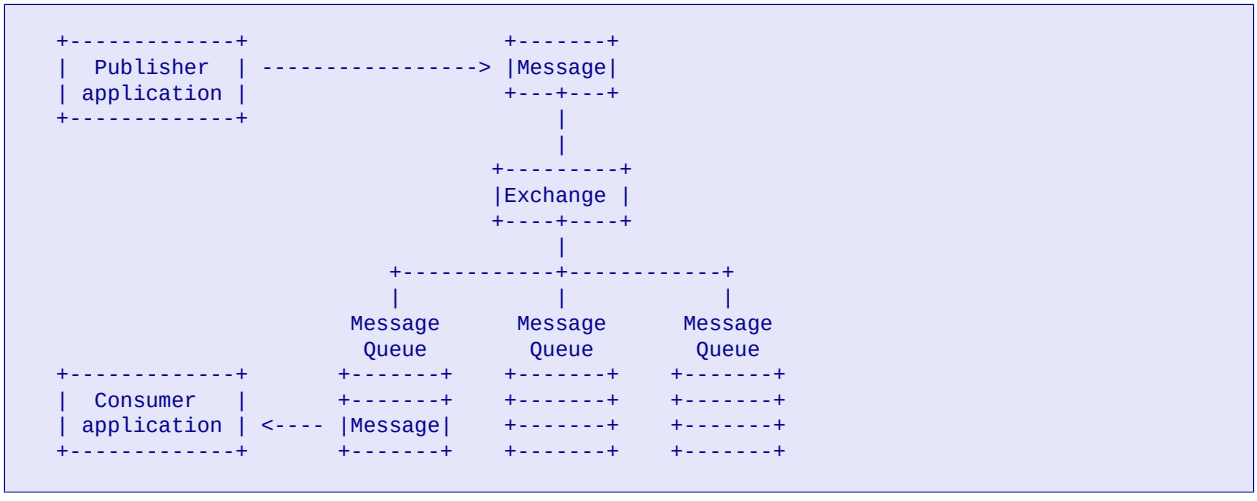
The power of AMQP comes from our ability to create queues (mailboxes), exchanges (MTA processes), and bindings (routing entries), at runtime, and to chain these together in ways that go far beyond a simple mapping from "to" address to mailbox name.

We should not take the email-AMQP analogy too far: there are fundamental differences. The challenge in AMQP is to route and store messages within a server. Routing within a server and routing between servers are distinct problems and have distinct solutions, if only for banal reasons such as maintaining transparent performance.

To route between AMQP servers owned by different entities, one sets up explicit bridges, where one AMQP server acts as the client of another server for the purpose of transferring messages between those separate entities. This way of working tends to suit the types of businesses where AMQP is expected to be used, because these bridges are likely to be underpinned by business processes, contractual obligations and security concerns. This model also makes AMQP 'spam' more difficult.

2.1.5. Message Flow

This diagram shows the flow of messages through the AMQP Model server:



2.1.5.1. Message Life-cycle

An AMQP message consists of a set of header properties plus an opaque body.

A new message is created by a producer application using a client API[1]. The producer places application data in the message body and perhaps sets some message properties. The producer labels the message with routing information,

[1]Note that the AMQP specification does not currently define a standard client API.

which is superficially similar to an address, but almost any scheme can be created. The producer then sends the message to an exchange on the server.

When the message arrives at the server, the exchange (usually) routes the message to a set of message queues which also exist on the server. If the message is unroutable, the exchange may drop it silently, reject it, or route it to an alternate exchange depending on the behavior requested by the producer.

A single message can exist on many message queues. An AMQP server implementation may handle this in different ways, by copying the message, by using reference counting, etc. This does not affect interoperability. However, when a message is routed to multiple message queues, it is identical on each message queue. There is no unique identifier that distinguishes the various copies.

When a message arrives in a message queue, the message queue tries immediately to pass it to a consumer application via AMQP. If this is not possible, the message queue stores the message (in memory or on disk as requested by the producer) and waits for a subscriber to be ready.

When the message queue can deliver the message to a subscriber, it removes the message from its internal buffers. This can happen immediately, or after the subscriber has successfully processed and explicitly accepted the message. The subscriber chooses how and when messages are accepted. The subscriber can also release a message back onto the queue, or reject a message as unprocessable.

Producer messages and subscriber accepts are grouped into "transactions". When an application plays both roles, which is often, it does a mix of work: sending and accepting messages, and then committing or rolling back the transaction [2].

2.1.5.2. What The Producer Sees

By analogy with the email system, we can see that a producer does not send messages directly to a message queue. Allowing this would break the abstraction in the AMQP Model. It would be like allowing email to bypass the MTA's routing tables and arrive directly in a mailbox. This would make it impossible to insert intermediate filtering and processing, spam detection, for instance.

The AMQP Model uses the same principle as an email system: all messages are sent to a single point, the exchange, which inspects the messages based on rules and information that are hidden from the sender, and routes them to drop-off points that are also hidden from the sender.

2.1.5.3. What The Consumer Sees

Our analogy with email starts to break down when we look at consumers. Email clients are passive - they can read their mailboxes, but they do not have any influence on how these mailboxes are filled. An AMQP consumer can also be passive, just like email clients. That is, we can write an application that expects a particular message queue to be ready and bound, and which will simply process messages off that message queue.

However, we also allow AMQP client applications to:

1. Create or destroy message queues.

2. Define the way these message queues are filled, by making bindings.

3. Select different exchanges which can completely change the routing semantics.

This is like having an email system where one can, via the protocol:

1. Create a new mailbox.

2. Tell the MTA that all messages with a specific header field should be copied into this mailbox.

[2] Message deliveries from the server to the subscriber are not transacted.

3. Completely change how the mail system interprets addresses and other message headers.

We see that AMQP is more like a language for wiring pieces together than a system. This is part of our objective, to make the server behavior programmable via the protocol.

2.1.5.4. Default Flow

Most integration architectures do not need this level of sophistication. Like the amateur photographer, a majority of AMQP users need a "point and shoot" mode. AMQP provides this through the use of two simplifying concepts:

1. A **default exchange for message producers**.

2. A **default binding for message queues** that selects messages based on a match between routing key and message queue name.

In effect, **the default binding lets a producer send messages directly to a message queue**, given suitable authority – it emulates the simplest "send to destination" addressing scheme people have come to expect of traditional middleware.

2.2. Virtual Hosts

A Virtual Host [3] comprises its own name space, a set of exchanges, message queues, and all associated objects. Each connection MUST BE associated with a single virtual host.

The client selects the virtual host after authentication. This requires that the authentication scheme of the server is shared between all virtual hosts on that server. The authorization scheme used MAY be unique to each virtual host.

All channels within the connection work with the same virtual host. There is no way to communicate with a different virtual host on the same connection, nor is there any way to switch to a different virtual host without tearing down the connection and beginning afresh.

The protocol offers no mechanisms for creating or configuring virtual hosts - this is done in an undefined manner within the server and is entirely implementation-dependent.

2.3. Exchanges

An exchange is a message routing agent within a virtual host. An exchange instance (which we commonly call "an exchange") accepts messages and routing information - principally a routing key - and either passes the messages to message queues, or possibly to some internal service defined in a vendor extension. Exchanges are named on a per-virtual host basis.

Applications can freely create, share, use, and destroy exchange instances, within the limits of their authority.

Exchanges may be durable, transient, or auto-deleted. Durable exchanges last until they are deleted. Transient exchanges last until the server shuts-down. Auto-deleted exchanges last until they are no longer used.

The server provides a specific set of exchange types. Each exchange type implements a specific matching and routing algorithm, as defined in the next section. AMQP mandates a small number of exchange types, and recommends some more. Further, each server implementation may add its own exchange types.

An exchange can route a single message to many message queues in parallel. This creates multiple instances of the message that are consumed independently.

[3]The term Virtual Host is taken from the use popularized by the Apache HTTP server. Apache Virtual Hosts enable Internet Service providers to provide bulk hosting from one shared server infrastructure. We hope that the inclusion of this capability within AMQP opens up similar opportunities to larger organizations.

2.3.1. Types of Exchange

Each exchange type implements a specific routing algorithm. There are a number of standard exchange types, explained below, but there are two that are particularly important:

1. the *direct* exchange type, which routes based on an exact match between the binding key and routing key

2. the *topic* exchange type, which routes based on a pattern match between the binding key and routing key

Note that:

1. the default exchange (See: Section 2.1.5.4, “Default Flow”) is a direct exchange

2. the server will create a direct and (if supported) a topic exchange at start-up with well-known names and client applications may depend on this

2.3.1.1. The Direct Exchange Type

The direct exchange type provides routing of messages to zero or more queues based on an exact match between the routing key of the message, and the binding key used to bind the queue to the exchange. This can be used to construct the classic point-to-point queue based messaging model, however, as with any of the defined exchange types, a message may end up in multiple queues when multiple binding keys match the message's routing key.

The direct exchange type works as follows:

1. A message queue is bound to the exchange using a binding key, K.

2. A publisher sends the exchange a message with the routing key R.

3. The message is passed to all message queues bound to the exchange with key K where K = R.

The server MUST implement the direct exchange type and MUST pre-declare within each virtual host at least two direct exchanges: one named "**amq.direct**", and one with **no public name** that serves as the default exchange for message transfers to the server.

Note that message queues can be bound using any valid binding key value, but most often message queues will be bound using their own name as the binding key.

In particular, all message queues MUST BE automatically bound to the nameless exchange using the message queue's name as the binding key.

2.3.1.2. The Fanout Exchange Type

The fanout exchange type provides routing of messages to all bound queues regardless of the message's routing key.

The fanout exchange type works as follows:

1. A message queue is bound to the exchange with no arguments.

2. A publisher sends the exchange a message.

3. The message is passed to the all message queues bound to the exchange unconditionally.

The server MUST implement the fanout exchange type and MUST pre-declare within each virtual host at least one fanout exchange named "**amq.fanout**".

2.3.1.3. The Topic Exchange Type

The topic exchange type provides routing to bound queues based on a pattern match between the binding key and the routing key of the message. This exchange type may be used to support the classic publish/subscribe paradigm using a topic namespace as the addressing model to select and deliver messages across multiple consumers based on a partial or full match on a topic pattern.

The topic exchange type works as follows:

1. A message queue is bound to the exchange using a binding key, K.

2. A publisher sends the exchange a message with the routing key R.

3. The message is passed to the all message queues where K matches R.

The binding key is formed using zero or more tokens, with each token delimited by the '.' char. The binding key MUST be specified in this form and additionally supports special wild-card characters: '*' matches a single word and '#' matches zero or more words.

Thus the binding key "*.stock.#" matches the routing keys "usd.stock" and "eur.stock.db" but not "stock.nasdaq".

This exchange type is optional.

The server SHOULD implement the topic exchange type and in that case, the server MUST pre-declare within each virtual host at least one topic exchange, named "**amq.topic**".

2.3.1.4. The Headers Exchange Type

The headers exchange provides for complex, multi-part expression routing based on header properties within the AMQP message.

The headers exchange type works as follows:

1. A message queue is bound to the exchange with a table of arguments containing the headers to be matched for that binding and optionally the values they should hold.

2. A publisher sends a message to the exchange where the 'headers' property contains a table of names and values.

3. The message is passed to the queue if the headers property matches the arguments with which the queue was bound.

The matching algorithm is controlled by a special bind argument passed as a name value pair in the arguments table. The name of this argument is 'x-match'. It can take one of two values, dictating how the rest of the name value pairs in the table are treated during matching:

(i) 'all' implies that all the other pairs must match the headers property of a message for that message to be routed (i.e. an AND match)

(ii) 'any' implies that the message should be routed if any of the fields in the headers property match one of the fields in the arguments table (i.e. an OR match)

A field in the bind arguments matches a field in the message headers if either (1) the field in the bind arguments has no value and a field of the same name is present in the message headers or (2) if the field in the bind arguments has a value and a field of the same name exists in the message headers and has that same value.

Any field starting with 'x-' other than 'x-match' is reserved for future use and will be ignored.

The server SHOULD implement the headers exchange type and in that case, the server MUST pre-declare within each virtual host at least one headers exchange, named "**amq.match**".

2.3.1.5. The System Exchange Type

The system exchange type works as follows:

1. A publisher sends the exchange a message with the routing key S.

2. The system exchange passes this to a system service S.

System services starting with "amq." are reserved for AMQP usage. All other names may be used freely by server implementations. This exchange type is optional.

2.3.1.6. Implementation-defined Exchange Types

All non-normative exchange types MUST be named starting with "x-". Exchange types that do not start with "x-" are reserved for future use in the AMQP standard.

2.3.2. Exchange Life-cycle

Each AMQP server pre-creates a number of exchanges (more pedantically, "exchange instances"). These exchanges exist when the server starts and cannot be destroyed.

AMQP applications can also create their own exchanges. AMQP does not use a "create" command as such; it uses a "declare" command, which means: "create if not present, otherwise continue". It is plausible that applications will create exchanges for private use and destroy them when their work is finished. AMQP provides a command to destroy exchanges but in general applications do not do this.

In our examples in this chapter, we will assume that the exchanges are all created by the server at start-up. We will not show the application declaring its exchanges.

2.4. Message Queues

A message queue is a named buffer that holds messages on behalf of a set of consumer applications. Applications can freely create, share, use, and destroy message queues, within the limits of their authority.

Message queues provide a limited FIFO guarantee. For messages of equal priority originating from a given producer, delivery to a given consumer will always be attempted in the order the messages were placed on the queue. Should the initial delivery attempt not result in a consumed message, those messages MAY be *re*delivered out of order.

Message queues may be durable, transient, or auto-deleted. Transient message queues last until the server shuts-down. Auto-deleted message queues last until they are no longer used. Any queue may be explicitly deleted if the user (client) has the appropriate permissions.

Message queues hold their messages in memory, on disk, or some combination of these.

Message queues are scoped to a virtual host.

Queue names must consist of between 1 and 255 characters. The first character must be limited to letters a-z or A-Z, digits 0-9, or the underscore character ('_'); all those following must be legal UTF-8 characters.

Message queues hold messages and distribute them to one or more subscribed clients.

Message queues track message *acquisition*. Messages must be acquired to be dequeued. This prevents multiple clients from acquiring and then consuming the same message simultaneously. This may be used to safely load balance messages from a single queue among multiple consumers.

Messages from a queue may be sent to more than one client if messages are released or were initially sent without being acquired. Message queues may distribute unacquired messages to clients in order to permit non destructive browsing of the queue contents.

2.4.1. Message Queue Properties

When a client application creates a message queue, it can select some important properties:

1. **name** - Generally, when applications share a message queue they agree on a message queue name beforehand.
2. **durable** - If specified, the message queue remains present and active when the server restarts. It may lose non-persistent messages if the server restarts.
3. **auto-delete** - If specified, the server will delete the message queue when all clients have finished using it, or shortly thereafter.

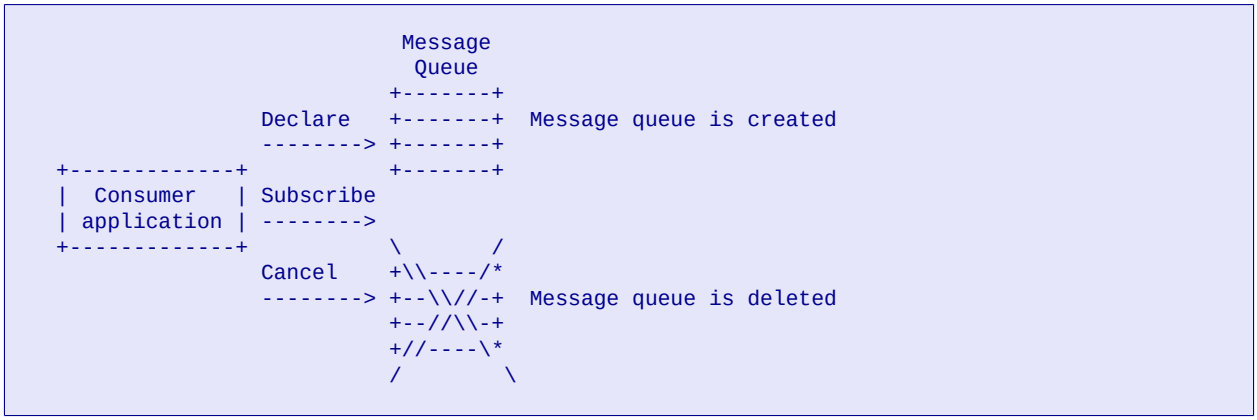
2.4.2. Queue Life-cycles

There are two main message queue life-cycles:

1. **Durable message queues** which are shared by many subscribers and have an independent existence - i.e. they will continue to exist and collect messages whether or not there are subscribers to receive them.
2. **Temporary message queues** which are private to one subscriber and are tied to that subscriber. When the subscriber is cancelled, the message queue is deleted.

There are some variations on these, such as **shared message queues** which are deleted when the last of many subscribers is cancelled.

This diagram shows the way temporary message queues are created and deleted:



2.5. Bindings

A binding is a relationship between a message queue and an exchange. The binding specifies routing arguments that tell the exchange which messages the queue should get.

Applications create and destroy bindings as needed to drive the flow of messages into their message queues. The lifespan of bindings depend on the message queues and exchanges they are defined for - when a message queue, or an exchange, is destroyed, its bindings are also destroyed.

Bindings are constructed by commands from the client application (the one owning and using the message queue) to an exchange. We can express a binding command in pseudo-code as follows:

```
Exchange.Bind <exchange> TO <queue> WHERE <condition>
```

The specific semantics of the Exchange.Bind command depends on the exchange type.

Let's look at three typical use cases: shared queues, private reply queues, and pub-sub subscriptions.

2.5.1. Constructing a Shared Queue

Shared queues are the classic middleware point-to-point queue. In AMQP we can use the default exchange and default binding. Let's assume our message queue is called "app.svc01". Here is the pseudo-code for creating the shared queue:

```
Queue.Declare
    queue=app.svc01
    exclusive=FALSE
```

We may have many consumers on this shared queue. To consume from the shared queue, each consumer does this:

```
Message.Subscribe
    queue=app.svc01
```

To publish to the shared queue, each producer sends a message to the default exchange:

```
Message.Transfer
    routing_key=app.svc01
```

2.5.2. Constructing a Reply Queue

Reply queues are usually temporary. They are also usually private, i.e. read by a single subscriber. Apart from these particularities, reply queues use the same matching criteria as standard queues, so we can also use the default exchange. In order to prevent namespace clashes between temporary queues generated by different clients, it is recommended that clients include a UUID (Universally Unique ID as defined by RFC-4122) or other globally unique identifier in the queue name.

Here is the pseudo-code for creating a reply queue:

```
Queue.Declare
    queue=tmp.550e8400-e29b-41d4-a716-446655440000
    exclusive=TRUE
    auto_delete=TRUE
```

To publish to the reply queue, a producer sends a message to the default exchange:

```
Message.Transfer
    routing_key=tmp.550e8400-e29b-41d4-a716-446655440000
```

One of the standard message properties is Reply-To, which is designed specifically for carrying the name of reply queues.

2.5.3. Constructing a Pub-Sub Subscription Queue

In classic middleware, the term "subscription" is vague and refers to at least two different concepts: the set of criteria that match messages and the temporary queue that holds matched messages. AMQP separates the work into bindings and message queues.

A pub-sub subscription queue collects messages from multiple sources through a set of bindings that match topics, message fields, or content in different ways. The key difference between a subscription queue and a named or reply queue is that the subscription queue name is irrelevant for the purposes of routing, and routing is done on abstracted matching criteria rather than a 1-to-1 matching of the routing key field.

Let's take the common pub-sub model of topic trees and implement this. We need an exchange type capable of matching on a topic tree. In AMQP, this is the topic exchange type. The topic exchange matches wild-cards like "STOCK.USD.*" against routing key values like "STOCK.USD.NYSE".

We **cannot** use the default exchange or binding because these do not support topic-style routing. So we have to create a binding explicitly. Here is the pseudo-code for creating and binding the pub-sub subscription queue:

```
Queue.Declare
    queue=tmp.2
    auto_delete=TRUE

Exchange.Bind
    exchange=amq.topic
    TO
    queue=tmp.2
    WHERE routing_key=STOCK.USD.*
```

As soon as the binding is created, messages will be routed from the exchange into the queue, however, to consume messages from the queue, the client must subscribe:

```
    Message.Subscribe
      queue=tmp.2
```

When publishing a message, the producer does something like this:

```
Message.Transfer
    exchange=amq.topic
    routing_key=STOCK.USD.IBM
```

The topic exchange processes the incoming routing key ("STOCK.USD.IBM") with its binding table, and finds one match, for tmp.2. It then routes the message to that subscription queue.

2.6. Messages

A message is the atomic unit of routing and queuing. Messages have a header consisting of a defined set of properties, and a body that is an opaque block of binary data.

Messages may be persistent - a persistent message is held securely on disk and guaranteed to be delivered even if there is a serious network failure, server crash, overflow etc.

Messages may have a priority level. A high priority message may be sent ahead of lower priority messages waiting in the same message queue. When messages must be discarded, the server will first discard low-priority messages.

The server does not modify message bodies, but may modify specific message headers prior to forwarding them to the consuming application.

2.6.1. Flow Control

Flow control may be used to match the rate at which messages are sent to the available resources at the receiver. The receiver may be an AMQP server receiving messages published by a client, or a client receiving messages sent by an AMQP server from a queue. The same mechanism is used in both cases. In general, flow control uses the concept of *credit* to specify how many messages or how many octets of data may be sent at any given point without exhausting the receiver's resources. Credit is depleted as messages or data is sent, and increased as resources become free at the receiver. Pre-fetch buffers may be used at the receiver to reduce latency.

2.6.2. Transfer of Responsibility

The receiver of a message signals the sender when responsibility for a message has been accepted. When a client sends a message to a server, an accept from the server to the client confirms successful routing and placement of the message on any queues. When a server sends a message to a client, an accept from the client to the server confirms successful processing of the message, and signals the server to remove the message from the queue. AMQP supports two different accept modes:

1. Explicit, in which the receiving application must send an accept for each message, or batch of messages, that is transferred.

2. None, in which the message is considered accepted as soon as it is sent.

2.7. Subscriptions

We use the term *subscription* to mean the entity that controls how a specific client application receives messages from a message queue. This is not to be confused with the separate notion of a subscriber in so-called publish/subscribe messaging. When a client "starts a subscription", it creates a subscription entity in the server. When the client "cancels a subscription", it destroys a subscription entity in the server.

Subscriptions belong to a single client session and cause the message queue to send messages asynchronously to the client.

2.8. Transactions

AMQP defines two separate transaction models, a one-phase commit transaction model (known as *tx*) and a two-phase commit model for distributed transactions (known as *dtx*). The standard, one-phase commit, model acts within the

scope of a single session. The client has control over selecting whether a session is to be transactional or not, but once a session has been selected as a transactional (by the issuance of a `tx.select` command) the session will remain transactional until the point at which it is destroyed.

Once a session has been selected as transactional, then the commands issued by the client that instruct message transfer and message acceptance on that session are only committed on the server once a tx.commit command has been issued. Other commands that alter server state are not transactional, and cannot be rolled back. For instance, the declaration of queues and exchanges are not transactional. [4]

If an AMQP client "publishes" a message (issues a `message.transfer` command) within the scope of a transaction, then the message will not be available for delivery from any queue to which it is routed until the transaction completes. The queues to which the message will be routed are determined at the point at which the message is published, and not at the point of the commit. If the server rejects the message which is published this does not cause rollback of the transaction. Rejection will happen as soon as the server has determined that the message should be rejected and will not wait until a commit is issued.

When a transaction is rolled-back then the effects of the client issued publish and accept commands are discarded. It should be noted that "acquiring" a message is not a transactional operation, and thus any message acquired by the client within the scope of the transaction remains acquired. In practice this means that after a rollback the client still owns all the messages which were delivered to it during the scope of the transaction (whether they were accepted or not). If the client wishes the server to re-take responsibility for these messages, it must issue appropriate release commands.

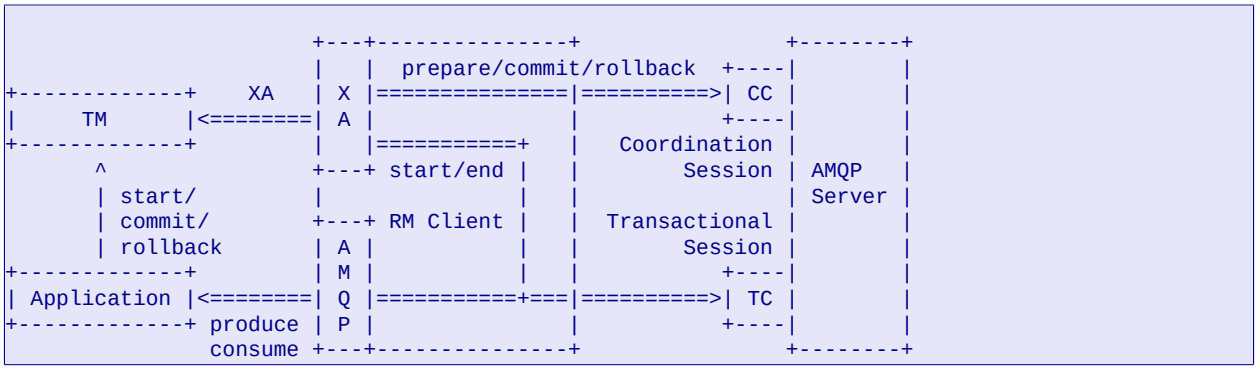
2.9. Distributed Transactions

The distributed transaction class provides support for the X-Open XA architecture.

The dtx class is used to demarcate and coordinate transactions. The dtx.start and dtx.end commands demarcate AMQP transactional work on a given session. Transaction coordination and recovery are provided by the remaining commands in the dtx class.

Both the OMG OTS and JTS/JTA models rely upon "Resource Manager Client" (RM Client) instances, which provide an implementation of the XA interface for the underlying resource that are necessary to participate within a global transaction. These RM Client instances are identified by Rmids through either the xa_switch in C/C++ or XAResource in Java.

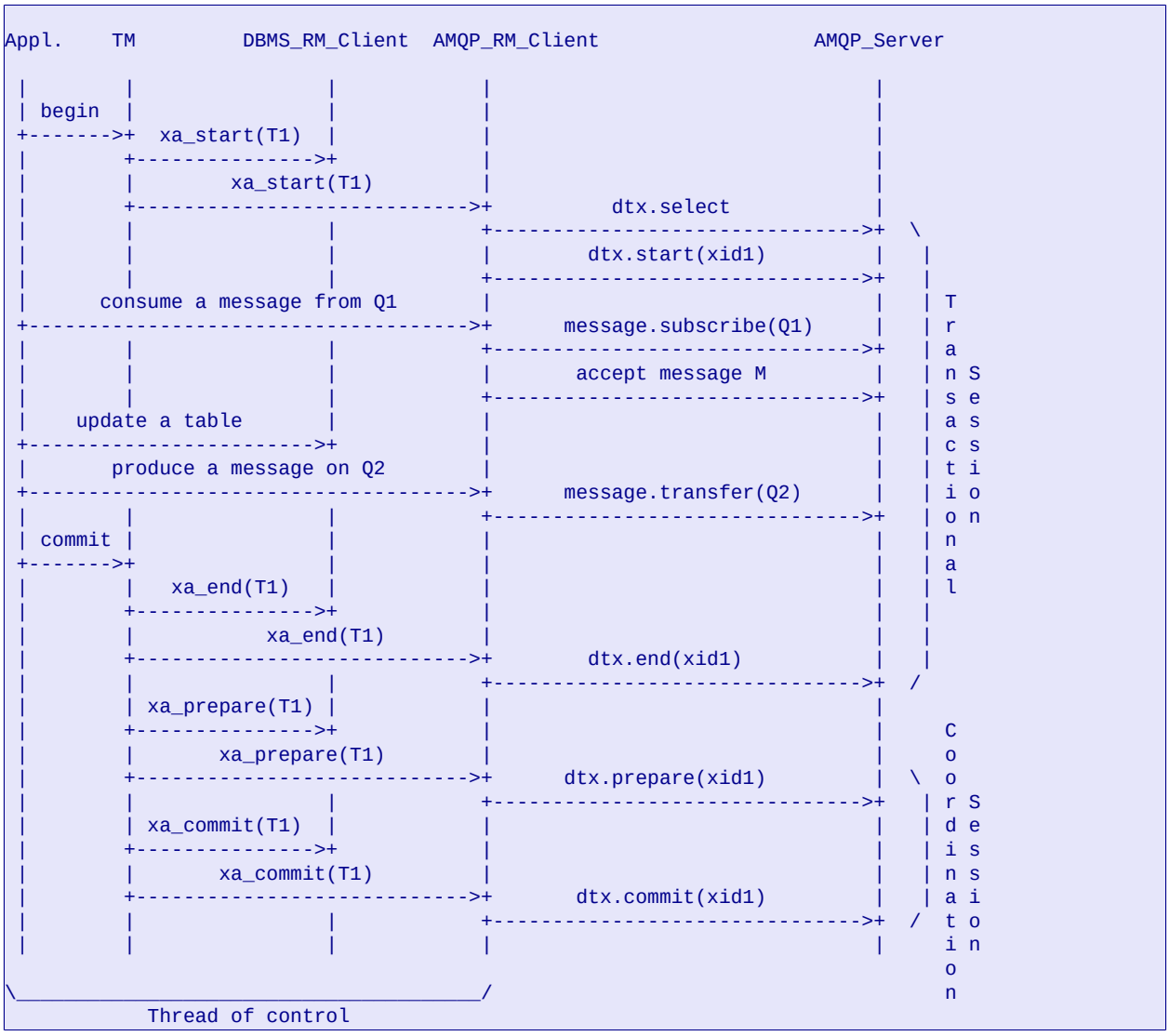
As depicted on the following figure, a Transaction Manager uses the RM Client XA interface to demarcate transaction boundaries and coordinate transaction outcomes. RM Clients use the dtx.start and dtx.end commands to associate a transaction with a session. The transactional session is then exposed to the application driving the transaction, and may be used to transactionally produce and consume messages. RM clients use the dtx coordination commands to propagate transaction outcomes and recovery operations to the AMQP server. A second coordination session can be used for that purpose.



[4] It may help to think of it as the "enqueue" and "de-queue" operations on the message queues as being those that are controlled by transactions.

2.9.1. Distributed Transaction Scenario

The following diagram illustrates a messaging scenario where an application "Application" transactionally consumes a message from a queue Q1 (using transaction T1 achieved through the transaction manger TM). Based on the consumed message, the application updates a database table Tb using DBMS and produces a message on queue Q2 on behalf of transaction T1.



3. Sessions

3.1. Session Definition

Sessions are named interactions between AMQP peers. A session name is scoped to an authentication principal, and the name is determined by the application layer. Sessions may have state associated with them, on one or both of the peers participating in the interaction. Every command which publishes a message, creates a queue or selects a transactional mode must take place within the context of a session. Sessions are the foundation upon which the rest of AMQP rests.

A session can be seen as:

- the context in which AMQP's built-in exactly-once delivery operates (wider contexts can of course usefully be defined at application levels)
- the interface between the network protocol mapping and the model layers
- a scope for the lifetime of entities model such as entities queues, exchanges, and subscriptions
- the scope for command identifiers (see Section 3.2.1, “Sequential Identification”)

3.1.1. Session Lifetime

Sessions are not explicitly created or destroyed, in a sense a session is "always there". Rather than creating a session, a peer must attempt to "attach" to a session on its partner. The receiver of this attachment request can then look-up whether it is holding any state for this session.

State related to a session must be retained by both peers while they are attached to a session. If the session becomes detached (either through an explicit request to detach, or through the network connection between the two peers being broken) then the state attached to the session may be held for some period of time which has previously been agreed between the two peers.

3.1.2. A Transport For Commands

The AMQP model interacts by sending "commands" between the two peers. These commands are effectively sent "over" the session. As a command is handed down from the model layer to the session, it is assigned an identifier. These identifiers can then be used to correlate commands with results, or to perform synchronization on the otherwise asynchronous AMQP command stream.

3.1.3. Session as a Layer

Sessions act as the interface between the network protocol mapping and the model layers. In particular it can be used as a mechanism to ensure exactly-once delivery of a command while the session state is retained by both peers. This state (the "session state") consists of at least

- a replay buffer of full or partial commands which a peer does not yet have confirmation its partner has received, and
- an idempotency barrier - a set of commands identifier which the peer knows that it has received but cannot be sure that its partner will not attempt to re-send.

Since the session name is assigned by the application layer, there may be more state associated with it than the state detailed here. This extra state may (for example) be used to perform recovery when the session state has expired. However, in this chapter, when we talk about session state we will be referring only to the state held at the session layer.

3.2. Session Functionality

The session layer provides a number of crucial services to the model built on top of it:

- sequential identification of commands,
- confirmation that commands will be executed,
- notification when commands are complete,
- replay and recovery from network failure,
- reconciliation of state when peers fail.

3.2.1. Sequential Identification

Each command issued by a peer must be individually identified in order for the system as a whole to be able to guarantee exactly-once execution. The session layer uses a sequential numbering scheme with rollover to identify each command uniquely within a session.

The notion of identity allows for correlation between commands and results being returned asynchronously. The command identifier is made visible through the model layer of the protocol. When a result is returned from a command the command identifier is used to correlate the result to the command which gave rise to it.

3.2.2. Confirmation

In order for a peer to be able to safely discard state related to a given command, it must receive a guarantee that the command will be executed. To be slightly more precise, the sender must receive a confirmation that the command has been executed, or that its delivery has been preserved to the desired degree of durability.

In practice there are two types of messages that one may wish to send through a messaging system: durable messages and transient messages. For a transient message the general contract of the messaging system to the application is that messages may be lost if the messaging system itself loses transient state (e.g. in the case of a power outage). For a durable message, the messaging system must make the guarantee that the message will be held in the most durable store available.

The session layer handles the sending and receiving of confirmations. This allows the session layer to manage the state that it needs to hold in order to be able to recover in the case of a temporary failure of either peer, or of the transport between the peers.

Confirmations can be batched or deferred indefinitely. In particular if a peer does not require an urgent confirmation notification, the confirmation may be omitted as it is implied by the completion of the command.

3.2.3. Completion

Separate from the notion of confirmation is the notion of completion. For the purposes of synchronization and to ensure a total ordering between different sessions, it is necessary for a peer to be informed when a particular command has been completely executed. Completion necessarily implies confirmation.

Where the peer has not requested urgent notification of completion, such notifications can be deferred and batched to apply to a range of commands. This reduces the amount of network traffic. The use of sequential ids to name commands allows a compact encoding of a batch of consecutive completions.

For example, if three queues are declared in sequence, as commands 1 to 3; the server may notify of completion as follows:

```
        -------- (1) Queue.Declare    -------------->
        queue = queue1

        -------- (2) Queue.Declare    -------------->
        queue = queue2

        <-------- Session.Completed              -----
        commands = (1)

        -------- (3) Queue.Declare    -------------->
        queue = queue3

        <-------- Session.Completed              -----
        commands = (1 to 3)
```

Note that the receiver of the commands will send the entire set of commands which have completed and for which it has not been informed by the sender of the commands that the completion is known. Since this is sent as a range (e.g. 1-3) rather than a discrete set, this is not as inefficient as it may at first appear.

3.2.4. Replay and Recovery

In general an AMQP system should be expected to cope with temporary network failures, or the failure of a single node in a cluster of AMQP servers. In order to survive such failures, the session must be used to replay commands whose receipt was in doubt at the point of failure. The session layer provides the tools necessary to identify the set of commands in-doubt, and to replay them without the risk of accidental duplicate delivery.

3.3. Transport requirements

The session layer lies on top of the underlying network mapping. The session requires that the network mapping provide the following

- ordered delivery, as in no overtaking
- atomic transmission of control and data units
- detection of network failure

3.4. Commands and Controls

There are two distinct data units transferred in AMQP: commands and controls. Commands are sent *on* sessions. Commands are assigned a name and reliably delivered by the session. Controls, on the other hand, are not reliably delivered and need not be on a session, they may be considered to be communicating *about* a session.

3.4.1. Commands

Commands consist of a (`class-code`,`command-code`) pair, a `session.header`, a structured set of arguments and possibly a payload consisting of an optional sequence of message headers and an opaque message body. The command also has assigned to it a unique name as explained above.

3.4.1.1. The sync bit

The `session.header` is a structure passed on all commands. It provides a uniform mechanism for the sender of the command to request immediate notification of the completion of the command. When this "sync-bit" is set, the receiver of the command is under an obligation to send out a completion notification as soon as it becomes possible.

3.4.1.2. Results

Some commands, such as queries, return a "result". Such commands will normally be executed synchronously. A special "generic" `Execution.Result` command is used to return results. It is correlated to the command which gave rise to the result by referencing the command-id of that command. Such a result must be generated before the command which gives rise to it has been identified as "completed". If a command is specified as generating a result, it MUST always generate a result - results are never optional.

3.4.1.3. Exceptions

AMQP uses exceptions to handle errors. That is:

1. Any operational error (e.g. *message queue not found* or *insufficient access rights*) results in an exception which will destroy session state.

2. Any structural error (e.g. *invalid argument* or *bad sequence of commands*) that can be expected to recur if the same series of set data were replayed between the peers also results in an exception which will destroy session state.

3. Error conditions that cannot be ascribed to a single session, or which may be related to a transient error state on one of the peers, are dealt with by closing the connection. Closing the connection may not destroy session state (session state will only be lost at the point where the timeout for the disconnected lifetime of the session expires).

Where a failure has occurred within the model layer, the reason for the failure will be conveyed using the `Execution.Exception` command. This command informs the recipient of the reason for failure (using a three digit error code), the command id of the command which caused the error (if applicable) along with other potential useful debugging information.

Immediately after the `Execution.Exception` has been sent, the sending peer will destroy all session layer state held for this session, issue a `session.request-timeout(0)`, and finally issue a `session.detach`.

3.4.2. Controls

In contrast to commands, *controls* are unreliable with respect to sessions. This means that the session layer itself does not identify or attempt to replay controls in the face of network outage. There is no notification of the completion of controls. Because of their inherent unreliability, session controls are designed to be idempotent (i.e. repeated issuing of the same control has the same affect as one successful issuance).

While commands must travel in a strictly ordered sequence, controls may interleave or interrupt this stream. In particular a control may be sent half way through a single large command. This allows urgent controls to be sent without being held up by application data transfer.

Controls are used to manage session state.

3.5. Session Lifecycle

A session may be in either an attached or detached state. While detached the two peers may hold on to state information about the session. If re-attachment of the session is attempted, the two peers must establish which commands must be replayed to establish a consistent view of the session between the two peers. To avoid holding on to unnecessary state

while the session is detached, and to remove the necessity for the replaying of commands at the time of attachment, the peers may attempt to cleanly close the session, by establishing a consistent view before the detach leaving no in-doubt state to be rectified on re-attachement.

3.5.1. Attachment

The client attempts to attach to the session by sending a `session.attach` control. If this succeeds a `session.attached` control will be sent in the other direction.

The successful attachment to a session provides a guarantee of exclusive access to the session from a given peer, this is important since sequence numbers must originate from a single source.

3.5.2. Session layer state

As alluded to previously, a session which is attached, or which has been detached in a "non-clean" way may have associated with it some amount of session state:

1. A logical list of identified (numbered) commands recording all commands issued by this peer and for which this peer has not yet received confirmation of receipt. This list is the set of commands that would potentially have to be replayed if the connection was lost at this point in time (these commands might all still be "on the wire").

2. A set of command identifiers representing commands sent to this peer which have been confirmed or completed by this peer; but for which this peer has not yet received notification that its partner knows the command to be complete. This set forms the idempotence barrier. If, on re-attachment a command with one of these identifiers is sent, it will be ignored as this peer has already received it.

3. Command sequence counters, storing the next sequence number to assign to outgoing commands; and the sequence number to associate to the next incoming command (since sequence numbers are implicit rather than explicitly sent; both peers need to keep track of both the last outgoing and incoming commands).

3.5.3. Reliability

Session layer reliability is obtained by the retention of the session state while the session is detached (either through an explicit detach or through failure of the underlying transport). The amount of time that the session state is retained while detached is governed by a timeout value that can be set while the session is attached. If the timeout value is 0, then the session state is lost as soon as the session becomes detached. For simple implementations of AMQP it is perfectly acceptable only to allow a timeout value of 0 on all sessions. This obviously removes the ability to recover from network failure by using sessions.

3.5.4. Replay

When a session is re-attached to, the two peers negotiate to establish which commands that have been sent were actually received by their partners. It is possible that some subset of the commands that they have previously sent were "on the wire" when the session was detached. In this case, these commands need to be replayed to ensure exactly once delivery of commands to the upper layers of the protocol.

While theoretically commands can be replayed at any time while they are still in the idempotence barrier of the peer's partner there is normally no reason to replay commands while AMQP is running over a "reliable" transport such as TCP or SCTP. The only point at which it becomes necessary to replay commands is when re-attaching to a session after an unclean detachment of a session (see Section 3.6.2, “Attempting to re-attach to an existing session”).

3.6. Using Session Controls

The full set of session controls is documented in the section *Class: session* in Part II of this document. However it is useful to demonstrate how certain common activities are achieved using the session controls.

3.6.1. Attaching to a "new" session

The following interaction details how a client and server establish an attachment to a "new" session.

First the client must attempt to attach to the session

```
        -------- Session.Attach(name: <session name>, force: false)  -------------->

        <------- Session.Attached(name: <session name>)              ---------------
```

The next action each peer must take is to verify the state which its peer holds about the session. It does this by requesting (via the flush control) the peer to send:

1. the identifiers of any commands which it is expecting (if the peer has any state at all, this will include the id of the next command to be sent),

2. the identifiers of commands which have been sent to the peer, and which it has confirmed the receipt of,

3. the identifiers of commands which have been sent to the peer, and which the peer has completed execution of, but for which the peer has not yet received a signal that the completion is acknowledged
   .

If any of these lists is not empty, then the peer is holding some state about the session, and therefore it is not "new". The client should thus abort the connection (since it is expecting to create a new session and does not know how to deal with the existing session state it has discovered exists).

The following interaction shows how the state is discovered. Note this interaction occurs in both directions. The flush control should be the first control sent by each peer on the session.

```
        -------- Session.Flush(expected: true, confirmed: true,      -------------->
                               completed: true)

        <------- Session.Expected(commands: <expected-commands>,     ---------------
                                  fragments: <expected-fragments>)

        <------- Session.Confirmed(commands: <confirmed-commands>,   ---------------
                                   fragments: <confirmed-fragments>)

        <------- Session.Completed(commands: <completed-commands>,   ---------------
                                   timely-reply: true)
```

At this point if any of the received <expected-commands>, <confirmed-commands>, or <completed-commands> are not empty, then the session name we are using is known to our peer, and instead of creating a new session we are unwittingly reattaching to an existing session.

Presuming that all three command sets are empty the client can proceed to request a timeout value for the session:

```
        -------- Session.Request-Timeout(timeout: <desired-timeout>) -------------->

        <------- Session.Timeout(timeout: <timeout>)                 -------------->
```

At this point both peers can also inform their partner of the identifier at which they wish to begin identifying their command sequence:

```
        -------- Session.Command-Point(command-id: 0,                -------------->
                                       command-offset: 0)

        <------- Session.Command-Point(command-id: 0,                ---------------
                                       command-offset: 0)
```

From this point on the session is established, and commands can be sent in either direction.

3.6.2. Attempting to re-attach to an existing session

The process for attempting to re-attach to a session starts similarly to the to that of attaching to a new session (the only difference is that we try to force the attachment in case an old attachment is still believed to be active by the receiving peer):

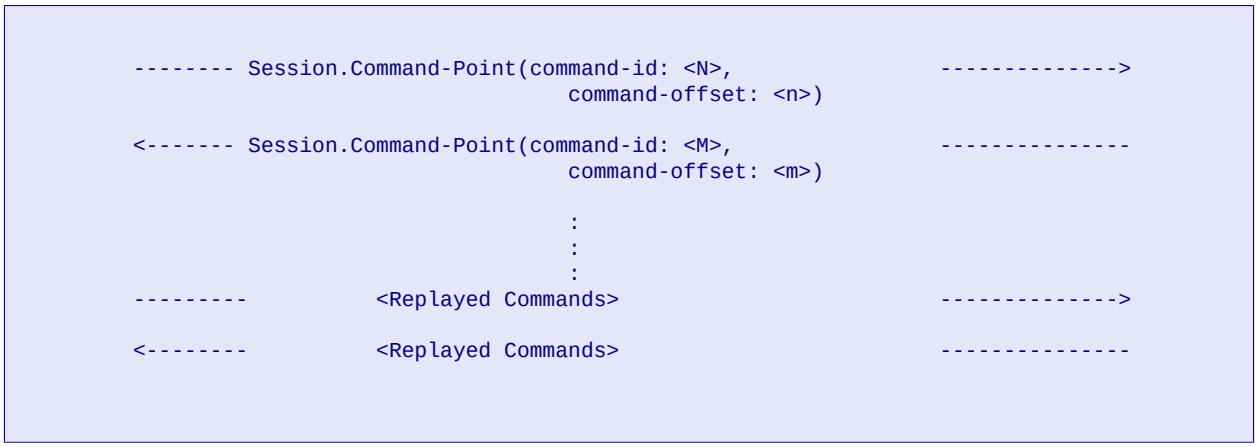
```
        -------- Session.Attach(name: <session name>, force: true)  -------------->

        <------- Session.Attached(name: <session name>)              ---------------
```

Followed by the symmetric request for state information (again this request occurs in both directions, although only one direction is shown here):

```
        -------- Session.Flush(expected: true, confirmed: true,      -------------->
                               completed: true)

        <------- Session.Expected(commands: <expected-commands>,     ---------------
                                  fragments: <expected-fragments>)

        <------- Session.Confirmed(commands: <confirmed-commands>,   ---------------
                                   fragments: <confirmed-fragments>)

        <------- Session.Completed(commands: <completed-commands>,   ---------------
                                   timely-reply: true)
```

Now, the attaching peer has a set of "pending" commands which at the time the session was previously detached, it knew it has sent, but for which it hadn't yet received confirmation of completion. For each such command it can check to see if the command is in the "confirmed" set sent by its peer. If the command is in the confirmed set then it can be removed from the "pending" list. The remaining commands in the "pending" list will have to be replayed.

Based on this information the peer can now set the command-point to the beginning of the replay list, then replay the commands.

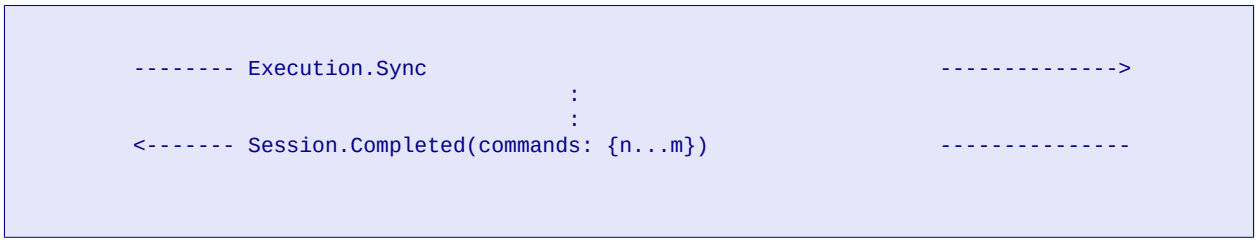


Following this it can then process the confirmed and completed sets in the same way it would during normal session activity (including the sending of session.known-completed controls).

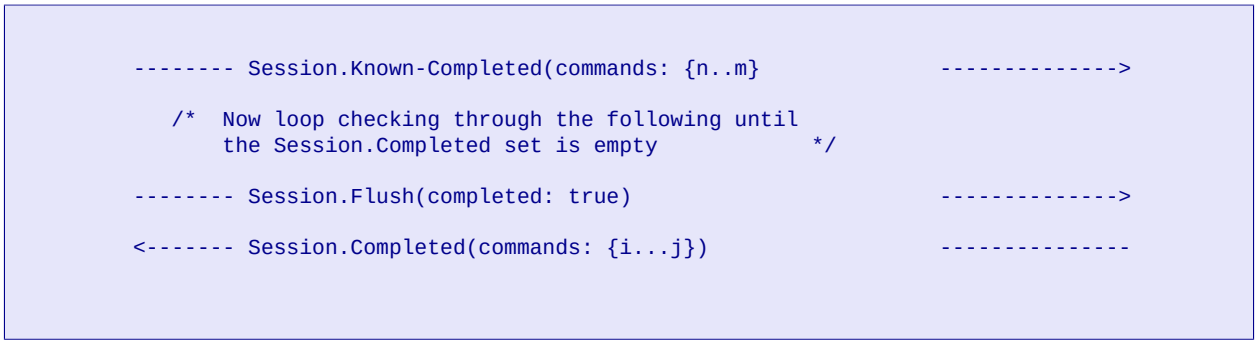
3.6.3. Detaching cleanly

If detaching from a session is planned, it is polite for both peers to minimize the amount of state that the other has to retain. Thus the peers should first attempt to quiesce the session before issuing the detach. A quiesce is performed in the following manner:

First the peer sends an `execution.sync` command to force the sending of outstanding `session.completed` controls once all current commands have been executed.



Next the peer processes the `session.completed`s and sends the `session.known-completed` responses. Once the peer is sure that its partner has received all the `known-completed` controls, it can be sure that the partner will have minimal state to hold on to. To ascertain that the `known-completed` controls have been received, it can repeatedly issue `session.flush` controls until the returned `session.completed` set is empty.



If the process of detaching is truly clean, then it will have to have been agreed at a higher level - at the AMQP model layer both peers will have been put into a state where no new commands will spontaneously be generated (e.g. all subscriptions will have been canceled). Both peers will perform this dialog simultaneously, and can work out when

they have arrived at a state where neither have any commands in doubt. At that point a detach can be carried out by either party, and the peers will have minimal (or if they choose, zero) state to maintain.

3.6.4. Closing

To close a session, you need to ensure that the peer you are communicating with retains no state. The simplest way to do this is to request that the session timeout be set to 0, wait for confirmation of this, and then detach. To cleanly close, the session should be quiesce before detaching. If there is outstanding session state when the session is closed, the session is effectively being aborted and this will most likely result in an error at the higher layers.

Part II. Specification

Table of Contents

4. Transport 40
4.1. IANA Port Number 40
4.2. Protocol Header 40
4.3. Version Negotiation 40
4.4. Framing 41
4.4.1. Assemblies, Segments, and Frames 41
4.4.2. Channels and Tracks 42
4.4.3. Frame Format 43
4.5. SCTP 44
5. Formal Notation 45
5.1. Docs and Rules 45
5.2. Types 46
5.3. Structs 47
5.4. Domains 50
5.4.1. Enums 51
5.5. Constants 51
5.6. Classes 52
5.6.1. Roles 52
5.7. Controls 53
5.7.1. Responses 54
5.8. Commands 54
5.8.1. Results 55
5.8.2. Exceptions 55
5.9. Segments 56
5.9.1. Header Segment 56
5.9.2. Body Segment 57
6. Constants 58
7. Types 59
7.1. Fixed width types 59
7.1.1. bin8 59
7.1.2. int8 60
7.1.3. uint8 61
7.1.4. char 62
7.1.5. boolean 63
7.1.6. bin16 64
7.1.7. int16 65
7.1.8. uint16 66
7.1.9. bin32 67
7.1.10. int32 68
7.1.11. uint32 69
7.1.12. float 70
7.1.13. char-utf32 71
7.1.14. sequence-no 72
7.1.15. bin64 73
7.1.16. int64 74
7.1.17. uint64 75
7.1.18. double 76
7.1.19. datetime 77
7.1.20. bin128 78
7.1.21. uuid 79
7.1.22. bin256 80

7.1.23. bin512 81
7.1.24. bin1024 82
7.1.25. bin40 83
7.1.26. dec32 84
7.1.27. bin72 85
7.1.28. dec64 86
7.1.29. void 87
7.1.30. bit 88
7.2. Variable width types 90
7.2.1. vbin8 90
7.2.2. str8-latin 91
7.2.3. str8 92
7.2.4. str8-utf16 93
7.2.5. vbin16 94
7.2.6. str16-latin 95
7.2.7. str16 96
7.2.8. str16-utf16 97
7.2.9. byte-ranges 98
7.2.10. sequence-set 99
7.2.11. vbin32 100
7.2.12. map 101
7.2.13. list 102
7.2.14. array 103
7.2.15. struct32 104
7.3. Mandatory Types 106
8. Domains 107
8.1. segment-type 107
8.2. track 107
8.3. str16-array 108
9. Control Classes 110
9.1. connection 110
9.1.1. connection.close-code 111
9.1.2. connection.amqp-host-url 112
9.1.3. connection.amqp-host-array 113
9.1.4. connection.start 114
9.1.5. connection.start-ok 116
9.1.6. connection.secure 117
9.1.7. connection.secure-ok 118
9.1.8. connection.tune 119
9.1.9. connection.tune-ok 120
9.1.10. connection.open 122
9.1.11. connection.open-ok 123
9.1.12. connection.redirect 124
9.1.13. connection.heartbeat 125
9.1.14. connection.close 126
9.1.15. connection.close-ok 127
9.2. session 129
9.2.1. Rules 130
9.2.2. session.header 130
9.2.3. session.command-fragment 131
9.2.4. session.name 132
9.2.5. session.detach-code 133
9.2.6. session.commands 134
9.2.7. session.command-fragments 135

9.2.8. session.attach 136
9.2.9. session.attached 137
9.2.10. session.detach 138
9.2.11. session.detached 139
9.2.12. session.request-timeout 140
9.2.13. session.timeout 141
9.2.14. session.command-point 142
9.2.15. session.expected 143
9.2.16. session.confirmed 144
9.2.17. session.completed 145
9.2.18. session.known-completed 146
9.2.19. session.flush 147
9.2.20. session.gap 148
10. Command Classes 150
10.1. execution 150
10.1.1. execution.error-code 150
10.1.2. execution.sync 151
10.1.3. execution.result 152
10.1.4. execution.exception 153
10.2. message 155
10.2.1. Rules 157
10.2.2. message.delivery-properties 158
10.2.3. message.fragment-properties 160
10.2.4. message.reply-to 161
10.2.5. message.message-properties 162
10.2.6. message.destination 164
10.2.7. message.accept-mode 165
10.2.8. message.acquire-mode 166
10.2.9. message.reject-code 167
10.2.10. message.resume-id 168
10.2.11. message.delivery-mode 169
10.2.12. message.delivery-priority 170
10.2.13. message.flow-mode 171
10.2.14. message.credit-unit 172
10.2.15. message.transfer 173
10.2.16. message.accept 175
10.2.17. message.reject 176
10.2.18. message.release 177
10.2.19. message.acquire 178
10.2.20. message.resume 179
10.2.21. message.subscribe 180
10.2.22. message.cancel 182
10.2.23. message.set-flow-mode 183
10.2.24. message.flow 184
10.2.25. message.flush 185
10.2.26. message.stop 186
10.3. tx 188
10.3.1. Rules 188
10.3.2. tx.select 188
10.3.3. tx.commit 189
10.3.4. tx.rollback 190
10.4. dtx 192
10.4.1. Rules 193
10.4.2. dtx.xa-result 193

10.4.3. dtx.xid 194
10.4.4. dtx.xa-status 195
10.4.5. dtx.select 196
10.4.6. dtx.start 197
10.4.7. dtx.end 199
10.4.8. dtx.commit 201
10.4.9. dtx.forget 203
10.4.10. dtx.get-timeout 204
10.4.11. dtx.prepare 205
10.4.12. dtx.recover 207
10.4.13. dtx.rollback 208
10.4.14. dtx.set-timeout 210
10.5. exchange 212
10.5.1. Rules 212
10.5.2. exchange.name 213
10.5.3. exchange.declare 214
10.5.4. exchange.delete 217
10.5.5. exchange.query 218
10.5.6. exchange.bind 219
10.5.7. exchange.unbind 222
10.5.8. exchange.bound 223
10.6. queue 226
10.6.1. Rules 226
10.6.2. queue.name 226
10.6.3. queue.declare 227
10.6.4. queue.delete 230
10.6.5. queue.purge 231
10.6.6. queue.query 232
10.7. file 234
10.7.1. Rules 235
10.7.2. file.file-properties 235
10.7.3. file.return-code 236
10.7.4. file.qos 237
10.7.5. file.qos-ok 238
10.7.6. file.consume 239
10.7.7. file.consume-ok 241
10.7.8. file.cancel 242
10.7.9. file.open 243
10.7.10. file.open-ok 244
10.7.11. file.stage 245
10.7.12. file.publish 246
10.7.13. file.return 248
10.7.14. file.deliver 249
10.7.15. file.ack 250
10.7.16. file.reject 251
10.8. stream 253
10.8.1. Rules 254
10.8.2. stream.stream-properties 254
10.8.3. stream.return-code 255
10.8.4. stream.qos 256
10.8.5. stream.qos-ok 257
10.8.6. stream.consume 258
10.8.7. stream.consume-ok 260
10.8.8. stream.cancel 261

10.8.9. stream.publish 262
10.8.10. stream.return 264
10.8.11. stream.deliver 265
11. The Model 267
11.1. Exchanges 267
11.1.1. Mandatory Exchange Types 267
11.1.2. Optional Exchange Types 268
11.1.3. System Exchanges 270
11.1.4. Implementation-defined Exchange Types 271
11.1.5. Exchange Naming 271
11.2. Queues 271
11.2.1. queue_naming 271
12. Protocol Grammar 272
12.1. Augmented BNF Rules 272
12.2. Grammar 272

4. Transport

4.1. IANA Port Number

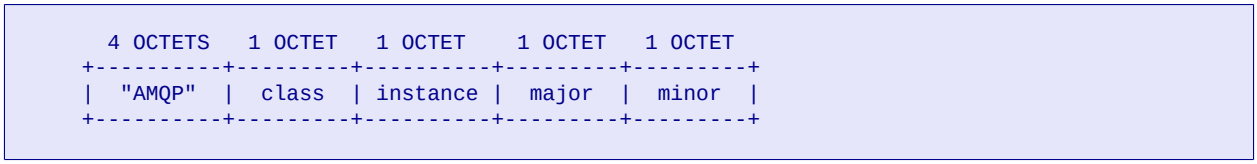
The standard AMQP port number has been assigned by IANA as 5672 for TCP, UDP and SCTP.

There is currently no UDP mapping defined for AMQP. The UDP port number is reserved for future transport mappings.

4.2. Protocol Header

Prior to sending any frames on a connection, each peer MUST start by sending a protocol header that indicates the protocol version used on the connection. [1]

This is an 8-octet sequence:



The protocol header consists of the upper case letters "AMQP" followed by:

1. The protocol class, which is 1 for all AMQP protocols.
2. The protocol instance, which is:

Instance	Value
AMQP over TCP/IP	1
AMQP over SCTP/IP	2

3. The major version of the protocol, used in accordance with Part II, “Specification”.
4. The minor version of the protocol, used in accordance with Part II, “Specification”.

4.3. Version Negotiation

The protocol negotiation model is compatible with 1) existing protocols such as HTTP that initiate a connection with a constant text string, and 2) firewalls that sniff the start of a protocol in order to decide what rules to apply.

An AMQP client and server agree on a protocol and version as follows:

- When the client opens a new socket connection to an AMQP server, it MUST send a protocol header with the client's preferred protocol version.
- If the requested protocol version is supported, the server MUST send its own protocol header with the requested version to the socket, and then implement the protocol accordingly.
- If the requested protocol version is *not* supported, the server MUST send a protocol header with a *supported* protocol version and then close the socket.

[1] Note that for protocol versions prior to 0-10 the protocol header was sent by the client only. An implementation that wishes to support these versions in addition to 0-10 should respond in a manner consistent with the requested version.

- If the server can't parse the protocol header, the server MUST send a valid protocol header with a supported protocol version and then close the socket.

Based on this behavior a client can discover which protocol and versions a server supports:

- An AMQP client MAY detect the server protocol version by attempting to connect with its highest supported version and reconnecting with a lower version received back from the server.
- An AMQP server MUST accept the AMQP protocol as defined by class = 1, instance = 1.

Examples:

```
Client sends:   Server responds:                  Comment:
-------------   ------------------------------    ----------------------------------------
AMQP%d1.1.0.10   AMQP%d1.1.0.10<start connection>   Server accepts connection for:
                                                   Class:1(AMQP), Instance:1(TCP), Vers:0-10

AMQP%d2.0.1.1    AMQP%d1.1.0.10<close connection>   Server rejects connection for:
                                                   Class:2(?), Instance:0(?), Vers:1-1
                                                   Server responds it supports:
                                                   AMQP, TCP, Vers:0-10

HTTP             AMQP%d1.1.0.10<close connection>   Server rejects connection for: HTTP
                                                   Server responds it supports:
                                                   AMQP, TCP, Vers:0-10
```

Please note that the above examples use the literal notation defined in RFC 2234 for non alphanumeric values.

4.4. Framing

4.4.1. Assemblies, Segments, and Frames

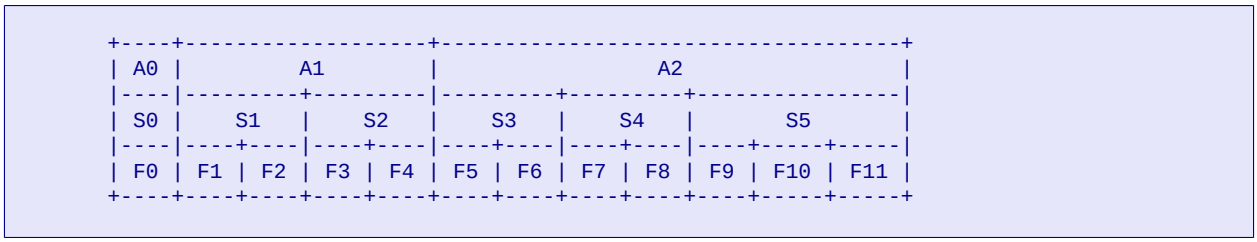
An *assembly* is the largest structural unit directly represented by the AMQP framing system. AMQP encodes each semantic unit (control or command) into exactly one assembly. Each assembly is divided into one or more *segments*. AMQP uses segments to represent distinct syntactic units (e.g. header vs body) within a given semantic unit. Finally, each segment is divided into one or more *frames*. A frame is the atomic unit of transmission within AMQP.

Assemblies and segments have no fixed size limit. Frames are always limited by the maximum frame size permitted by the transport mapping. In addition, for a given connection, frame sizes are also limited by a per connection maximum negotiated between the endpoints.

For the wire level encoding, the three-level structure is flattened into a single uniform frame representation. In addition to a payload, each frame carries an additional four flags. Two flags mark the position of the payload within the segment, and the other two mark the position of the segment within the assembly.

flag	**meaning**
first-segment	The frame is part of the first segment in the assembly.
last-segment	The frame is part of the last segment in the assembly.
first-frame	The frame is the first in the segment.
last-frame	The frame is the last in the segment.

Based on these four flags, the segment and assembly boundaries, as well as the full payload can be reconstructed from a sequence of frames as depicted below.



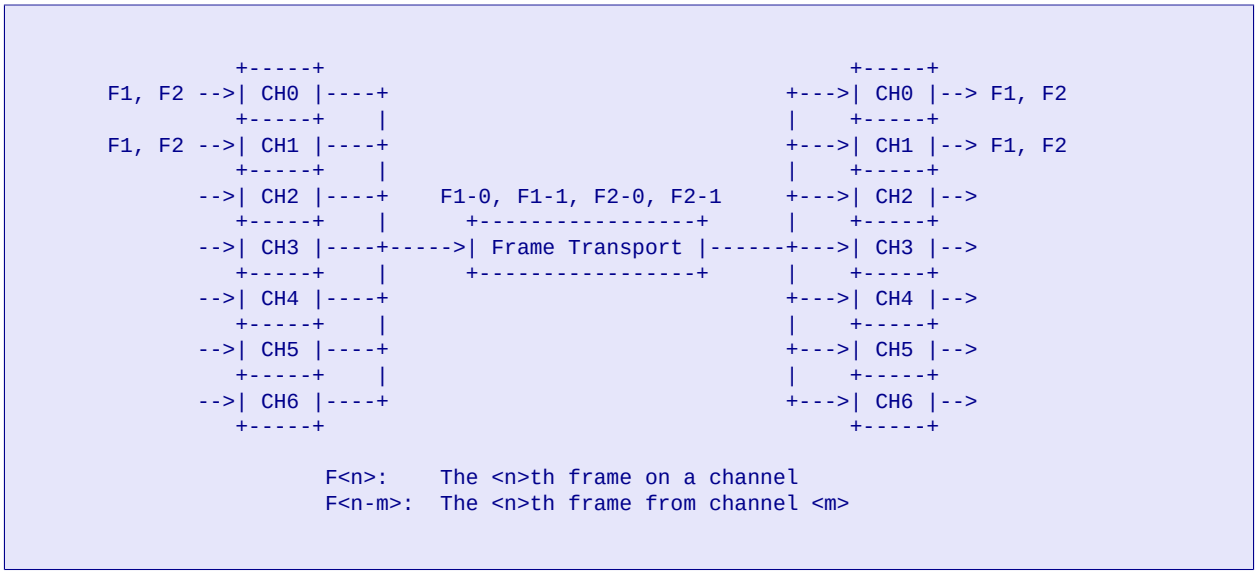
A<n>	the nth assembly
S<n>	the nth segment
F<n>	the nth frame

The length of a segment is determined by the accumulated payload length of all the contained frames. Likewise the length of an assembly is determined from the accumulated length of the contained segments. Note that the size of the frame header itself must be subtracted from the frame size in order to correctly calculate the payload length.

4.4.2. Channels and Tracks

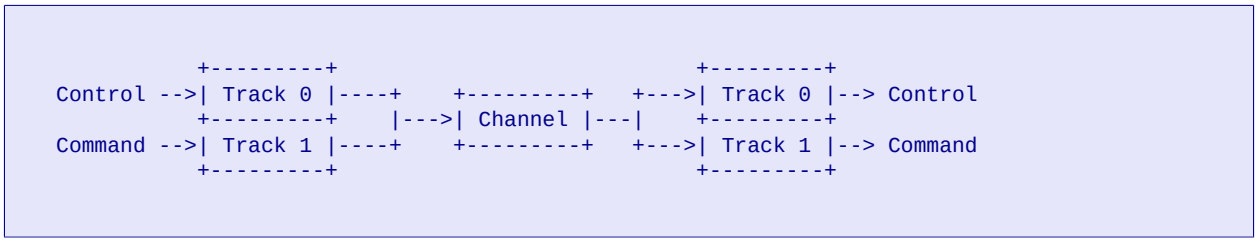
AMQP framing permits multiple independent dialogs to share a single frame transport (connection). Each frame includes a number that uniquely identifies the dialog to which a frame belongs. This number divides a single frame transport into distinct *channels*. There is no order preserved between frames sent on different channels. Even when the frame transport provides a total ordering, an implementation MAY service frames on distinct channels in any desired order.

The figure below depicts frames from two separate channels traveling along on a single frame transport divided into many channels:



Within each channel there is a further division of frames by *track*. Like channels within the frame transport, tracks permit multiple concurrent dialogs within a single channel. However, unlike channel multiplexing, the order of frames within a channel is retained regardless of the track number. An implementation MUST service frames on distinct tracks within a single channel according to the total ordering provided by the channel. Together, the (channel, track) pair provides the sequencing used to reconstruct assembly payloads from the fragments transmitted on a given frame transport.

The AMQP frame format permits up to 64K channels, and up to 16 tracks. The specification only defines two of the 16 tracks. The remaining tracks are reserved. AMQP channels provide the frame transport for sessions. AMQP defines two tracks within a session to distinguish commands from controls. Controls are carried on track zero, and commands are carried on track one. This permits controls to be sent in-between consecutive frames of a single command. This prevents transmission of large multi-frame commands from blocking the control dialog between the communication endpoints. In many respects this is an ordinary multiplexing of a channel into two tracks, however because frames and the constructed assemblies MUST be processed in the order defined by the channel, it is possible to define the semantics of controls to operate with respect to a well-defined point in the command track.

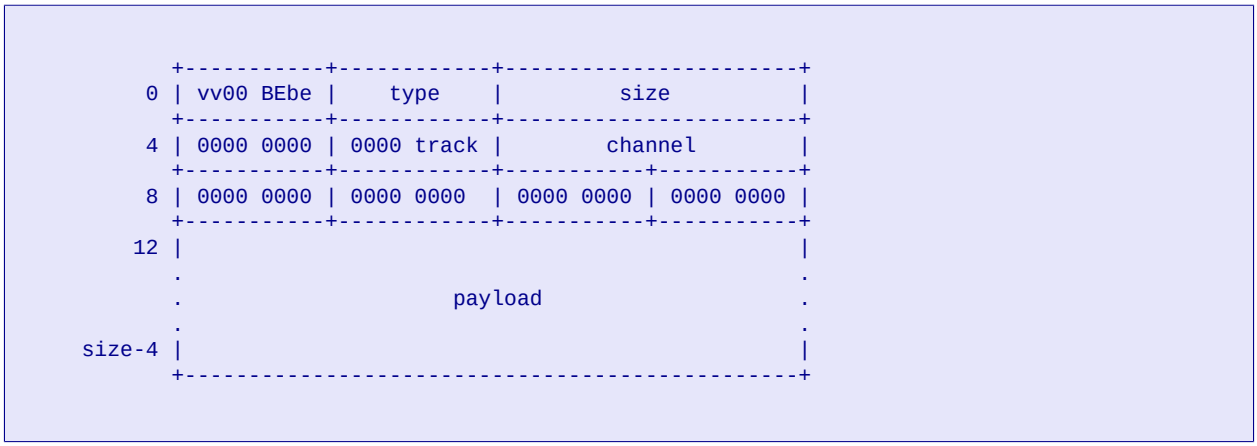


Channels and tracks provide full-duplex communication. Where two way dialogs are specified, the defined responses, unless otherwise specified, are carried on the same channel and track as the initiating requests.

- An AMQP peer MUST permit communication on channel 0 for any established connection. This channel always exists and can never be negotiated away.
- An AMQP peer SHOULD support multiple channels. The maximum number of channels is defined at connection negotiation, and a peer MAY negotiate this down to 1.
- Each peer SHOULD balance the traffic on all open channels in a fair fashion. This balancing can be done on a per-frame basis, or on the basis of amount of traffic per channel. A peer SHOULD NOT allow one very busy channel to starve the progress of a less busy channel.

4.4.3. Frame Format

All frames consist of a 12 octet header, and a payload of variable size:



Field	Identifier	Description
vv	frame-format-version	Set to 00 for this framing format.
0	reserved	All reserved bits MUST be 0.
B	first-segment	Set to 1 for the first (or only) segment of an assembly, 0 otherwise.

Field	Identifier	Description
E	last-segment	Set to 1 for the last (or only) segment of an assembly, 0 otherwise.
b	first-frame	Set to 1 for the first (or only) frame of a segment, 0 otherwise.
e	last-frame	Set to 1 for the last (or only) frame of a segment, 0 otherwise.
type	segment-type	Indicates the format and purpose of a segment.
size	frame-size	The total frame size. This includes the frame header. If the size < 12, the frame is malformed.
track	track-number	The track to which the frame belongs.
channel	channel-number	The channel to which the frame belongs.

4.5. SCTP

SCTP provides additional capabilities beyond TCP. This section describes how AMQP implementations should take advantage of these.

SCTP manages transmission through individual data items (called "messages" in the SCTP specification), each of which can be fragmented or bundled, depending on the path MTU size. The specification requires that the protocol header be sent as a single SCTP message, and then defines a one-to-one mapping between an AMQP frame and an SCTP message. This means that the SCTP stack's EOM notification to the SCTP application layer directly corresponds to the AMQP end-of-frame condition.

SCTP allows for multiple concurrent streams of data on the same association. The data within a stream is delivered to the application in order, but without respect to data in other streams, so one stream will not block another due to packet loss. AMQP has a concept of channels, which can benefit from the property of one channel not being able to block another due to packet loss.

Since SCTP streams are unidirectional, and AMQP channels are bidirectional, the specification maps one AMQP channel to two SCTP streams, one in each direction. The SCTP stream ID in each direction corresponding to the same AMQP channel will have the same value (i.e. AMQP channel {X} maps to two SCTP streams {Y, Y} where one is inbound and one is outbound). It is always true that a given AMQP channel maps to a single set of two SCTP streams. However, the inverse is not necessarily true — a given set of two SCTP streams could have multiple AMQP channels mapped to it. (In the degenerate case, where you might have only 2 SCTP streams on the whole association, it becomes similar to a full-duplex TCP connection).

To avoid server resource usage handling unsupported protocol versions (common when a protocol is being upgraded from one version to another), SCTP's Adaptation Layer Indicator should be used to allow for early rejection of unsupported versions, before an association is established. The server will set its Adaptation Layer Indicator to a value assigned by IANA. This value will correspond to AMQP 0-10.

The SCTP Payload Protocol ID field will contain a value assigned by IANA to indicate AMQP. This is a generic AMQP indication, not a version indicator.

SCTP stream 0 in each direction is used for all communication before framing is set up (i.e. for the SASL negotiation). After framing is set up, Stream 0 is used for all communication that is defined to happen on AMQP channel 0.

Note

The IANA assigned constants referred to in this section are not yet defined. When available, they will be included in a future publication of the specification.

5. Formal Notation

AMQP semantics are defined in terms of *types*, *structs*, *domains*, *constants*, *controls*, and *commands*. AMQP formally defines the semantics of each protocol construct using an XML notation. Each kind of construct is defined with a similarly named element. These definitions are grouped into related *classes* of functionality.

Construct	Definition	Notation
Type:	a set of values with formally defined operations and encoding	<type name="..." ... > ... </type>
Struct:	a compound type of named fields	<struct name="..." ... > ... </struct>
Domain:	a restricted type	<domain name="..." ... > ... </domain>
Constant:	a constant value	<constant name="..." ... > ... </constant>
Control:	a one-way instruction	<control name="..." ... > ... </control>
Command:	an acknowledged instruction	<command name="..." ... > ... </command>

5.1. Docs and Rules

The semantics of each AMQP construct are formally defined by documentation and rules that appear within the definition of the given construct. Documentation is expressed with the doc element:

```
<doc title="..."
     type="...">
  ...
</doc>
```

Attributes of a doc element:

title If present, this attribute contains a title for the contained documentation.

type Permitted values: grammar, scenario, picture, bnf

If present, this attribute indicates the type of the contained documentation. This primarily serves as a formatting hint for processing tools.

The doc elements of type grammar use the following notation:

1. 'S:' indicates a control or command sent from the server to the client
2. 'C:' indicates a control or command sent from the client to the server
3. '*:' indicates a control or command initiated from either peer
4. 'R:' indicates a control or command sent by the partner of the initiating peer
5. [...] means zero or one instance
6. +term or +(...) expression means '1 or more instances'
7. *term or *(...) expression means 'zero or more instances'.

Rules

Rules are used to formally name a particular aspect of the semantics of a given construct:

```
<rule  name="..."
      label="...">
  <doc ... > ... </doc>
  ...
</rule>
```

Attributes of a rule:

name The name of the rule. This is unique within the defining context.

label A sentence fragment containing a short description of the rule.

5.2. Types

Each AMQP *type* defines a format for encoding a particular kind of data. Additionally, most AMQP types are assigned a unique code that functions as a descriminator when more than one type may be encoded in a given position.

AMQP types broadly fall into two categories: fixed-width and variable-width. Variable-width types are always prefixed by a byte count of the encoded size, excluding the bytes required for the byte count itself.

Unless otherwise specified, AMQP uses network byte order for all numeric values.

AMQP types are formally defined with the type element:

```
<type name="..."
      code="...">
  <doc type="bnf">
    ...
  </doc>
</type>
```

Attributes of a type definition:

name The name of the type. This is unique among all top-level AMQP constructs.

code The type code.

A type code is a single octet which may hold 256 distinct values. Ranges of types are mapped to specific sizes of data so that an implementation can easily skip over any data types not natively supported.

Code	Category	Format
0x00 - 0x0F	Fixed width.	One octet of data.
0x10 - 0x1F	Fixed width.	Two octets of data.
0x20 - 0x2F	Fixed width.	Four octets of data.
0x30 - 0x3F	Fixed width.	Eight octets of data.

Code	Category	Format
0x40 - 0x4F	Fixed width.	Sixteen octets of data.
0x50 - 0x5F	Fixed width.	Thirty-two octets of data.
0x60 - 0x6F	Fixed width.	Sixty-four octets of data.
0x70 - 0x7F	Fixed width.	One hundred twenty-eight octets of data.
0x80 - 0x8F	Variable width.	One octet of size, 0-255 octets of data.
0x90 - 0x9F	Variable width.	Two octets of size, 0-65535 octets of data.
0xA0 - 0xAF	Variable width.	Four octets of size, 0-4294967295 octets of data.
0xB1 - 0xBF	Reserved	
0xC0 - 0xCF	Fixed width.	Five octets of data.
0xD0 - 0xDF	Fixed width.	Nine octets of data.
0xE0 - 0xEF	Reserved	
0xF0 - 0xFF	Fixed width.	Zero octets of data.

The particular type code ranges were chosen with the following rationale in mind:

```
       Bit:  7    6    5    4    3    2    1    0
       ----------------------------------------
             0  |   fix-exp    |     subtype
             1    0  | var-exp |     subtype
             1    1  | fix-odd |     subtype
       ----------------------------------------

  fix-exp = log2(size of fixed width type)
  var-exp = log2(size of size of variable width type) (Note: 11 is reserved)
  fix-odd = 00, for 5-byte fixed width
            01, for 9-byte fixed width
            10, reserved
            11, for 0-byte fixed width
```

5.3. Structs

An AMQP *struct* defines a *compound* type. That is a type whose format is defined entirely in terms of other types. The simplest kind of struct consists of an ordered sequence of encoded *field* data for a well known set of fields. Each field is encoded according to the type definition for that field. Several options may be used to augment this encoding:

- A struct may be *packed*, in which case logically absent fields are omitted from the encoded data. For these structs, the field data is directly preceded by either 1, 2 or 4 octets of packing flags to indicate which fields are present. Note that although the notation and encoding scheme described here would function equally well for any number of packing flags, the structs defined by the specification only make use of 0, 1, 2, and 4 octets worth of packing flags.

- A struct may be *coded*, in which case the field data, and any packing flags are preceded by a 2 octet code that uniquely identifies the struct definition. This includes the number of packing flags (if any) as well as the index, name, and type of each field.

- A struct may be *sized*, in which case the field data, any packing flags, and the struct code (if present) are all prefixed by either a 1, 2, or 4 octet unsigned byte count.

The general layout of all structs is defined in the following BNF:

```
       struct = [struct-size] [class-code struct-code] [packing-flags] data

  struct-size = uint8 / uint16 / uint32
   class-code = uint8 ; zero for top-level structs
  struct-code = uint8
packing-flags = uint8 / 2 uint8 / 4 uint8

         data = *OCTET ; encoded field values as defined by the
                       ; order and type of the fields specified
                       ; in the struct definition
```

A struct is formally defined with the struct element:

```
      <struct name="..."
              size="..."
              code="..."
              pack="...">
        <field name="..."
               type="..."
               required="..." />
        ...
      </struct>
```

The order of the fields within the struct definition defines the order that field data is encoded.

Attributes of a struct definition:

size Permitted values: 0, 1, 2, 4

If a non-zero size width is specified in the struct definition, the encoded struct is preceded by a byte count of the indicated width. In addition to the encoded field data, this byte count MUST include the struct code and packing flags if present.

The size field MUST be omitted if no size is specified or size="0" is specified in the struct definition.

code Permitted values: 0-255

If a code is included in the struct definition, the specified value MUST be preceded with the class-code, and the resulting two octets encoded prior to the encoded field data and packing flags (if any), but after the size (if any).

The class-code and struct-code MUST be omitted from encoded structs if no code or code="none" is specified in the struct definition.

The value of the combined class-code and struct-code is unique to any given struct definition, and MAY be used when decoding to determine which struct definition has been encoded.

pack Permitted values: 0, 1, 2, 4

If a non-zero pack width is specified in the struct definition, the encoded field data MUST be preceded by the indicated number of octets. The n^{th} octet contains packing flags for the n^{th} group of 8 fields specified in the struct definition. Within each octet the fields map in order from the least significant bit to the most significant bit.

If a packing flag is set the corresponding field MUST be included in the encoded data.

If a packing flag is *not* set the corresponding field MUST NOT be included in the encoded data.

If the struct has fewer properties than packing flags the extra packing flags are reserved for future extension of the struct and MUST be set to zero.

Attributes of a field definition:

name — The name of the field. This uniquely identifies the field within the struct.

type — The type attribute identifies how the field data is to be encoded. This could refer to a primitive type, another struct definition, or a domain definition.

required — Permitted values: true, false

If this attribute is true, then the given field MUST always be present. If a struct is parsed and the field is absent, then the whole struct SHOULD be considered malformed.

Examples

Simple structs consist only of the encoded field data:

```
<struct name="error-info" pack="0">
  <field name="code" type="uint16" />
  <field name="text" type="str16" />
</struct>
```

```
+--------+--------+
|  code  |  text  |
+--------+--------+
| uint16 |  str16 |
+--------+--------+
```

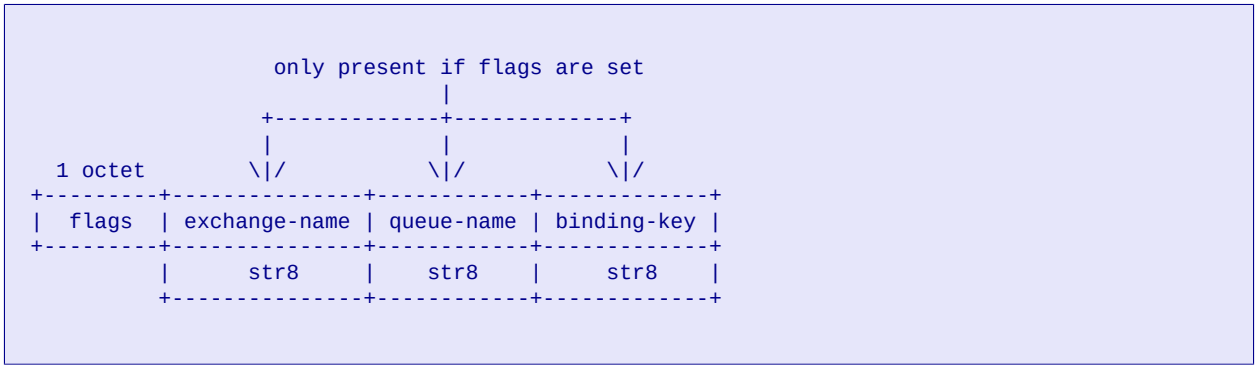
A sized struct prefixes the encoded representation with a byte count:

```
<struct name="address" size="2">
  <field name="host" type="str8"/>
  <field name="port" type="uint16" />
</struct>
```

```
+---------+---------------------+
| 2 octet |       n octets      |
+---------+----------+----------+
|    n    |   host   |   port   |
+---------+----------+----------+
          |   str8   |  uint16  |
          +----------+----------+
```

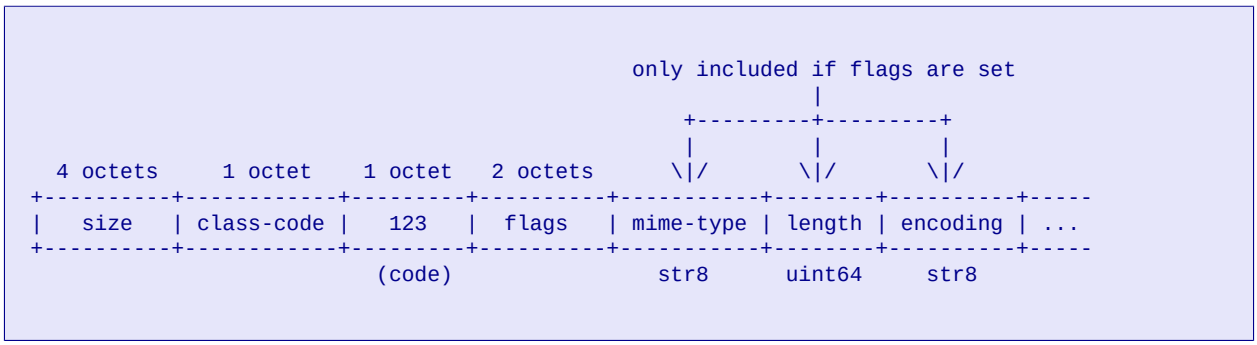
Packed structs omit logically absent fields from the wire encoding:

```
      <struct name="binding" pack="1">
        <field name="exchange-name" type="str8"/>
        <field name="queue-name" type="str8"/>
        <field name="binding-key" type="str8"/>
      </struct>
```



These encoding options can be combined:

```
      <struct name="content-headers" size="4" pack="2" code="123">
        <field name="mime-type" type="str8"/>
        <field name="length" type="uint64"/>
        <field name="encoding" type="str8"/>
        ...
      </struct>
```



5.4. Domains

An AMQP *domain* defines a new type with a format identical to another type, but with a restricted range of values. In some cases a closed set of permitted values is specified with an *enum*, and in other cases an open set of values is specified with docs and rules.

An AMQP domain is formally defined with the domain element:

```
<domain name="..."
        type="...">
  <doc>
    ...
  </doc>
  ...
</domain>
```

Attributes of a domain definition:

name The name of the domain. This is unique within the defining context.

type The type that defines the format for this domain. This could refer to a primitive type, a struct, or another domain definition.

5.4.1. Enums

If a domain definition includes an *enum*, the values permitted by the domain are restricted to a set of explicitly named *choices*:

```
<domain name="..."
        type="...">
  <doc>
    ...
  </doc>
  ...
  <enum>
    <choice name="..."
            value="..."/>
    ...
  </enum>
</domain>
```

Attributes of a choice:

name The name of the choice. This uniquely identifies the choice within the enum.

value The value of the choice. This is any value that can be represented by the type of the enum and is distinct from other choices associated with the enum.

5.5. Constants

An AMQP *constant* is a fixed value referenced throughout the specification. Constants are formally defined with the constant element:

```
<constant name="..."
          value="...">
  <doc>
    ...
  </doc>
</constant>
```

Attributes of a constant definition:

name The name of the constant. This is unique among all top-level AMQP constructs.

value The value of the constant.

5.6. Classes

An AMQP *class* groups together related command, control, struct, and domain definitions. Classes function as a namespace for those constructs defined within. Additionally, each class is assigned a code that forms the high byte of any control-codes, command-codes, or struct-codes associated with the contained definitions.

A class is formally defined with the following notation:

```
<class name="..."
       code="...">
  ...
  <role .../>
  <struct ... />
  <domain ... />
  <control ... />
  <command ... />
  ...
</class>
```

name The name of the class. This is unique among all top-level AMQP constructs.

code Permitted values: 1-255

An octet that uniquely identifies the class. The special value zero is reserved to identify globally defined constructs.

5.6.1. Roles

Each class formally defines the different roles an implementation may fulfill. These roles are referenced from within the contained control and command definitions when defining the levels of implementation optionality.

A role is formally defined with the following notation:

```
<role name="..." implement="...">
  <doc>...</doc>
</role>
```

name The name of the role. This is unique within the class.

implement Permitted values: MAY, SHOULD, MUST

Defines whether an AMQP implemention MAY, SHOULD, or MUST implement the specified role.

Each control and command formally defines its implementation requirements using the following notation:

```
<implement role="..." handle="..." />
```

role The name of a role defined within the containing class. The implementation requirement is interpreted relative to this role.

handle Permitted values: MAY, SHOULD, MUST

Defines whether an AMQP implemention implementing the specified role MAY, SHOULD, or MUST be able to receive the containing control or command.

5.7. Controls

An AMQP *control* defines the format and semantics of a one-way instruction. Because controls are one-way, in the event of transport failure they may need to be repeated until the effect of the instruction can be observed. For this reason, controls often ellicit a reply that permits the peer to observe the effect. Because even the reply may be lost to a transport failure, the semantics of controls are usually defined to be idempotent so that repeated execution of the same instruction does not cause undesirable side-effects.

An AMQP control is encoded into the control segment of an assembly as follows:

- The class-code is placed in the first octet.
- The control-code is placed in the second octet.
- The field values are then encoded as an unsized, uncoded struct with two octets of packing flags.

```
   1 OCTET       1 OCTET        2 OCTETS
+------------+--------------+---------------+------------+
| class-code | control-code | packing-flags | fields ... |
+------------+--------------+---------------+------------+
```

An AMQP control is formally defined with the control element:

```
<control name="..."
         code="...">
  ...
  <implement role="..." handle="..." />
  ...
  <field name="..."
         type="..."/>
  ...
</control>
```

Attributes of a control definition:

name The name of the control. This is unique within the defining class.

code Permitted values: 0-255

An octet that uniquely identifies the control within the class.

The field definitions define the arguments for the control. These are identical to field definitions within a struct.

5.7.1. Responses

If the effect of a control is communicated with a direct reply, the permitted response(s) are formally defined with the response element. Multiple response elements within a single control definition indicate alternative replies. Responses are always sent on the same channel as the initiating control or command.

```
<control name="..."
         code="...">
  ...
  <implement role="..." handle="..." />
  ...
  <response name="..."/>
  ...
  <field name="..."
         type="..."/>
  ...
</control>
```

Attributes of a response definition:

name The name of the reply control.

5.8. Commands

An AMQP *command* defines the format and semantics of an acknowledged instruction. Commands are *not* assumed to be idempotent, therefore each command is assigned a sequential command-id prior to transmission, and the receiving peer MUST execute each command in order, and exactly once regardless of how many times it is received.

An AMQP command is encoded into the command segment of an assembly as follows:

- The class-code is placed into the first octet.
- The command-code is placed into the second octet.
- The session.header struct is encoded after the class-code and command-code.
- The command arguments are then encoded as an unsized, untype struct with two octets of packing flags.

```
   1 OCTET       1 OCTET                        2 OCTETS
+------------+--------------+----------------+---------------+------------+
| class-code | control-code | session.header | packing-flags | fields ... |
+------------+--------------+----------------+---------------+------------+
```

A command is formally defined with the command element:

```
<command name="..."
         code="...">
  <field name="..."
         type="..."/>
  ...
</command>
```

5.8.1. Results

When commands produce results during execution, the result is defined as a struct and carried by the execution.result command. The result is always sent on the same channel as the initiating command. The format of a command result is formally defined with the optional result element:

```
<command name="..."
         code="...">
  <field name="..."
         type="..."/>
  ...

  <result type="..."/>
</command>
```

Attributes of a result definition:

type Identifies the format of the result. This MUST refer to a coded struct with size="4".

A result definition may also contain the struct definition rather than reference it by name:

```
<command name="..."
         code="...">
  <field name="..."
         type="..."/>
  ...

  <result>
    <struct name="..."
            size="4"
            type="..."
            pack="...">
      <field name="..."
             type="..."/>
      ...
    </struct>
  </result>
</command>
```

5.8.2. Exceptions

When exceptional conditions occur during command execution, the execution.exception command is used to indicate that an exception has occured. The exception command is sent on the channel where the problem occurred. The

execution.error-code enum defines error codes for all the defined error conditions that can occur during command execution.

Exceptional conditions are formally defined with the exception element. These may appear within a command or field definition:

```
<command name="..."
         code="...">
  <exception       name="..."
             error-code="..."/>
  ...
  <field name="..."
         type="...">
    <exception       name="..."
               error-code="..."/>
  </field>
  ...
</command>
```

Attributes of an exception:

name — The name of the exception. This uniquely identifies the exceptional condition within the field or command.

error-code — The name of a choice defined within the execution.error-code enum.

5.9. Segments

Specific commands or controls may be defined to carry additional segments. In addition to command segments and control segments, AMQP defines header and body segments that are used for carrying message content. If permitted, the presence, contents, and order of these additional segments is formally defined with the segments element:

```
<command name="..." ... >
  ...
  <segments>
    <header ... > ... </header>
    <body ... />
  </segments>
</command>
```

5.9.1. Header Segment

The contents of a header segment consists of a set of sized (size="4"), coded, packed structs. These are sequentially encoded into the segment in an undefined order. When parsing the header segment, an implementation MUST assume that the entries may be in any order, and intermediaries MAY reorder or insert additional entries. A header segment MUST include at most one instance of each type.

Should an intermediary encounter a struct entry with an unrecognized code, it MUST pass the entry through unmodified.

```
<header required="...">
  <entry type="..." />
  ...
</header>
```

required Permitted values: true, false

Defines whether the header segment is always present or may be omitted.

Entry

Each entry in the definition of a header segment refers to a sized (size="4"), coded, packed struct that is permitted to appear within the segment. An entry is formally defined with the entry element:

```
<entry type="...">
  <doc> ... </doc>
  ...
</entry>
```

type References a valid struct by name:

- The struct MUST include a 32 bit size, size="4".
- The struct MUST be coded.

5.9.2. Body Segment

A body segment contains opaque data. It is formally defined with the body element:

```
<body required="..." />
```

Attributes of a body segment:

required Permitted values: true, false

Defines whether the body segment is always present or may be omitted from the assembly.

6. Constants

Name	Value	Description
`MIN-MAX-FRAME-SIZE`	`4096`	During the initial connection negotiation, the two peers must agree upon a maximum frame size. This constant defines the minimum value to which the maximum frame size can be set. By defining this value, the peers can guarantee that they can send frames of up to this size until they have agreed a definitive maximum frame size for that connection.

7. Types

Fixed width types

Name	Code	Width in Octets	Description
bin8	0x00	1	octet of unspecified encoding
int8	0x01	1	8-bit signed integral value (-128 - 127)
uint8	0x02	1	8-bit unsigned integral value (0 - 255)
char	0x04	1	an iso-8859-15 character
boolean	0x08	1	boolean value (zero represents false, nonzero represents true)
bin16	0x10	2	two octets of unspecified binary encoding
int16	0x11	2	16-bit signed integral value
uint16	0x12	2	16-bit unsigned integer
bin32	0x20	4	four octets of unspecified binary encoding
int32	0x21	4	32-bit signed integral value
uint32	0x22	4	32-bit unsigned integral value
float	0x23	4	single precision IEEE 754 32-bit floating point
char-utf32	0x27	4	single unicode character in UTF-32 encoding
sequence-no		4	serial number defined in RFC-1982
bin64	0x30	8	eight octets of unspecified binary encoding
int64	0x31	8	64-bit signed integral value
uint64	0x32	8	64-bit unsigned integral value
double	0x33	8	double precision IEEE 754 floating point
datetime	0x38	8	datetime in 64 bit POSIX time_t format
bin128	0x40	16	sixteen octets of unspecified binary encoding
uuid	0x48	16	UUID (RFC-4122 section 4.1.2) - 16 octets
bin256	0x50	32	thirty two octets of unspecified binary encoding
bin512	0x60	64	sixty four octets of unspecified binary encoding
bin1024	0x70	128	one hundred and twenty eight octets of unspecified binary encoding
bin40	0xc0	5	five octets of unspecified binary encoding
dec32	0xc8	5	32-bit decimal value (e.g. for use in financial values)
bin72	0xd0	9	nine octets of unspecified binary encoding
dec64	0xd8	9	64-bit decimal value (e.g. for use in financial values)
void	0xf0	0	the void type
bit	0xf1	0	presence indicator

Type: `bin8`

The bin8 type consists of exactly one octet of opaque binary data.

Wire Format

BNF:

```
bin8 = OCTET
```

Type: `int8`

The int8 type is a signed integral value encoded using an 8-bit two's complement representation.

Wire Format

BNF:

```
int8 = OCTET
```

Type: `uint8`

The uint8 type is an 8-bit unsigned integral value.

Wire Format

BNF:

```
uint8 = OCTET
```

Type: `char`

The char type encodes a single character from the iso-8859-15 character set.

Wire Format

BNF:

```
char = OCTET
```

Type: `boolean`

The boolean type is a single octet that encodes a true or false value. If the octet is zero, then the boolean is false. Any other value represents true.

Wire Format

BNF:

```
boolean = OCTET
```

Type: `bin16`

The bin16 type consists of two consecutive octets of opaque binary data.

Wire Format

BNF:

```
bin16 = 2 OCTET
```

Type: `int16`

The int16 type is a signed integral value encoded using a 16-bit two's complement representation in network byte order.

Wire Format

BNF:

```
           int16 = high-byte low-byte
       high-byte = OCTET
        low-byte = OCTET
```

Type: `uint16`

The uint16 type is a 16-bit unsigned integral value encoded in network byte order.

Wire Format

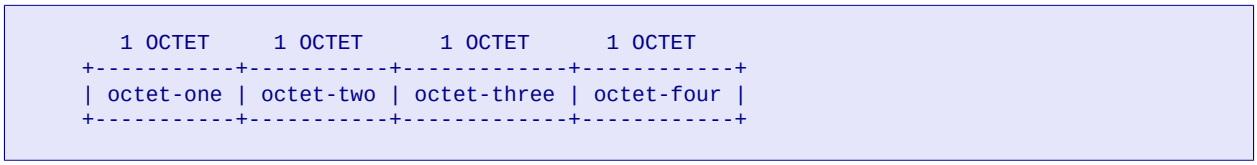
BNF:

```
          uint16 = high-byte low-byte
       high-byte = OCTET
        low-byte = OCTET
```

Type: `bin32`

The bin32 type consists of 4 consecutive octets of opaque binary data.

Wire Format



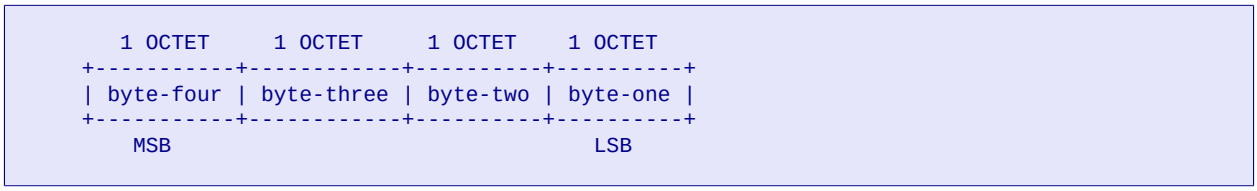
BNF:

```
bin32 = 4 OCTET
```

Type: `int32`

The int32 type is a signed integral value encoded using a 32-bit two's complement representation in network byte order.

Wire Format



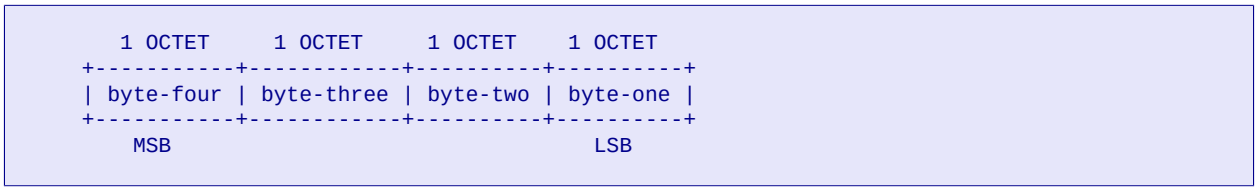
BNF:

```
           int32 = byte-four byte-three byte-two byte-one
      byte-four = OCTET ; most significant byte (MSB)
     byte-three = OCTET
       byte-two = OCTET
       byte-one = OCTET ; least significant byte (LSB)
```

Type: `uint32`

The uint32 type is a 32-bit unsigned integral value encoded in network byte order.

Wire Format



BNF:

```
          uint32 = byte-four byte-three byte-two byte-one
       byte-four = OCTET ; most significant byte (MSB)
      byte-three = OCTET
        byte-two = OCTET
        byte-one = OCTET ; least significant byte (LSB)
```

Type: `float`

The float type encodes a single precision 32-bit floating point number. The format and operations are defined by the IEEE 754 standard for 32-bit floating point numbers.

Wire Format

BNF:

```
float = 4 OCTET ; IEEE 754 32-bit floating point number
```

Type: `char-utf32`

The char-utf32 type consists of a single unicode character in the UTF-32 encoding.

Wire Format

BNF:

```
char-utf32 = 4 OCTET ; single UTF-32 character
```

Type: sequence-no

The sequence-no type encodes, in network byte order, a serial number as defined in RFC-1982. The arithmetic, operators, and ranges for numbers of this type are defined by RFC-1982.

Wire Format

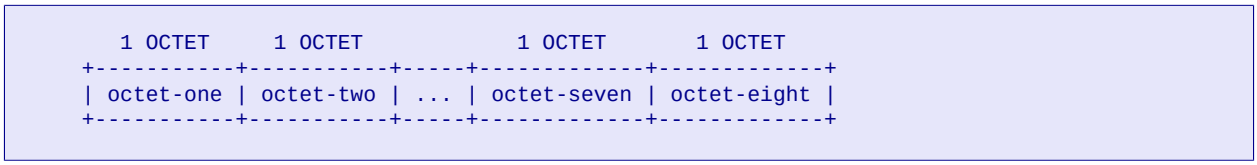
BNF:

```
sequence-no = 4 OCTET ; RFC-1982 serial number
```

Type: `bin64`

The bin64 type consists of eight consecutive octets of opaque binary data.

Wire Format



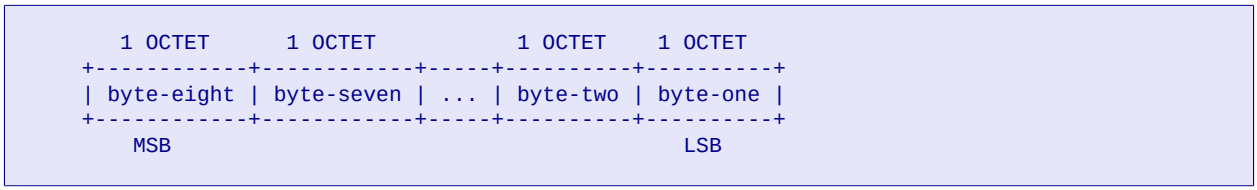
BNF:

```
bin64 = 8 OCTET
```

Type: `int64`

The int64 type is a signed integral value encoded using a 64-bit two's complement representation in network byte order.

Wire Format



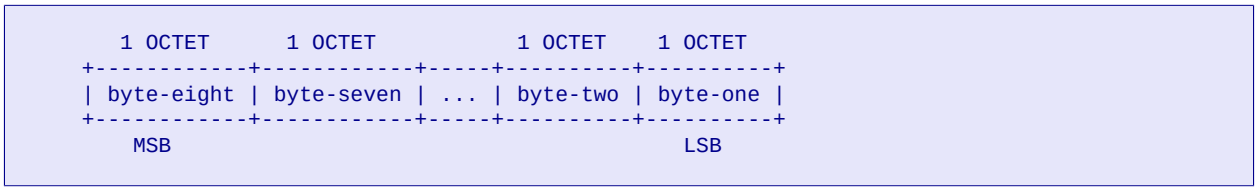
BNF:

```
           int64 = byte-eight byte-seven byte-six byte-five
                   byte-four byte-three byte-two byte-one
      byte-eight = 1 OCTET ; most significant byte (MSB)
      byte-seven = 1 OCTET
        byte-six = 1 OCTET
       byte-five = 1 OCTET
       byte-four = 1 OCTET
      byte-three = 1 OCTET
        byte-two = 1 OCTET
        byte-one = 1 OCTET ; least significant byte (LSB)
```

Type: `uint64`

The uint64 type is a 64-bit unsigned integral value encoded in network byte order.

Wire Format



BNF:

```
          uint64 = byte-eight byte-seven byte-six byte-five
                   byte-four byte-three byte-two byte-one
      byte-eight = 1 OCTET ; most significant byte (MSB)
      byte-seven = 1 OCTET
        byte-six = 1 OCTET
       byte-five = 1 OCTET
       byte-four = 1 OCTET
      byte-three = 1 OCTET
        byte-two = 1 OCTET
        byte-one = 1 OCTET ; least significant byte (LSB)
```

Type: `double`

The double type encodes a double precision 64-bit floating point number. The format and operations are defined by the IEEE 754 standard for 64-bit double precision floating point numbers.

Wire Format

BNF:

```
double = 8 OCTET ; double precision IEEE 754 floating point number
```

Type: `datetime`

The datetime type encodes a date and time using the 64 bit POSIX time_t format.

Wire Format

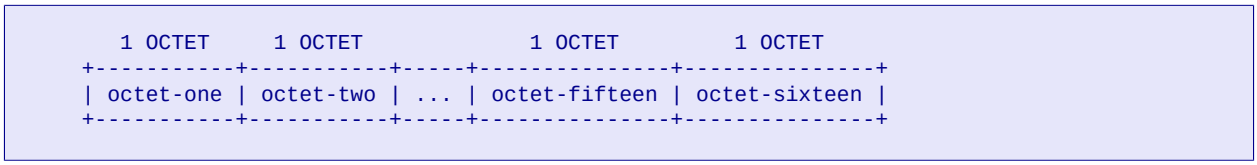
BNF:

```
datetime = 8 OCTET ; 64 bit posix time_t format
```

Type: `bin128`

The bin128 type consists of 16 consecutive octets of opaque binary data.

Wire Format



BNF:

```
bin128 = 16 OCTET
```

Type: `uuid`

The uuid type encodes a universally unique id as defined by RFC-4122. The format and operations for this type can be found in section 4.1.2 of RFC-4122.

Wire Format

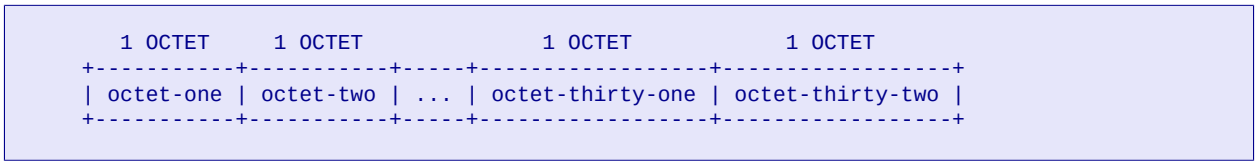
BNF:

```
uuid = 16 OCTET ; RFC-4122 section 4.1.2
```

Type: `bin256`

The bin256 type consists of thirty two consecutive octets of opaque binary data.

Wire Format



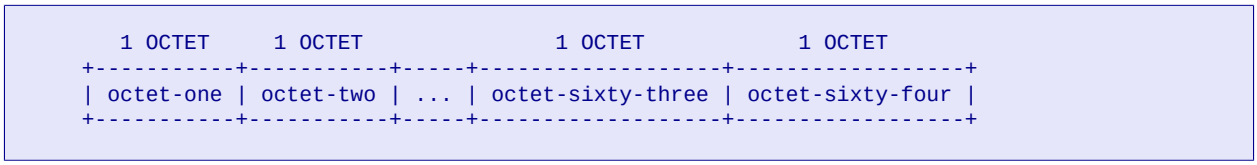
BNF:

```
bin256 = 32 OCTET
```

Type: `bin512`

The bin512 type consists of sixty four consecutive octets of opaque binary data.

Wire Format



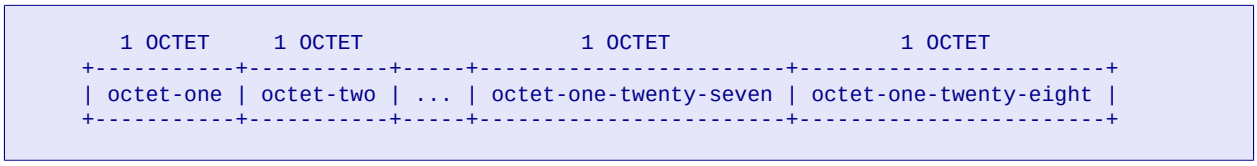
BNF:

```
bin512 = 64 OCTET
```

Type: `bin1024`

The bin1024 type consists of one hundred and twenty eight octets of opaque binary data.

Wire Format



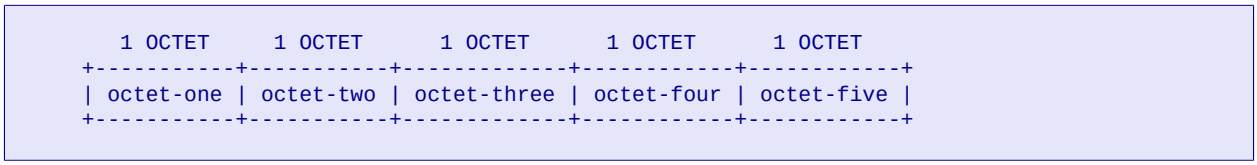
BNF:

```
bin1024 = 128 OCTET
```

Type: `bin40`

The bin40 type consists of five consecutive octets of opaque binary data.

Wire Format



BNF:

```
bin40 = 5 OCTET
```

Type: dec32

The dec32 type is decimal value with a variable number of digits following the decimal point. It is encoded as an 8-bit unsigned integral value representing the number of decimal places. This is followed by the signed integral value encoded using a 32-bit two's complement representation in network byte order.

The former value is referred to as the exponent of the divisor. The latter value is the mantissa. The decimal value is given by: mantissa / 10^exponent.

Wire Format

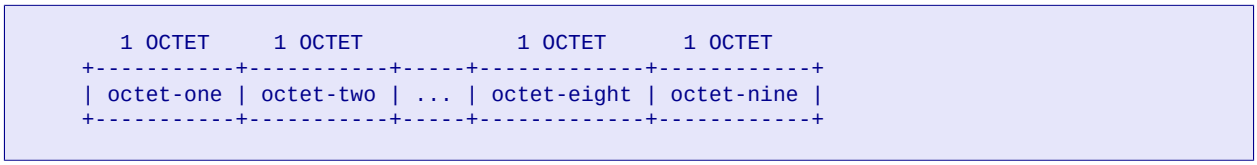
BNF:

```
           dec32 = exponent mantissa
        exponent = uint8
        mantissa = int32
```

Type: `bin72`

The bin72 type consists of nine consecutive octets of opaque binary data.

Wire Format



BNF:

```
bin64 = 9 OCTET
```

Type: dec64

The dec64 type is decimal value with a variable number of digits following the decimal point. It is encoded as an 8-bit unsigned integral value representing the number of decimal places. This is followed by the signed integral value encoded using a 64-bit two's complement representation in network byte order.

The former value is referred to as the exponent of the divisor. The latter value is the mantissa. The decimal value is given by: mantissa / 10^exponent.

Wire Format

BNF:

```
           dec64 = exponent mantissa
        exponent = uint8
        mantissa = int64
```

Type: `void`

The void type is used within tagged data structures such as maps and lists to indicate an empty value. The void type has no value and is encoded as an empty sequence of octets.

Type: `bit`

The bit type is used to indicate that a packing flag within a packed struct is being used to represent a boolean value based on the presence of an empty value. The bit type has no value and is encoded as an empty sequence of octets.

Variable width types

Variable width types consist of a number of octets which represent an unsgigned integral size; followed by the given number of octets. The size field should be read as if it were a uint8, if there is one size octet, as a uint16 if there are two size octets, a unit32 if there are four size octets, and so on.

Name	Code	Size Octets	Description
vbin8	0x80	1	up to 255 octets of opaque binary data
str8-latin	0x84	1	up to 255 iso-8859-15 characters
str8	0x85	1	up to 255 octets worth of UTF-8 unicode
str8-utf16	0x86	1	up to 255 octets worth of UTF-16 unicode
vbin16	0x90	2	up to 65535 octets of opaque binary data
str16-latin	0x94	2	up to 65535 iso-8859-15 characters
str16	0x95	2	up to 65535 octets worth of UTF-8 unicode
str16-utf16	0x96	2	up to 65535 octets worth of UTF-16 unicode
byte-ranges		2	byte ranges within a 64-bit payload
sequence-set		2	ranged set representation
vbin32	0xa0	4	up to 4294967295 octets of opaque binary data
map	0xa8	4	a mapping of keys to typed values
list	0xa9	4	a series of consecutive type-value pairs
array	0xaa	4	a defined length collection of values of a single type
struct32	0xab	4	a coded struct with a 32-bit size

Type: vbin8

The vbin8 type encodes up to 255 octets of opaque binary data. The number of octets is first encoded as an 8-bit unsigned integral value. This is followed by the actual data.

Wire Format

BNF:

```
vbin8 = size octets
 size = uint8
octets = 0*255 OCTET ; size OCTETs
```

Type: `str8-latin`

The str8-latin type encodes up to 255 octets of iso-8859-15 characters. The number of octets is first encoded as an 8-bit unsigned integral value. This is followed by the actual characters.

Wire Format

BNF:

```
str8-latin = size characters
      size = uint8
characters = 0*255 OCTET ; size OCTETs
```

Type: `str8`

The str8 type encodes up to 255 octets worth of UTF-8 unicode. The number of octets of unicode is first encoded as an 8-bit unsigned integral value. This is followed by the actual UTF-8 unicode. Note that the encoded size refers to the number of octets of unicode, not necessarily the number of characters since the UTF-8 unicode may include multi-byte character sequences.

Wire Format

BNF:

```
            str8 = size utf8-unicode
            size = uint8
     utf8-unicode = 0*255 OCTET ; size OCTETs
```

Type: `str8-utf16`

The str8-utf16 type encodes up to 255 octets worth of UTF-16 unicode. The number of octets of unicode is first encoded as an 8-bit unsigned integral value. This is followed by the actual UTF-16 unicode. Note that the encoded size refers to the number of octets of unicode, not the number of characters since the UTF-16 unicode will include at least two octets per unicode character.

Wire Format

BNF:

```
   str8-utf16 = size utf16-unicode
         size = uint8
utf16-unicode = 0*255 OCTET ; size OCTETs
```

Type: `vbin16`

The vbin16 type encodes up to 65535 octets of opaque binary data. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual data.

Wire Format

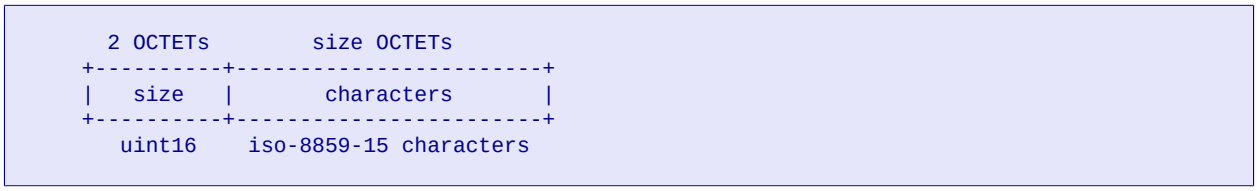
BNF:

```
vbin16 = size octets
  size = uint16
octets = 0*65535 OCTET ; size OCTETs
```

Type: `str16-latin`

The str16-latin type encodes up to 65535 octets of is-8859-15 characters. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual characters.

Wire Format



BNF:

```
str16-latin = size characters
       size = uint16
 characters = 0*65535 OCTET ; size OCTETs
```

Type: `str16`

The str16 type encodes up to 65535 octets worth of UTF-8 unicode. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual UTF-8 unicode. Note that the encoded size refers to the number of octets of unicode, not necessarily the number of unicode characters since the UTF-8 unicode may include multi-byte character sequences.

Wire Format

BNF:

```
           str16 = size utf8-unicode
            size = uint16
    utf8-unicode = 0*65535 OCTET ; size OCTETs
```

Type: `str16-utf16`

The str16-utf16 type encodes up to 65535 octets worth of UTF-16 unicode. The number of octets is first encoded as a 16-bit unsigned integral value in network byte order. This is followed by the actual UTF-16 unicode. Note that the encoded size refers to the number of octets of unicode, not the number of unicode characters since the UTF-16 unicode will include at least two octets per unicode character.

Wire Format

```
      2 OCTETs    size OCTETs
    +----------+---------------+
    |   size   | utf16-unicode |
    +----------+---------------+
       uint16
```

BNF:

```
      str16-utf16 = size utf16-unicode
             size = uint16
    utf16-unicode = 0*65535 OCTET ; size OCTETs
```

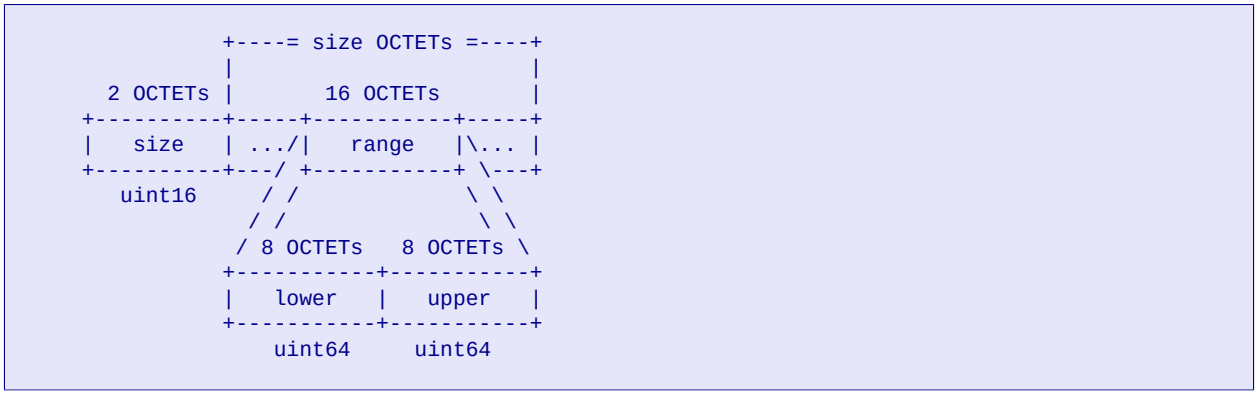
Type: `byte-ranges`

The byte-ranges type encodes up to 65535 octets worth of non-overlapping, non-touching, ascending byte ranges within a 64-bit sequence of bytes. Each range is represented as an inclusive lower and upper bound that identifies all the byte offsets included within a given range.

The number of octets of data is first encoded as a 16-bit unsigned integral value in network byte order. This is then followed by the encoded representation of the ranges included in the set. These MUST be encoded in ascending order, and any two ranges included in a given set MUST NOT include overlapping or touching byte offsets.

Each range is encoded as a pair of 64-bit unsigned integral values in network byte order respectively representing the lower and upper bounds for that range. Note that because each range is exactly 16 octets, the size in octets of the encoded ranges will always be 16 times the number of ranges in the set.

Wire Format



BNF:

```
byte-ranges = size *range
       size = uint16
      range = lower upper
      lower = uint64
      upper = uint64
```

Type: `sequence-set`

The sequence-set type is a set of pairs of RFC-1982 numbers representing a discontinuous range within an RFC-1982 sequence. Each pair represents a closed interval within the list.

Sequence-sets can be represented as lists of pairs of positive 32-bit numbers, each pair representing a closed interval that does not overlap or touch with any other interval in the list. For example, a set containing words 0, 1, 2, 5, 6, and 15 can be represented:

```
    [(0, 2), (5, 6), (15, 15)]
```

1) The list-of-pairs representation is sorted ascending (as defined by RFC 1982 (http://www.ietf.org/rfc/rfc1982.txt)) by the first elements of each pair.

2) The list-of-pairs is flattened into a list-of-words.

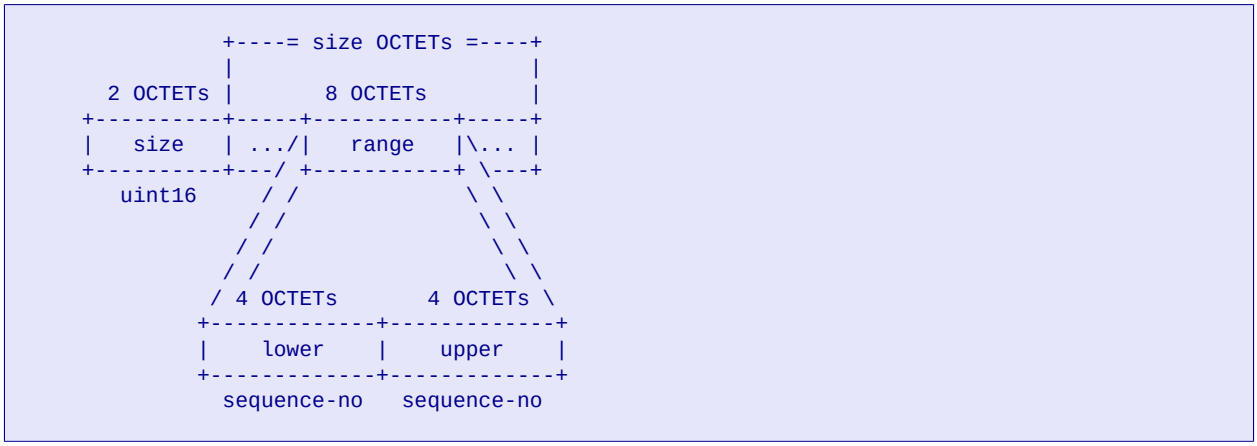
3) Each word in the list is packed into ascending locations in memory with network byte ordering.

4) The size in bytes, represented as a 16-bit network-byte-order unsigned value, is prepended.

For instance, the example from above would be encoded:

```
    [(0, 2), (5, 6), (15, 15)]                          -- already sorted.
    [0, 2, 5, 6, 15, 15]                                -- flattened.
    0000000000000002000000050000000600000000F0000000F     -- bytes in hex
    0018000000000000000200000005000000060000000F0000000F -- bytes in hex,
                                                             length (24) prepended
```

Wire Format



BNF:

```
 sequence-set = size *range
        size = uint16         ; length of variable portion in bytes

       range = lower upper    ; inclusive
       lower = sequence-no
       upper = sequence-no
```

Type: vbin32

The vbin32 type encodes up to 4294967295 octets of opaque binary data. The number of octets is first encoded as a 32-bit unsigned integral value in network byte order. This is followed by the actual data.

Wire Format

BNF:

```
vbin32 = size octets
  size = uint32
octets = 0*4294967295 OCTET ; size OCTETs
```

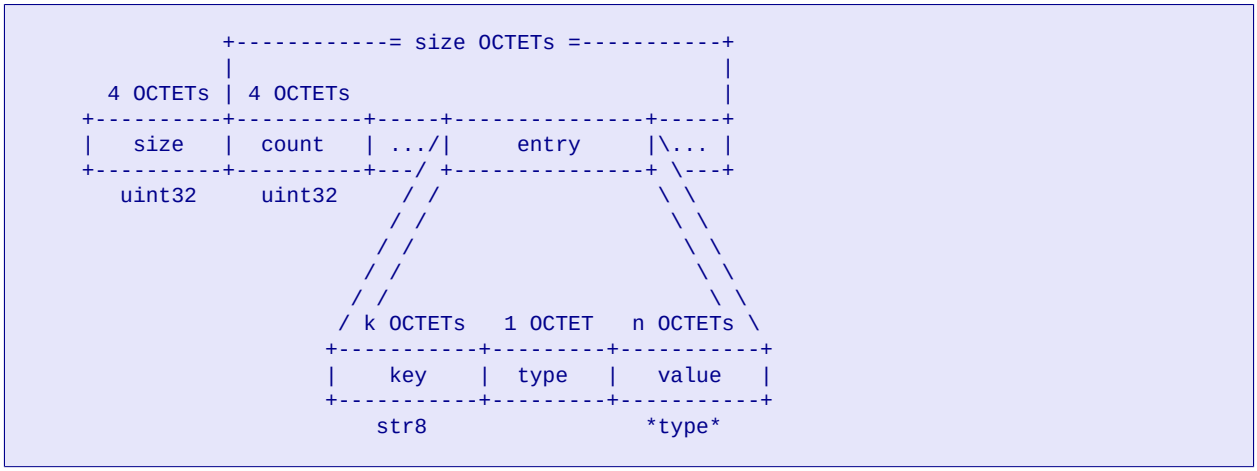
Type: `map`

A map is a set of distinct keys where each key has an associated (type,value) pair. The triple of the key, type, and value, form an entry within a map. Each entry within a given map MUST have a distinct key. A map is encoded as a size in octets, a count of the number of entries, followed by the encoded entries themselves.

An encoded map may contain up to (4294967295 - 4) octets worth of encoded entries. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded entries plus 4. (The extra 4 octets is added for the entry count.) The size is then followed by the number of entries encoded as a 32-bit unsigned integral value in network byte order. Finally the entries are encoded sequentially.

An entry is encoded as the key, followed by the type, and then the value. The key is always a string encoded as a str8. The type is a single octet that may contain any valid AMQP type code. The value is encoded according to the rules defined by the type code for that entry.

Wire Format



BNF:

```
  map = size count *entry

 size = uint32            ; size of count and entries in octets
count = uint32            ; number of entries in the map

entry = key type value
  key = str8
 type = OCTET             ; type code of the value
value = *OCTET            ; the encoded value
```

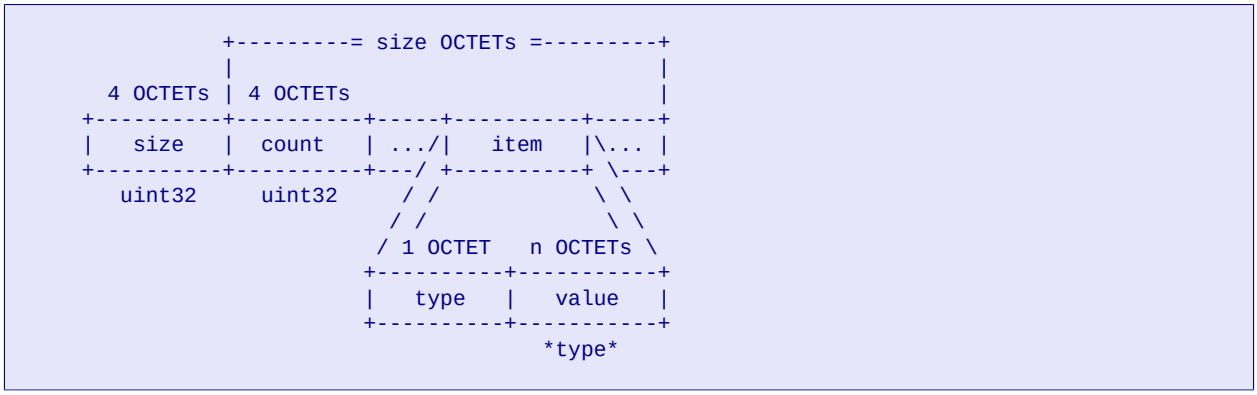
Type: `list`

A list is an ordered sequence of (type, value) pairs. The (type, value) pair forms an item within the list. The list may contain items of many distinct types. A list is encoded as a size in octets, followed by a count of the number of items, followed by the items themselves encoded in their defined order.

An encoded list may contain up to (4294967295 - 4) octets worth of encoded items. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded items plus 4. (The extra 4 octets is added for the item count.) The size is then followed by the number of items encoded as a 32-bit unsigned integral value in network byte order. Finally the items are encoded sequentially in their defined order.

An item is encoded as the type followed by the value. The type is a single octet that may contain any valid AMQP type code. The value is encoded according to the rules defined by the type code for that item.

Wire Format



BNF:

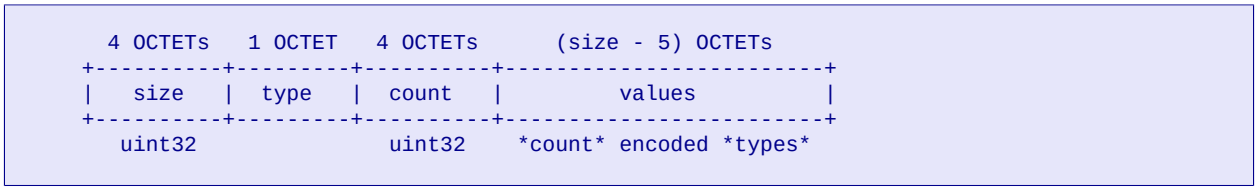
```
 list = size count *item

 size = uint32           ; size of count and items in octets
count = uint32           ; number of items in the list

 item = type value
 type = OCTET            ; type code of the value
value = *OCTET           ; the encoded value
```

Type: `array`

An array is an ordered sequence of values of the same type. The array is encoded in as a size in octets, followed by a type code, then a count of the number values in the array, and finally the values encoded in their defined order.

An encoded array may contain up to (4294967295 - 5) octets worth of encoded values. The size is encoded as a 32-bit unsigned integral value in network byte order equal to the number of octets worth of encoded values plus 5. (The extra 5 octets consist of 4 octets for the count of the number of values, and one octet to hold the type code for the items in the array.) The size is then followed by a single octet that may contain any valid AMQP type code. The type code is then followed by the number of values encoded as a 32-bit unsigned integral value in network byte order. Finally the values are encoded sequentially in their defined order according to the rules defined by the type code for the array.

Wire Format



BNF:

```
array = size type count values

 size = uint32           ; size of type, count, and values in octets
 type = OCTET            ; the type of the encoded values
count = uint32           ; number of items in the array

values = 0*4294967290 OCTET ; (size - 5) OCTETs
```

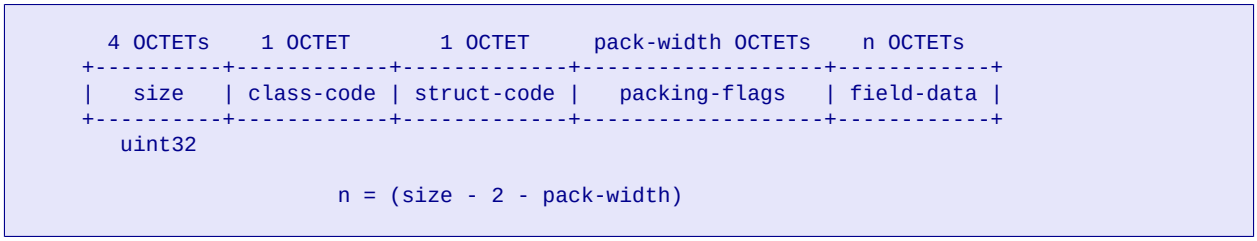
Type: `struct32`

The struct32 type describes any coded struct with a 32-bit (4 octet) size. The type is restricted to be only coded structs with a 32-bit size, consequently the first six octets of any encoded value for this type MUST always contain the size, class-code, and struct-code in that order.

The size is encoded as a 32-bit unsigned integral value in network byte order that is equal to the size of the encoded field-data, packing-flags, class-code, and struct-code. The class-code is a single octet that may be set to any valid class code. The struct-code is a single octet that may be set to any valid struct code within the given class-code.

The first six octets are then followed by the packing flags and encoded field data. The presence and quantity of packing-flags, as well as the specific fields are determined by the struct definition identified with the encoded class-code and struct-code.

Wire Format



BNF:

```
         struct32 = size class-code struct-code packing-flags field-data

             size = uint32

       class-code = OCTET     ; zero for top-level structs
      struct-code = OCTET     ; together with class-code identifies the struct
                              ; definition which determines the pack-width and
                              ; fields

    packing-flags = 0*4 OCTET ; pack-width OCTETs

       field-data = *OCTET    ; (size - 2 - pack-width) OCTETs
```

Mandatory Types

The following types MUST be natively understood by a conforming AMQP Server:

uint8, uint16, uint32, sequence-no, uint64, datetime, uuid, vbin8, str8, vbin16, str16, byte-ranges, sequence-set, vbin32, map, array, struct32, bit.

Other types are defined for the use of application defined properties which may be passed in the header sections of messages. Since such values are passed through unchanged by an AMQP server, there is no need for the server to parse them.

8. Domains

Domain: `segment-type`

Name	Type	Description
`segment-type`	`uint8`	valid values for the frame type indicator.

Segments are defined in Section 4.4.1, “Assemblies, Segments, and Frames”. The segment domain defines the valid values that may be used for the segment indicator within the frame header.

Valid Values

Value	Name	Description
`0`	`control`	The frame type indicator for Control segments (see Section 5.7, “Controls”).
`1`	`command`	The frame type indicator for Command segments (see Section 5.8, “Commands”).
`2`	`header`	The frame type indicator for Header segments (see Section 5.9.1, “Header Segment”).
`3`	`body`	The frame type indicator for Body segments (see Section 5.9.2, “Body Segment”).

Domain: `track`

Name	Type	Description
`track`	`uint8`	Valid values for transport level tracks

Tracks are defined in Section 4.4.2, “Channels and Tracks”. The track domain defines the valid values that may used for the track indicator within the frame header

Valid Values

Value	Name	Description
`0`	`control`	The track used for all controls. All controls defined in this specification MUST be sent on track 0.
`1`	`command`	The track used for all commands. All commands defined in this specification MUST be sent on track 1.

Domain: `str16-array`

Name	Type	Description
`str16-array`	`array`	An array of values of type str16.

An array of values of type str16.

9. Control Classes

Class: connection

Code	Name	Description
0x1	connection	work with connections

An AMQP server MUST implement the connection class.

An AMQP client MUST implement the connection class.

Methods

Code	Name
0x1	start(server-properties: *map*, mechanisms: *str16-array*, locales: *str16-array*)
	start connection negotiation
0x2	start-ok(client-properties: *map*, mechanism: *str8*, response: *vbin32*, locale: *str8*)
	select security mechanism and locale
0x3	secure(challenge: *vbin32*)
	security mechanism challenge
0x4	secure-ok(response: *vbin32*)
	security mechanism response
0x5	tune(channel-max: *uint16*, max-frame-size: *uint16*, heartbeat-min: *uint16*, heartbeat-max: *uint16*)
	propose connection tuning parameters
0x6	tune-ok(channel-max: *uint16*, max-frame-size: *uint16*, heartbeat: *uint16*)
	negotiate connection tuning parameters
0x7	open(virtual-host: *str8*, capabilities: *str16-array*, insist: *bit*)
	open connection to virtual host
0x8	open-ok(known-hosts: *amqp-host-array*)
	signal that connection is ready
0x9	redirect(host: *amqp-host-url*, known-hosts: *amqp-host-array*)
	redirects client to other server
0xa	heartbeat()
	indicates connection is still alive
0xb	close(reply-code: *close-code*, reply-text: *str8*)
	request a connection close
0xc	close-ok()
	confirm a connection close

The connection class provides controls for a client to establish a network connection to a server, and for both peers to operate the connection thereafter.

Grammar:

```
connection        = open-connection
                    *use-connection
                    close-connection
open-connection   = C:protocol-header
                    S:START C:START-OK
                    *challenge
                    S:TUNE C:TUNE-OK
                    C:OPEN S:OPEN-OK | S:REDIRECT
challenge         = S:SECURE C:SECURE-OK
use-connection    = *channel
close-connection  = C:CLOSE S:CLOSE-OK
                  / S:CLOSE C:CLOSE-OK
```

Domain: `connection.close-code`

Name	Type	Description
`close-code`	`uint16`	code used in the connection.close control to indicate reason for closure

Valid Values

Value	Name	Description
`200`	`normal`	The connection closed normally.
`320`	`connection-forced`	An operator intervened to close the connection for some reason. The client may retry at some later date.
`402`	`invalid-path`	The client tried to work with an unknown virtual host.
`501`	`framing-error`	A valid frame header cannot be formed from the incoming byte stream.

Domain: `connection.amqp-host-url`

Name	Type	Description
`amqp-host-url`	`str16`	URL for identifying an AMQP Server

The amqp-url domain defines a format for identifying an AMQP Server. It is used to provide alternate hosts in the case where a client has to reconnect because of failure, or because the server requests the client to do so upon initial connection.

BNF:

```
amqp_url          = "amqp:" prot_addr_list
prot_addr_list    = [prot_addr ","]* prot_addr
prot_addr         = tcp_prot_addr | tls_prot_addr

tcp_prot_addr     = tcp_id tcp_addr
tcp_id            = "tcp:" | ""
tcp_addr          = [host [":" port] ]
host              = <as per http://www.ietf.org/rfc/rfc3986.txt>
port              = number
```

Domain: `connection.amqp-host-array`

Name	Type	Description
`amqp-host-array`	`array`	An array of values of type amqp-host-url

Used to provide a list of alternate hosts.

Control: `connection.start`

Name	`start`
Code	`0x1`
Response	start-ok

An AMQP client MUST handle incoming connection.start controls.

This control starts the connection negotiation process by telling the client the supported security mechanisms and locales from which the client can choose.

Arguments

Name	Type	Description	
`server-properties`	`map`	server properties	**optional**
`mechanisms`	`str16-array`	available security mechanisms	**required**
	A list of the security mechanisms that the server supports.		
`locales`	`str16-array`	available message locales	**required**
	A list of the message locales that the server supports. The locale defines the language in which the server will send reply texts.		

Rules

Rule: protocol-name

If the server cannot support the protocol specified in the protocol header, it MUST close the socket connection without sending any response control.

Scenario: The client sends a protocol header containing an invalid protocol name. The server must respond by closing the connection.

Rule: client-support

If the client cannot handle the protocol version suggested by the server it MUST close the socket connection.

Scenario: The server sends a protocol version that is lower than any valid implementation, e.g. 0.1. The client must respond by closing the connection.

Rule: required-fields

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.

Scenario: Client connects to server and inspects the server properties. It checks for the presence of the required fields.

Rule: required-support

The server MUST support at least the en_US locale.

Scenario: Client connects to server and inspects the locales field. It checks for the presence of the required locale(s).

Control: `connection.start-ok`

Name	`start-ok`
Code	`0x2`

An AMQP server MUST handle incoming connection.start-ok controls.

This control selects a SASL security mechanism.

Arguments

Name	Type	Description	
`client-properties`	`map`	client properties	**optional**
`mechanism`	`str8`	selected security mechanism	**required**
	A single security mechanisms selected by the client, which must be one of those specified by the server.		
`response`	`vbin32`	security response data	**required**
	A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.		
`locale`	`str8`	selected message locale	**required**
	A single message locale selected by the client, which must be one of those specified by the server.		

Rules

Rule: required-fields

The properties SHOULD contain at least these fields: "product", giving the name of the client product, "version", giving the name of the client version, "platform", giving the name of the operating system, "copyright", if appropriate, and "information", giving other general information.

Rule: security

The client SHOULD authenticate using the highest-level security profile it can handle from the list provided by the server.

Rule: validity

If the mechanism field does not contain one of the security mechanisms proposed by the server in the Start control, the server MUST close the connection without sending any further data.

Scenario: Client connects to server and sends an invalid security mechanism. The server must respond by closing the connection (a socket close, with no connection close negotiation).

Control: `connection.secure`

Name	`secure`
Code	`0x3`
Response	secure-ok

An AMQP client MUST handle incoming connection.secure controls.

The SASL protocol works by exchanging challenges and responses until both peers have received sufficient information to authenticate each other. This control challenges the client to provide more information.

Arguments

Name	Type	Description	
`challenge`	`vbin32`	security challenge data	**required**
	Challenge information, a block of opaque binary data passed to the security mechanism.		

Control: `connection.secure-ok`

Name	`secure-ok`
Code	`0x4`

An AMQP server MUST handle incoming connection.secure-ok controls.

This control attempts to authenticate, passing a block of SASL data for the security mechanism at the server side.

Arguments

Name	Type	Description	
`response`	`vbin32`	security response data	**required**
	A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.		

Control: `connection.tune`

Name	`tune`
Code	`0x5`
Response	tune-ok

An AMQP client MUST handle incoming connection.tune controls.

This control proposes a set of connection configuration values to the client. The client can accept and/or adjust these.

Arguments

Name	Type	Description	
`channel-max`	`uint16`	proposed maximum channels	**optional**
	The maximum total number of channels that the server allows per connection. If this is not set it means that the server does not impose a fixed limit, but the number of allowed channels may be limited by available server resources.		
`max-frame-size`	`uint16`	proposed maximum frame size	**optional**
	The largest frame size that the server proposes for the connection. The client can negotiate a lower value. If this is not set means that the server does not impose any specific limit but may reject very large frames if it cannot allocate resources for them.		
`heartbeat-min`	`uint16`	the minimum supported heartbeat delay	**optional**
	The minimum delay, in seconds, of the connection heartbeat supported by the server. If this is not set it means the server does not support sending heartbeats.		
`heartbeat-max`	`uint16`	the maximum supported heartbeat delay	**optional**
	The maximum delay, in seconds, of the connection heartbeat supported by the server. If this is not set it means the server has no maximum.		

Rules

Rule: minimum

Until the max-frame-size has been negotiated, both peers MUST accept frames of up to MIN-MAX-FRAME-SIZE octets large, and the minimum negotiated value for max-frame-size is also MIN-MAX-FRAME-SIZE.

Scenario: Client connects to server and sends a large properties field, creating a frame of MIN-MAX-FRAME-SIZE octets. The server must accept this frame.

Rule: permitted-range

The heartbeat-max value must be greater than or equal to the value supplied in the heartbeat-min field.

Rule: no-heartbeat-min

If no heartbeat-min is supplied, then the heartbeat-max field MUST remain empty.

Control: `connection.tune-ok`

Name	`tune-ok`
Code	`0x6`

An AMQP server MUST handle incoming connection.tune-ok controls.

This control sends the client's connection tuning parameters to the server. Certain fields are negotiated, others provide capability information.

Arguments

Name	Type	Description	
`channel-max`	`uint16`	negotiated maximum channels	**required**
	The maximum total number of channels that the client will use per connection.		
`max-frame-size`	`uint16`	negotiated maximum frame size	**optional**
	The largest frame size that the client and server will use for the connection. If it is not set means that the client does not impose any specific limit but may reject very large frames if it cannot allocate resources for them. Note that the max-frame-size limit applies principally to content frames, where large contents can be broken into frames of arbitrary size.		
`heartbeat`	`uint16`	negotiated heartbeat delay	**optional**
	The delay, in seconds, of the connection heartbeat chosen by the client. If it is not set it means the client does not want a heartbeat.		

Rules

Rule: upper-limit

If the client specifies a channel max that is higher than the value provided by the server, the server MUST close the connection without attempting a negotiated close. The server may report the error in some fashion to assist implementers.

Rule: available-channels

If the client agrees to a channel-max of N channels, then the channels available for communication between client and server are precisely the channels numbered 0 to (N-1).

Rule: minimum

Until the max-frame-size has been negotiated, both peers MUST accept frames of up to MIN-MAX-FRAME-SIZE octets large, and the minimum negotiated value for max-frame-size is also MIN-MAX-FRAME-SIZE.

Rule: upper-limit

If the client specifies a max-frame-size that is higher than the value provided by the server, the server MUST close the connection without attempting a negotiated close. The server may report the error in some fashion to assist implementers.

Rule: max-frame-size

A peer MUST NOT send frames larger than the agreed-upon size. A peer that receives an oversized frame MUST close the connection with the framing-error close-code.

Rule: permitted-range

The chosen heartbeat MUST be in the range supplied by the heartbeat-min and heartbeat-max fields of connection.tune.

Rule: no-heartbeat-min

The heartbeat field MUST NOT be set if the heartbeat-min field of connection.tune was not set by the server.

Control: connection.open

Name	open
Code	0x7
Response	open-ok
Response	redirect

An AMQP server MUST handle incoming connection.open controls.

This control opens a connection to a virtual host, which is a collection of resources, and acts to separate multiple application domains within a server. The server may apply arbitrary limits per virtual host, such as the number of each type of entity that may be used, per connection and/or in total.

Arguments

Name	Type	Description	
virtual-host	str8	virtual host name	**required**
	The name of the virtual host to work with.		
capabilities	str16-array	required capabilities	**optional**
	The client can specify zero or more capability names. The server can use this to determine how to process the client's connection request.		
insist	bit	insist on connecting to server	**optional**
	In a configuration with multiple collaborating servers, the server may respond to a connection.open control with a Connection.Redirect. The insist option tells the server that the client is insisting on a connection to the specified server.		

Rules

Rule: separation

If the server supports multiple virtual hosts, it MUST enforce a full separation of exchanges, queues, and all associated entities per virtual host. An application, connected to a specific virtual host, MUST NOT be able to access resources of another virtual host.

Rule: security

The server SHOULD verify that the client has permission to access the specified virtual host.

Rule: behavior

When the client uses the insist option, the server MUST NOT respond with a Connection.Redirect control. If it cannot accept the client's connection request it should respond by closing the connection with a suitable reply code.

Control: `connection.open-ok`

Name	`open-ok`
Code	`0x8`

An AMQP client MUST handle incoming connection.open-ok controls.

This control signals to the client that the connection is ready for use.

Arguments

Name	Type	Description	
`known-hosts`	`amqp-host-array`	alternate hosts which may be used in the case of failure	**optional**
	Specifies an array of equivalent or alternative hosts that the server knows about, which will normally include the current server itself. Each entry in the array will be in the form of an IP address or DNS name, optionally followed by a colon and a port number. Clients can cache this information and use it when reconnecting to a server after a failure. This field may be empty.		

Control: `connection.redirect`

Name	`redirect`
Code	`0x9`

An AMQP client MUST handle incoming connection.redirect controls.

This control redirects the client to another server, based on the requested virtual host and/or capabilities.

Arguments

Name	Type	Description	
`host`	`amqp-host-url`	server to connect to	**required**
	Specifies the server to connect to.		
`known-hosts`	`amqp-host-array`	alternate hosts to try in case of failure	**optional**
	An array of equivalent or alternative hosts that the server knows about.		

Rules

Rule: usage

When getting the connection.redirect control, the client SHOULD reconnect to the host specified, and if that host is not present, to any of the hosts specified in the known-hosts list.

Control: `connection.heartbeat`

Name	`heartbeat`
Code	`0xa`

The heartbeat control may be used to generate artificial network traffic when a connection is idle. If a connection is idle for more than twice the negotiated heartbeat delay, the peers MAY be considered disconnected.

Control: `connection.close`

Name	`close`
Code	`0xb`
Response	close-ok

An AMQP client MUST handle incoming connection.close controls.

An AMQP server MUST handle incoming connection.close controls.

This control indicates that the sender wants to close the connection. The reason for close is indicated with the reply-code and reply-text. The channel this control is sent on MAY be used to indicate which channel caused the connection to close.

Arguments

Name	Type	Description	
`reply-code`	`close-code`	the numeric reply code	**required**
	Indicates the reason for connection closure.		
`reply-text`	`str8`	the localized reply text	**optional**
	This text can be logged as an aid to resolving issues.		

Control: `connection.close-ok`

Name	`close-ok`
Code	`0xc`

An AMQP client MUST handle incoming connection.close-ok controls.

An AMQP server MUST handle incoming connection.close-ok controls.

This control confirms a connection.close control and tells the recipient that it is safe to release resources for the connection and close the socket.

Rules

Rule: reporting

A peer that detects a socket closure without having received a Close-Ok handshake control SHOULD log the error.

Class: session

Code	Name	Description
0x2	session	session controls

An AMQP server MUST implement the session class.

An AMQP client MUST implement the session class.

An AMQP sender MUST implement the session class.

An AMQP receiver MUST implement the session class.

Methods

Code	Name
0x1	attach(name: *name*, force: *bit*)
	attach to the named session
0x2	attached(name: *name*)
	confirm attachment to the named session
0x3	detach(name: *name*)
	detach from the named session
0x4	detached(name: *name*, code: *detach-code*)
	confirm detachment from the named session
0x5	request-timeout(timeout: *uint32*)
	requests the execution timeout be changed
0x6	timeout(timeout: *uint32*)
	the granted timeout
0x7	command-point(command-id: *sequence-no*, command-offset: *uint64*)
	the command id and byte offset of subsequent data
0x8	expected(commands: *commands*, fragments: *command-fragments*)
	informs the peer of expected commands
0x9	confirmed(commands: *commands*, fragments: *command-fragments*)
	notifies of confirmed commands
0xa	completed(commands: *commands*, timely-reply: *bit*)
	notifies of command completion
0xb	known-completed(commands: *commands*)
	Inform peer of which commands are known to be completed
0xc	flush(expected: *bit*, confirmed: *bit*, completed: *bit*)
	requests a session.completed
0xd	gap(commands: *commands*)
	indicates missing segments in the stream

A session is a named interaction between two peers. Session names are chosen by the upper layers and may be used indefinitely. The model layer may associate long-lived or durable state with a given session name. The session layer provides transport of commands associated with this interaction.

The controls defined within this class are specified in terms of the "sender" of commands and the "receiver" of commands. Since both client and server send and receive commands, the overall session dialog is symmetric, however the semantics of the session controls are defined in terms of a single sender/receiver pair, and it is assumed that the client and server will each contain both a sender and receiver implementation.

Rules

Rule: attachment
The transport MUST be attached in order to use any control other than "attach", "attached", "detach", or "detached". A peer receiving any other control on a detached transport MUST discard it and send a session.detached with the "not-attached" reason code.

Domain: `session.header`

The session header appears on commands after the class and command id, but prior to command arguments.

Struct Type

Size	Packing
1	1

Fields

Name	Type	Description	
`sync`	`bit`	request notification of completion	**optional**
	Request notification of completion for this command.		

Domain: `session.command-fragment`

Struct Type

Size	Packing
0	0

Fields

Name	Type	Description	
`command-id`	`sequence-no`		**required**
`byte-ranges`	`byte-ranges`		**required**

Domain: `session.name`

Name	Type	Description
`name`	`vbin16`	opaque session name

The session name uniquely identifies an interaction between two peers. It is scoped to a given authentication principal.

Domain: `session.detach-code`

Name	Type	Description
`detach-code`	`uint8`	reason for detach

Valid Values

Value	Name	Description
`0`	`normal`	The session was detached by request.
`1`	`session-busy`	The session is currently attached to another transport.
`2`	`transport-busy`	The transport is currently attached to another session.
`3`	`not-attached`	The transport is not currently attached to any session.
`4`	`unknown-ids`	Command data was received prior to any use of the command-point control.

Domain: `session.commands`

Name	Type	Description
`commands`	`sequence-set`	identifies a set of commands

Domain: `session.command-fragments`

Name	Type	Description
`command-fragments`	`array`	an array of values of type command-fragment

Control: session.attach

Name	attach
Code	0x1
Response	attached
Response	detached

An AMQP server MUST handle incoming session.attach controls.

An AMQP client MAY handle incoming session.attach controls.

Requests that the current transport be attached to the named session. Success or failure will be indicated with an attached or detached response. This control is idempotent.

Arguments

Name	Type	Description	
name	name	the session name	**required**
	Identifies the session to be attached to the current transport.		
force	bit	force attachment to a busy session	**optional**
	If set then a busy session will be forcibly detached from its other transport and reattached to the current transport.		

Rules

Rule: one-transport-per-session

A session MUST NOT be attached to more than one transport at a time.

Rule: one-session-per-transport

A transport MUST NOT be attached to more than one session at a time.

Rule: idempotence

Attaching a session to its current transport MUST succeed and result in an attached response.

Rule: scoping

Attachment to the same session name from distinct authentication principals MUST succeed.

Control: `session.attached`

Name	`attached`
Code	`0x2`

An AMQP server MUST handle incoming session.attached controls.

An AMQP client MUST handle incoming session.attached controls.

Confirms successful attachment of the transport to the named session.

Arguments

Name	Type	Description	
`name`	`name`	the session name	**required**
	Identifies the session now attached to the current transport.		

Control: `session.detach`

Name	`detach`
Code	`0x3`
Response	detached

An AMQP server MUST handle incoming session.detach controls.

An AMQP client MUST handle incoming session.detach controls.

Detaches the current transport from the named session.

Arguments

Name	Type	Description	
`name`	`name`	the session name	**required**
	Identifies the session to detach.		

Control: `session.detached`

Name	`detached`
Code	`0x4`

An AMQP server MUST handle incoming session.detached controls.

An AMQP client MUST handle incoming session.detached controls.

Confirms detachment of the current transport from the named session.

Arguments

Name	Type	Description	
`name`	`name`	the session name	**required**
	Identifies the detached session.		
`code`	`detach-code`	the reason for detach	**required**
	Identifies the reason for detaching from the named session.		

Control: `session.request-timeout`

Name	`request-timeout`
Code	`0x5`
Response	timeout

An AMQP sender MUST handle incoming session.request-timeout controls.

An AMQP receiver MUST handle incoming session.request-timeout controls.

This control may be sent by either the sender or receiver of commands. It requests that the execution timeout be changed. This is the minimum amount of time that a peer must preserve execution state for a detached session.

Arguments

Name	Type	Description	
`timeout`	`uint32`	the requested timeout	**optional**
	The requested timeout for execution state in seconds. If not set, this control requests that execution state is preserved indefinitely.		

Rules

Rule: maximum-granted-timeout

The handler of this request MUST set his timeout to the maximum allowed value less than or equal to the requested timeout, and MUST convey the chosen timeout in the response.

Control: `session.timeout`

Name	`timeout`
Code	`0x6`

An AMQP sender MUST handle incoming session.timeout controls.

An AMQP receiver MUST handle incoming session.timeout controls.

This control may be sent by the either the sender or receiver of commands. It is a one-to-one reply to the request-timeout control that indicates the granted timeout for execution state.

Arguments

Name	Type	Description	
`timeout`	`uint32`	the execution timeout	**optional**
	The timeout for execution state. If not set, then execution state is preserved indefinitely.		

Control: `session.command-point`

Name	`command-point`
Code	`0x7`

An AMQP receiver MUST handle incoming session.command-point controls.

This control is sent by the sender of commands and handled by the receiver of commands. This establishes the sequence numbers associated with all subsequent command data sent from the sender to the receiver. The subsequent command data will be numbered starting with the values supplied in this control and proceeding sequentially. This must be used at least once prior to sending any command data on newly attached transports.

Arguments

Name	Type	Description	
`command-id`	`sequence-no`	the command-id of the next command	**required**
`command-offset`	`uint64`	the byte offset within the command	**required**

Rules

Rule: newly-attached-transports

If command data is sent on a newly attached transport the session MUST be detached with an "unknown-id" reason-code.

Rule: zero-offset

If the offset is zero, the next data frame MUST have the first-frame and first-segment flags set. Violation of this is a framing error.

Rule: nonzero-offset

If the offset is nonzero, the next data frame MUST NOT have both the first-frame and first-segment flag set. Violation of this is a framing error.

Control: `session.expected`

Name	`expected`
Code	`0x8`

An AMQP sender MUST handle incoming session.expected controls.

This control is sent by the receiver of commands and handled by the sender of commands. It informs the sender of what commands and command fragments are expected at the receiver. This control is only sent in response to a flush control with the expected flag set. The expected control is never sent spontaneously.

Arguments

Name	Type	Description	
`commands`	`commands`	expected commands	**required**
`fragments`	`command-fragments`	expected fragments	**optional**

Rules

Rule: include-next-command

The set of expected commands MUST include the next command after the highest seen command.

Rule: commands-empty-means-new-session

The set of expected commands MUST have zero elements if and only if the sender holds no execution state for the session (i.e. it is a new session).

Rule: no-overlaps

If a command-id appears in the commands field, it MUST NOT appear in the fragments field.

Rule: minimal-fragments

When choice is permitted, a command MUST appear in the commands field rather than the fragments field.

Control: session.confirmed

Name	confirmed
Code	0x9

An AMQP sender MUST handle incoming session.confirmed controls.

This control is sent by the receiver of commands and handled by the sender of commands. This sends the set of commands that will definitely be completed by this peer to the sender. This excludes commands known by the receiver to be considered confirmed or complete at the sender.

This control must be sent if the partner requests the set of confirmed commands using the session.flush control with the confirmed flag set.

This control may be sent spontaneously. One reason for separating confirmation from completion is for large persistent messages, where the receipt (and storage to a durable store) of part of the message will result in less data needing to be replayed in the case of transport failure during transmission.

A simple implementation of an AMQP client or server may be implemented to take no action on receipt of session.confirmed controls, and take action only when receiving session.completed controls.

A simple implementation of an AMQP client or server may be implemented such that it never spontaneously sends session.confirmed and that when requested for the set of confirmed commands (via the session.flush control) it responds with the same set of commands as it would to when the set of completed commands was requested (trivially all completed commands are confirmed).

Arguments

Name	Type	Description	
commands	commands	entirely confirmed commands	**optional**
fragments	command-fragments	partially confirmed commands	**optional**

Rules

Rule: durability

If a command has durable implications, it MUST NOT be confirmed until the fact of the command has been recorded on durable media.

Rule: no-overlaps

If a command-id appears in the commands field, it MUST NOT appear in the fragments field.

Rule: minimal-fragments

When choice is permitted, a command MUST appear in the commands field rather than the fragments field.

Rule: exclude-known-complete

Command-ids included in prior known-complete replies MUST be excluded from the set of all confirmed commands.

Control: `session.completed`

Name	`completed`
Code	`0xa`

An AMQP sender MUST handle incoming session.completed controls.

This control is sent by the receiver of commands, and handled by the sender of commands. It informs the sender of all commands completed by the receiver. This excludes commands known by the receiver to be considered complete at the sender.

Arguments

Name	Type	Description	
`commands`	`commands`	completed commands	**optional**
	The ids of all completed commands. This excludes commands known by the receiver to be considered complete at the sender.		
`timely-reply`	`bit`		**optional**
	If set, the sender is no longer free to delay the known-completed reply.		

Rules

Rule: known-completed-reply

The sender MUST eventually reply with a known-completed set that covers the completed ids.

Rule: delayed-reply

The known-complete reply MAY be delayed at the senders discretion if the timely-reply field is not set.

Rule: merged-reply

Multiple replies may be merged by sending a single known-completed that includes the union of the merged command-id sets.

Rule: completed-implies-confirmed

The sender MUST consider any completed commands to also be confirmed.

Rule: exclude-known-complete

Command-ids included in prior known-complete replies MUST be excluded from the set of all completed commands.

Control: `session.known-completed`

Name	`known-completed`
Code	`0xb`

An AMQP receiver MUST handle incoming session.known-completed controls.

This control is sent by the sender of commands, and handled by the receiver of commands. It is sent in reply to one or more completed controls from the receiver. It informs the receiver that commands are known to be completed by the sender.

Arguments

Name	Type	Description	
`commands`	`commands`	commands known to be complete	**optional**
	The set of completed commands for one or more session.completed controls.		

Rules

Rule: stateless

The sender need not keep state to generate this reply. It is sufficient to reply to any completed control with an exact echo of the completed ids.

Rule: known-completed-implies-known-confirmed

The receiver MUST treat any of the specified commands to be considered by the sender as confirmed as well as completed.

Control: `session.flush`

Name	`flush`
Code	`0xc`

An AMQP receiver MUST handle incoming session.flush controls.

This control is sent by the sender of commands and handled by the receiver of commands. It requests that the receiver produce the indicated command sets. The receiver should issue the indicated sets at the earliest possible opportunity.

Arguments

Name	Type	Description	
`expected`	`bit`	request notification of expected commands	**optional**
`confirmed`	`bit`	request notification of confirmed commands	**optional**
`completed`	`bit`	request notification of completed commands	**optional**

Control: `session.gap`

Name	`gap`
Code	`0xd`

An AMQP receiver MUST handle incoming session.gap controls.

This control is sent by the sender of commands and handled by the receiver of commands. It sends command ranges for which there will be no further data forthcoming. The receiver should proceed with the next available commands that arrive after the gap.

Arguments

Name	Type	Description	
`commands`	`commands`		**optional**
	The set of command-ids that are contained in this gap.		

Rules

Rule: gap-confirmation-and-completion

The command-ids covered by a session.gap MUST be added to the completed and confirmed sets by the receiver.

Rule: aborted-commands

If a session.gap covers a partially received command, the receiving peer MUST treat the command as aborted.

Rule: completed-or-confirmed-commands

If a session.gap covers a completed or confirmed command, the receiving peer MUST continue to treat the command as completed or confirmed.

10. Command Classes

Class: `execution`

Code	Name	Description
`0x3`	`execution`	execution commands

An AMQP server MUST implement the execution class.

An AMQP client MUST implement the execution class.

Methods

Code	Name
`0x1`	sync()
	request notification of completion for issued commands
`0x2`	result(command-id: *sequence-no*, value: *struct32*)
	carries execution results
`0x3`	exception(error-code: *error-code*, command-id: *sequence-no*, class-code: *uint8*, command-code: *uint8*, field-index: *uint8*, description: *str16*, error-info: *map*)
	notifies a peer of an execution error

The execution class provides commands that carry execution information about other model level commands.

Domain: `execution.error-code`

Name	Type	Description
`error-code`	`uint16`	

Valid Values

Value	Name	Description
403	`unauthorized-access`	The client attempted to work with a server entity to which it has no access due to security settings.
404	`not-found`	The client attempted to work with a server entity that does not exist.
405	`resource-locked`	The client attempted to work with a server entity to which it has no access because another client is working with it.
406	`precondition-failed`	The client requested a command that was not allowed because some precondition failed.
408	`resource-deleted`	A server entity the client is working with has been deleted.
409	`illegal-state`	The peer sent a command that is not permitted in the current state of the session.
503	`command-invalid`	The command segments could not be decoded.
506	`resource-limit-exceeded`	The client exceeded its resource allocation.
530	`not-allowed`	The peer tried to use a command a manner that is inconsistent with the rules described in the specification.
531	`illegal-argument`	The command argument is malformed, i.e. it does not fall within the specified domain. The illegal-argument exception can be raised on execution of any command which has domain valued fields.
540	`not-implemented`	The peer tried to use functionality that is not implemented in its partner.
541	`internal-error`	The peer could not complete the command because of an internal error. The peer may require intervention by an operator in order to resume normal operations.
542	`invalid-argument`	An invalid argument was passed to a command, and the operation could not proceed. An invalid argument is not illegal (see illegal-argument), i.e. it matches the domain definition; however the particular value is invalid in this context.

Command: `execution.sync`

Name	`sync`
Code	`0x1`

An AMQP server MUST handle incoming execution.sync commands.

An AMQP client MUST handle incoming execution.sync commands.

This command is complete when all prior commands are completed.

Command: `execution.result`

Name	`result`
Code	`0x2`

An AMQP server MUST handle incoming execution.result commands.

An AMQP client MUST handle incoming execution.result commands.

This command carries data resulting from the execution of a command.

Arguments

Name	Type	Description	
`command-id`	`sequence-no`		**required**
`value`	`struct32`		**optional**

Command: `execution.exception`

Name	`exception`
Code	`0x3`

An AMQP client MUST handle incoming execution.exception commands.

An AMQP server MUST handle incoming execution.exception commands.

This command informs a peer of an execution exception. The command-id, when given, correlates the error to a specific command.

Arguments

Name	Type	Description	
`error-code`	`error-code`	error code indicating the type of error	**required**
`command-id`	`sequence-no`	exceptional command	**optional**
	The command-id of the command which caused the exception. If the exception was not caused by a specific command, this value is not set.		
`class-code`	`uint8`	the class code of the command whose execution gave rise to the error (if appropriate)	**optional**
`command-code`	`uint8`	the class code of the command whose execution gave rise to the error (if appropriate)	**optional**
`field-index`	`uint8`	index of the exceptional field	**optional**
	The zero based index of the exceptional field within the arguments to the exceptional command. If the exception was not caused by a specific field, this value is not set.		
`description`	`str16`	descriptive text on the exception	**optional**
	The description provided is implementation defined, but MUST be in the language appropriate for the selected locale. The intention is that this description is suitable for logging or alerting output.		
`error-info`	`map`	map to carry additional information about the error	**optional**

Class: message

Code	Name	Description
0x4	message	message transfer

An AMQP server MUST implement the message class.

An AMQP client MUST implement the message class.

Methods

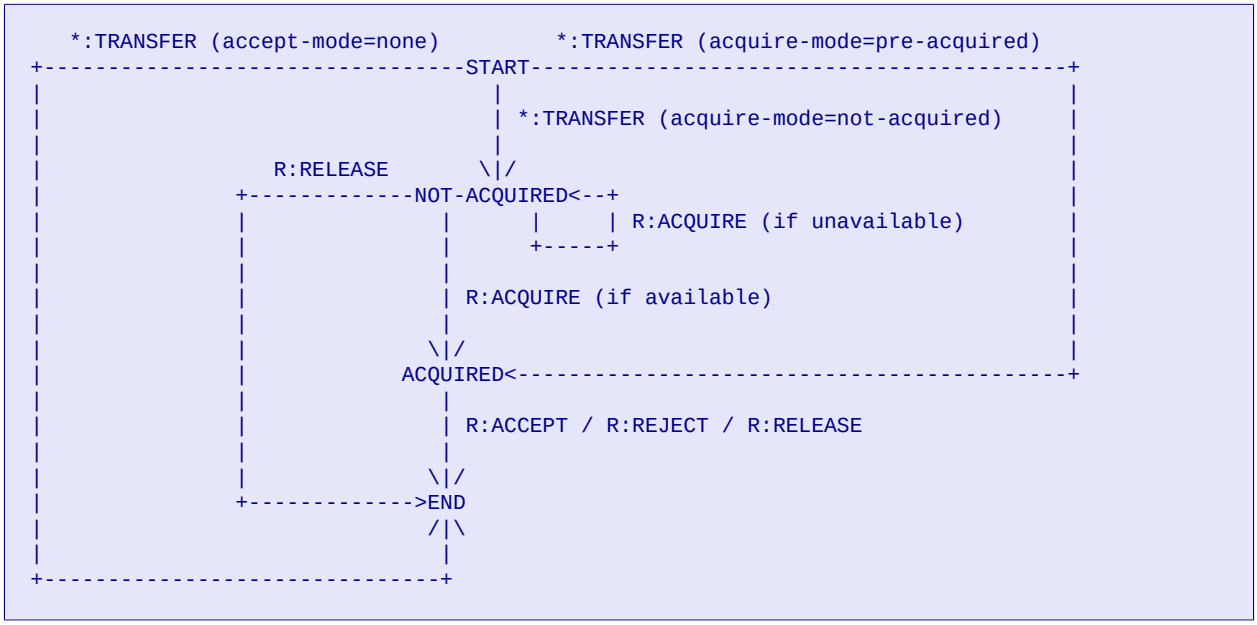
Code	Name
0x1	transfer(destination: *destination*, accept-mode: *accept-mode*, acquire-mode: *acquire-mode*)
	transfer a message
0x2	accept(transfers: *session.commands*)
	reject a message
0x3	reject(transfers: *session.commands*, code: *reject-code*, text: *str8*)
	reject a message
0x4	release(transfers: *session.commands*, set-redelivered: *bit*)
	release a message
0x5	acquire(transfers: *session.commands*)
	acquire messages for consumption
0x6	resume(destination: *destination*, resume-id: *resume-id*)
	resume an interrupted message transfer
0x7	subscribe(queue: *queue.name*, destination: *destination*, accept-mode: *accept-mode*, acquire-mode: *acquire-mode*, exclusive: *bit*, resume-id: *resume-id*, resume-ttl: *uint64*, arguments: *map*)
	start a queue subscription
0x8	cancel(destination: *destination*)
	end a queue subscription
0x9	set-flow-mode(destination: *destination*, flow-mode: *flow-mode*)
	set the flow control mode
0xa	flow(destination: *destination*, unit: *credit-unit*, value: *uint32*)
	control message flow
0xb	flush(destination: *destination*)
	force the sending of available messages
0xc	stop(destination: *destination*)
	stop the sending of messages

The message class provides commands that support an industry-standard messaging model.

Transfer States

```
    START:

      The message has yet to be sent to the recipient.

    NOT-ACQUIRED:

      The message has been sent to the recipient, but is not
      acquired by the recipient.

    ACQUIRED:

      The message has been sent to and acquired by the recipient.

    END:

      The transfer is complete.
```

State Transitions



Grammar:

```
    message = *:TRANSFER [ R:ACQUIRE ] [ R:ACCEPT / R:REJECT / R:RELEASE ]
            / *:RESUME
            / *:SET-FLOW-MODE
            / *:FLOW
            / *:STOP
            / C:SUBSCRIBE
            / C:CANCEL
            / C:FLUSH
```

Rules

Rule: persistent-message

The server SHOULD respect the delivery-mode property of messages and SHOULD make a best-effort to hold persistent messages on a reliable storage mechanism.

Scenario: Send a persistent message to queue, stop server, restart server and then verify whether message is still present. Assumes that queues are durable. Persistence without durable queues makes no sense.

Rule: no-persistent-message-discard

The server MUST NOT discard a persistent message in case of a queue overflow.

Scenario: Create a queue overflow situation with persistent messages and verify that messages do not get lost (presumably the server will write them to disk).

Rule: throttling

The server MAY use the message.flow command to slow or stop a message publisher when necessary.

Rule: non-persistent-message-overflow

The server MAY overflow non-persistent messages to persistent storage.

Rule: non-persistent-message-discard

The server MAY discard or dead-letter non-persistent messages on a priority basis if the queue size exceeds some configured limit.

Rule: min-priority-levels

The server MUST implement at least 2 priority levels for messages, where priorities 0 and 9 are treated as two distinct levels.

Rule: priority-level-implementation

The server SHOULD implement distinct priority levels in the following manner:

If the server implements n distinct priorities then priorities 0 to 5 - ceiling(n/2) should be treated equivalently and should be the lowest effective priority. The priorities 4 + floor(n/2) should be treated equivalently and should be the highest effective priority. The priorities (5 - ceiling(n/2)) to (4 + floor(n/2)) inclusive must be treated as distinct priorities.

Thus, for example, if 2 distinct priorities are implemented, then levels 0 to 4 are equivalent, and levels 5 to 9 are equivalent and levels 4 and 5 are distinct. If 3 distinct priorities are implements the 0 to 3 are equivalent, 5 to 9 are equivalent and 3, 4 and 5 are distinct.

This scheme ensures that if two priorities are distinct for a server which implements m separate priority levels they are also distinct for a server which implements n different priority levels where n > m.

Rule: priority-delivery

The server MUST deliver messages of the same priority in order irrespective of their individual persistence.

Scenario: Send a set of messages with the same priority but different persistence settings to a queue. Subscribe and verify that messages arrive in same order as originally published.

Domain: `message.delivery-properties`

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`discard-unroutable`	`bit`	controls discard of unroutable messages	**optional**
	If set on a message that is not routable the broker can discard it. If not set, an unroutable message should be handled by reject when accept-mode is explicit; or by routing to the alternate-exchange if defined when accept-mode is none.		
`immediate`	`bit`	Consider message unroutable if it cannot be processed immediately	**optional**
	If the immediate flag is set to true on a message transferred to a Server, then the message should be considered unroutable (and not delivered to any queues) if, for any queue that it is to be routed to according to the standard routing behavior, there is not a subscription on that queue able to receive the message. The treatment of unroutable messages is dependent on the value of the discard-unroutable flag. The immediate flag is ignored on transferred to a Client.		
`redelivered`	`bit`	redelivery flag	**optional**
	This boolean flag indicates that the message may have been previously delivered to this or another client. If the redelivered flag is set on transfer to a Server, then any delivery of the message from that Server to a Client must also have the redelivered flag set to true.		
`priority`	`delivery-priority`	message priority, 0 to 9	**required**
	Message priority, which can be between 0 and 9. Messages with higher priorities may be delivered before those with lower priorities.		
`delivery-mode`	`delivery-mode`	message persistence requirement	**required**
	The delivery mode may be non-persistent or persistent.		
`ttl`	`uint64`	time to live in ms	**optional**
	Duration in milliseconds for which the message should be considered "live". If this is set then a message expiration time will be computed based on the current time plus this value. Messages that live longer than their expiration time will be discarded (or dead lettered).		
`timestamp`	`datetime`	message timestamp	**optional**

Name	Type	Description	
	The timestamp is set by the broker on arrival of the message.		
expiration	datetime	message expiration time	**optional**
	The expiration header assigned by the broker. After receiving the message the broker sets expiration to the sum of the ttl specified in the publish command and the current time. (ttl=expiration - timestamp)		
exchange	exchange.name	originating exchange	**optional**
	Identifies the exchange specified in the destination field of the message.transfer used to publish the message. This MUST be set by the broker upon receipt of a message.		
routing-key	str8	message routing key	**optional**
	The value of the key determines to which queue the exchange will send the message. The way in which keys are used to make this routing decision depends on the type of exchange to which the message is sent. For example, a direct exchange will route a message to a queue if that queue is bound to the exchange with a binding-key identical to the routing-key of the message.		
resume-id	resume-id	global id for message transfer	**optional**
	When a resume-id is provided the recipient MAY use it to retain message data should the session expire while the message transfer is still incomplete.		
resume-ttl	uint64	ttl in ms for interrupted message data	**optional**
	When a resume-ttl is provided the recipient MAY use it has a guideline for how long to retain the partially complete data when a resume-id is specified. If no resume-id is specified then this value should be ignored.		

Rules

Rule: implementation

The server MUST try to signal redelivered messages when it can. When redelivering a message that was not successfully accepted, the server SHOULD deliver it to the original client if possible.

Scenario: Create a shared queue and publish a message to the queue. Subscribe using explicit accept-mode, but do not accept the message. Close the session, reconnect, and subscribe to the queue again. The message MUST arrive with the redelivered flag set.

Rule: hinting

The client should not rely on the redelivered field to detect duplicate messages where publishers may themselves produce duplicates. A fully robust client should be able to track duplicate received messages on non-transacted, and locally-transacted sessions.

Rule: ttl-decrement

If a message is transferred between brokers before delivery to a final subscriber the ttl should be decremented before peer to peer transfer and both timestamp and expiration should be cleared.

Domain: `message.fragment-properties`

These properties permit the transfer of message fragments. These may be used in conjunction with byte level flow control to limit the rate at which large messages are received. Only the first fragment carries the delivery-properties and message-properties. Syntactically each fragment appears as a complete message to the lower layers of the protocol, however the model layer is required to treat all the fragments as a single message. For example all fragments must be delivered to the same client. In pre-acquired mode, no message fragments can be delivered by the broker until the entire message has been received.

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`first`	`bit`		**default: 1**
	True if this fragment contains the start of the message, false otherwise.		
`last`	`bit`		**default: 1**
	True if this fragment contains the end of the message, false otherwise.		
`fragment-size`	`uint64`		**optional**
	This field may optionally contain the size of the fragment.		

Domain: `message.reply-to`

The reply-to domain provides a simple address structure for replying to to a message to a destination within the same virtual-host.

Struct Type

Size	Packing
2	2

Fields

Name	Type	Description	
`exchange`	`exchange.name`	the name of the exchange to reply to	**optional**
`routing-key`	`str8`	the routing-key to use when replying	**optional**

Domain: `message.message-properties`

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`content-length`	`uint64`	length of the body segment in bytes	**optional**
	The length of the body segment in bytes.		
`message-id`	`uuid`	application message identifier	**optional**
	Message-id is an optional property of UUID type which uniquely identifies a message within the message system. The message producer is usually responsible for setting the message-id. The server MAY discard a message as a duplicate if the value of the message-id matches that of a previously received message. Duplicate messages MUST still be accepted if transferred with an accept-mode of "explicit".		
`correlation-id`	`vbin16`	application correlation identifier	**optional**
	This is a client-specific id that may be used to mark or identify messages between clients. The server ignores this field.		
`reply-to`	`reply-to`	destination to reply to	**optional**
	The destination of any message that is sent in reply to this message.		
`content-type`	`str8`	MIME content type	**optional**
	The RFC-2046 MIME type for the message content (such as "text/plain"). This is set by the originating client.		
`content-encoding`	`str8`	MIME content encoding	**optional**
	The encoding for character-based message content. This is set by the originating client. Examples include UTF-8 and ISO-8859-15.		
`user-id`	`vbin16`	creating user id	**optional**
	The identity of the user responsible for producing the message. The client sets this value, and it is authenticated by the broker.		
`app-id`	`vbin16`	creating application id	**optional**
	The identity of the client application responsible for producing the message.		
`application-headers`	`map`	application specific headers table	**optional**
	This is a collection of user-defined headers or properties which may be set by the producing client and retrieved by the consuming client.		

Rules

Rule: unique

A message-id MUST be unique within a given server instance. A message-id SHOULD be globally unique (i.e. across different systems).

Rule: immutable

A message ID is immutable. Once set, a message-id MUST NOT be changed or reassigned, even if the message is replicated, resent or sent to multiple queues.

Rule: authentication

The server MUST produce an unauthorized-access exception if the user-id field is set to a principle for which the client is not authenticated.

Domain: `message.destination`

Name	Type	Description
`destination`	`str8`	destination for a message

Specifies the destination to which the message is to be transferred.

Domain: `message.accept-mode`

Name	Type	Description
`accept-mode`	`uint8`	indicates a confirmation mode

Controls how the sender of messages is notified of successful transfer.

Valid Values

Value	Name	Description
`0`	`explicit`	Successful transfer is signaled by message.accept. An acquired message (whether acquisition was implicit as in pre-acquired mode or explicit as in not-acquired mode) is not considered transferred until a message.accept that includes the transfer command is received.
`1`	`none`	Successful transfer is assumed when accept-mode is "pre-acquired". Messages transferred with an accept-mode of "not-acquired" cannot be acquired when accept-mode is "none".

Domain: `message.acquire-mode`

Name	Type	Description
`acquire-mode`	`uint8`	indicates the transfer mode

Indicates whether a transferred message can be considered as automatically acquired or whether an explicit request is necessary in order to acquire it.

Valid Values

Value	Name	Description
`0`	`pre-acquired`	the message is acquired when the transfer starts
`1`	`not-acquired`	the message is not acquired when it arrives, and must be explicitly acquired by the recipient

Domain: `message.reject-code`

Name	Type	Description
`reject-code`	`uint16`	reject code for transfer

Code specifying the reason for a message reject.

Valid Values

Value	Name	Description
`0`	`unspecified`	Rejected for an unspecified reason.
`1`	`unroutable`	Delivery was attempted but there were no queues which the message could be routed to.
`2`	`immediate`	The rejected message had the immediate flag set to true, but at the time of the transfer at least one of the queues to which it was to be routed did not have any subscriber able to take the message.

Domain: `message.resume-id`

Name	Type	Description
`resume-id`	`str16`	

A resume-id serves to identify partially transferred message content. The id is chosen by the sender, and must be unique to a given user. A resume-id is not expected to be unique across users.

Domain: `message.delivery-mode`

Name	Type	Description
`delivery-mode`	`uint8`	indicates whether a message should be treated as transient or durable

Used to set the reliability requirements for a message which is transferred to the server.

Valid Values

Value	Name	Description
1	`non-persistent`	A non-persistent message may be lost in event of a failure, but the nature of the communication is such that an occasional message loss is tolerable. This is the lowest overhead mode. Non-persistent messages are delivered at most once only.
2	`persistent`	A persistent message is one which must be stored on a persistent medium (usually hard drive) at every stage of delivery so that it will not be lost in event of failure (other than of the medium itself). This is normally accomplished with some additional overhead. A persistent message may be delivered more than once if there is uncertainty about the state of its delivery after a failure and recovery.

Domain: `message.delivery-priority`

Name	Type	Description
`delivery-priority`	`uint8`	indicates the desired priority to assign to a message transfer

Used to assign a priority to a message transfer. Priorities range from 0 (lowest) to 9 (highest).

Valid Values

Value	Name	Description
0	`lowest`	Lowest possible priority message.
1	`lower`	Very low priority message
2	`low`	Low priority message.
3	`below-average`	Below average priority message.
4	`medium`	Medium priority message.
5	`above-average`	Above average priority message
6	`high`	High priority message
7	`higher`	Higher priority message
8	`very-high`	Very high priority message.
9	`highest`	Highest possible priority message.

Domain: `message.flow-mode`

Name	Type	Description
`flow-mode`	`uint8`	the flow-mode for allocating flow credit

Valid Values

Value	Name	Description
`0`	`credit`	Credit based flow control.
`1`	`window`	Window based flow control.

Domain: `message.credit-unit`

Name	Type	Description
`credit-unit`	`uint8`	specifies the unit of credit balance

Valid Values

Value	Name	Description
`0`	`message`	Indicates a value specified in messages.
`1`	`byte`	Indicates a value specified in bytes.

Command: `message.transfer`

Name	`transfer`
Code	`0x1`

An AMQP server MUST handle incoming message.transfer commands.

An AMQP client MUST handle incoming message.transfer commands.

This command transfers a message between two peers. When a client uses this command to publish a message to a broker, the destination identifies a specific exchange. The message will then be routed to queues as defined by the exchange configuration. The client may request a broker to transfer messages to it, from a particular queue, by issuing a subscribe command. The subscribe command specifies the destination that the broker should use for any resulting transfers.

Arguments

Name	Type	Description	
`destination`	`destination`	message destination	**optional**
	Specifies the destination to which the message is to be transferred.		
`accept-mode`	`accept-mode`		**required**
	Indicates whether message.accept, session.complete, or nothing at all is required to indicate successful transfer of the message.		
`acquire-mode`	`acquire-mode`		**required**
	Indicates whether or not the transferred message has been acquired.		

Segments

Following the command segment, the following segments may follow.

header

This segment is optional.

The header segment consists of at most one of each of the following entries:

- `delivery-properties` [**optional**].
- `fragment-properties` [**optional**].
- `message-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Rules

Rule: transactional-publish

If a transfer to an exchange occurs within a transaction, then it is not available from the queue until the transaction commits. It is not specified whether routing takes place when the transfer is received or when the transaction commits.

Rule: blank-destination

The server MUST accept a blank destination to mean the default exchange.

Exceptions

Exception: nonexistent-exchange

Error:	`not-found`

If the destination refers to an exchange that does not exist, the peer MUST raise a session exception.

Command: `message.accept`

Name	`accept`
Code	`0x2`

An AMQP server MUST handle incoming message.accept commands.

An AMQP client MUST handle incoming message.accept commands.

Accepts the message. Once a transfer is accepted, the command-id may no longer be referenced from other commands.

Arguments

Name	Type	Description	
`transfers`	`session.commands`		**required**
	Identifies the messages previously transferred that should be accepted.		

Rules

Rule: acquisition

The recipient MUST have acquired a message in order to accept it.

Command: `message.reject`

Name	`reject`
Code	`0x3`

An AMQP server MUST handle incoming message.reject commands.

An AMQP client MUST handle incoming message.reject commands.

Indicates that the message transfers are unprocessable in some way. A server may reject a message if it is unroutable. A client may reject a message if it is invalid. A message may be rejected for other reasons as well. Once a transfer is rejected, the command-id may no longer be referenced from other commands.

Arguments

Name	Type	Description	
`transfers`	`session.commands`		**required**
	Identifies the messages previously transferred that should be rejected.		
`code`	`reject-code`		**required**
	Code describing the reason for rejection.		
`text`	`str8`	informational text for message reject	**optional**
	Text describing the reason for rejection.		

Rules

Rule: alternate-exchange

When a client rejects a message, the server MUST deliver that message to the alternate-exchange on the queue from which it was delivered. If no alternate-exchange is defined for that queue the broker MAY discard the message.

Rule: acquisition

The recipient MUST have acquired a message in order to reject it. If the message is not acquired any reject MUST be ignored.

Command: `message.release`

Name	`release`
Code	`0x4`

An AMQP server MUST handle incoming message.release commands.

An AMQP client MAY handle incoming message.release commands.

Release previously transferred messages. When acquired messages are released, they become available for acquisition by any subscriber. Once a transfer is released, the command-id may no longer be referenced from other commands.

Arguments

Name	Type	Description	
`transfers`	`session.commands`		**required**
	Indicates the messages to be released.		
`set-redelivered`	`bit`	mark the released messages as redelivered	**optional**
	By setting set-redelivered to true, any acquired messages released to a queue with this command will be marked as redelivered on their next transfer from that queue. If this flag is not set, then an acquired message will retain its original redelivered status on the queue. Messages that are not acquired are unaffected by the value of this flag.		

Rules

Rule: ordering

Acquired messages that have been released MAY subsequently be delivered out of order. Implementations SHOULD ensure that released messages keep their position with respect to undelivered messages of the same priority.

Command: `message.acquire`

Name	`acquire`
Code	`0x5`

An AMQP server MUST handle incoming message.acquire commands.

Acquires previously transferred messages for consumption. The acquired ids (if any) are sent via message.acquired.

Arguments

Name	Type	Description	
`transfers`	`session.commands`		**required**
	Indicates the messages to be acquired.		

Rules

Rule: one-to-one

Each acquire MUST produce exactly one message.acquired even if it is empty.

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`transfers`	`session.commands`		**required**
	Indicates the acquired messages.		

Command: message.resume

Name	resume
Code	0x6

An AMQP server MUST handle incoming message.resume commands.

An AMQP client MUST handle incoming message.resume commands.

This command resumes an interrupted transfer. The recipient should return the amount of partially transferred data associated with the given resume-id, or zero if there is no data at all. If a non-zero result is returned, the recipient should expect to receive message fragment(s) containing the remainder of the interrupted message.

Arguments

Name	Type	Description	
destination	destination		**optional**
	The destination to which the remaining message fragments are transferred.		
resume-id	resume-id		**required**
	The name of the transfer being resumed.		

Rules

Rule: unknown-resume-id

If the resume-id is not known, the recipient MUST return an offset of zero.

Exceptions

Exception: destination-not-found

Error:	not-found

If the destination does not exist, the recipient MUST close the session.

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
offset	uint64		**optional**
	Indicates the amount of data already transferred.		

Command: `message.subscribe`

Name	`subscribe`
Code	`0x7`

An AMQP server MUST handle incoming message.subscribe commands.

This command asks the server to start a "subscription", which is a request for messages from a specific queue. Subscriptions last as long as the session they were created on, or until the client cancels them.

Arguments

Name	Type	Description	
`queue`	`queue.name`		**required**
	Specifies the name of the subscribed queue.		
`destination`	`destination`	incoming message destination	**optional**
	The client specified name for the subscription. This is used as the destination for all messages transferred from this subscription. The destination is scoped to the session.		
`accept-mode`	`accept-mode`		**required**
	The accept-mode to use for messages transferred from this subscription.		
`acquire-mode`	`acquire-mode`		**required**
	The acquire-mode to use for messages transferred from this subscription.		
`exclusive`	`bit`	request exclusive access	**optional**
	Request an exclusive subscription. This prevents other subscribers from subscribing to the queue.		
`resume-id`	`resume-id`		**optional**
	Requests that the broker use the supplied resume-id when transferring messages for this subscription.		
`resume-ttl`	`uint64`		**optional**
	Requested duration in milliseconds for the broker use as resume-ttl when transferring messages for this subscription.		
`arguments`	`map`	arguments for vendor extensions	**optional**
	The syntax and semantics of these arguments depends on the providers implementation.		

Rules

Rule: simultaneous-subscriptions

The server SHOULD support at least 16 subscriptions per queue, and ideally, impose no limit except as defined by available resources.

Scenario: Create a queue and create subscriptions on that queue until the server closes the connection. Verify that the number of subscriptions created was at least sixteen and report the total number.

Rule: default-flow-mode

The default flow mode for new subscriptions is window-mode.

Rule: initial-credit

Immediately after a subscription is created, the initial byte and message credit for that destination is zero.

Exceptions

Exception: queue-deletion

Error:	`resource-deleted`
Field:	`queue`

If the queue for this subscription is deleted, any subscribing sessions MUST be closed. This exception may occur at any time after the subscription has been completed.

Exception: queue-not-found

Error:	`not-found`
Field:	`queue`

If the queue for this subscription does not exist, then the subscribing session MUST be closed.

Exception: unique-subscriber-destination

Error:	`not-allowed`

The client MUST NOT specify a destination that refers to an existing subscription on the same session.

Scenario: Attempt to create two subscriptions on the same session with the same non-empty destination.

Exception: in-use

Error:	`resource-locked`

The server MUST NOT grant an exclusive subscription to a queue that already has subscribers.

Scenario: Open two connections to a server, and in one connection create a shared (non-exclusive) queue and then subscribe to the queue. In the second connection attempt to subscribe to the same queue using the exclusive option.

Command: `message.cancel`

Name	`cancel`
Code	`0x8`

An AMQP server MUST handle incoming message.cancel commands.

This command cancels a subscription. This does not affect already delivered messages, but it does mean the server will not send any more messages for that subscription. The client may receive an arbitrary number of messages in between sending the cancel command and receiving notification that the cancel command is complete.

Arguments

Name	Type	Description	
`destination`	`destination`		**required**

Rules

Rule: post-cancel-transfer-resolution

Canceling a subscription MUST NOT affect pending transfers. A transfer made prior to canceling transfers to the destination MUST be able to be accepted, released, acquired, or rejected after the subscription is canceled.

Exceptions

Exception: subscription-not-found

Error:	`not-found`

If the subscription specified by the destination is not found, the server MUST close the session.

Command: `message.set-flow-mode`

Name	`set-flow-mode`
Code	`0x9`

An AMQP server MUST handle incoming message.set-flow-mode commands.

An AMQP client MUST handle incoming message.set-flow-mode commands.

Sets the mode of flow control used for a given destination to either window or credit based flow control. With credit based flow control, the sender of messages continually maintains its current credit balance with the recipient. The credit balance consists of two values, a message count, and a byte count. Whenever message data is sent, both counts must be decremented. If either value reaches zero, the flow of message data must stop. Additional credit is received via the message.flow command. The sender MUST NOT send partial assemblies. This means that if there is not enough byte credit available to send a complete message, the sender must either wait or use message fragmentation (see the fragment-properties header struct) to send the first part of the message data in a complete assembly. Window based flow control is identical to credit based flow control, however message transfer completion implicitly grants a single unit of message credit, and the size of the message in byte credits for each completed message transfer. Completion of the transfer command with session.completed is the only way credit is implicitly updated; message.accept, message.release, message.reject, tx.commit and tx.rollback have no effect on the outstanding credit balances.

Arguments

Name	Type	Description	
`destination`	`destination`		**optional**
`flow-mode`	`flow-mode`		**required**
	The new flow control mode.		

Rules

Rule: byte-accounting

The byte count is decremented by the payload size of each transmitted frame with segment type header or body appearing within a message.transfer command. Note that the payload size is the frame size less the frame header size.

Rule: mode-switching

Mode switching may only occur if both the byte and message credit balance are zero. There are three ways for a recipient of messages to be sure that the sender's credit balances are zero: 1) The recipient may send a message.stop command to the sender. When the recipient receives notification of completion for the message.stop command, it knows that the sender's credit is zero. 2) The recipient may perform the same steps described in (1) with the message.flush command substituted for the message.stop command. 3) Immediately after a subscription is created with message.subscribe, the credit for that destination is zero.

Rule: default-flow-mode

Prior to receiving an explicit set-flow-mode command, a peer MUST consider the flow-mode to be window.

Command: `message.flow`

Name	`flow`
Code	`0xa`

An AMQP server MUST handle incoming message.flow commands.

An AMQP client MUST handle incoming message.flow commands.

This command controls the flow of message data to a given destination. It is used by the recipient of messages to dynamically match the incoming rate of message flow to its processing or forwarding capacity. Upon receipt of this command, the sender must add "value" number of the specified unit to the available credit balance for the specified destination. A value of (0xFFFFFFFF) indicates an infinite amount of credit. This disables any limit for the given unit until the credit balance is zeroed with message.stop or message.flush.

Arguments

Name	Type	Description	
`destination`	`destination`		**optional**
`unit`	`credit-unit`		**required**
	The unit of value.		
`value`	`uint32`		**optional**
	If the value is not set then this indicates an infinite amount of credit.		

Command: `message.flush`

Name	`flush`
Code	`0xb`

An AMQP server MUST handle incoming message.flush commands.

Forces the sender to exhaust his credit supply. The sender's credit will always be zero when this command completes. The command completes when immediately available message data has been transferred, or when the credit supply is exhausted.

Arguments

Name	Type	Description	
`destination`	`destination`		**optional**

Command: `message.stop`

Name	`stop`
Code	`0xc`

An AMQP server MUST handle incoming message.stop commands.

An AMQP client MUST handle incoming message.stop commands.

On receipt of this command, a producer of messages MUST set his credit to zero for the given destination. When notifying of completion, credit MUST be zero and no further messages will be sent until such a time as further credit is received.

Arguments

Name	Type	Description	
`destination`	`destination`		**optional**

Class: tx

Code	Name	Description
0x5	tx	work with standard transactions

An AMQP server SHOULD implement the tx class.

Methods

Code	Name
0x1	select()
	select standard transaction mode
0x2	commit()
	commit the current transaction
0x3	rollback()
	abandon the current transaction

Standard transactions provide so-called "1.5 phase commit". We can ensure that work is never lost, but there is a chance of confirmations being lost, so that messages may be resent. Applications that use standard transactions must be able to detect and ignore duplicate messages.

Grammar:

```
tx  = C:SELECT
    / C:COMMIT
    / C:ROLLBACK
```

Rules

Rule: duplicate-tracking

An client using standard transactions SHOULD be able to track all messages received within a reasonable period, and thus detect and reject duplicates of the same message. It SHOULD NOT pass these to the application layer.

Command: `tx.select`

Name	`select`
Code	`0x1`

An AMQP server MUST handle incoming tx.select commands (if the tx class is implemented).

This command sets the session to use standard transactions. The client must use this command exactly once on a session before using the Commit or Rollback commands.

Exceptions

Exception: exactly-once

Error:	`illegal-state`

A client MUST NOT select standard transactions on a session that is already transactional.

Exception: no-dtx

Error:	`illegal-state`

A client MUST NOT select standard transactions on a session that is already enlisted in a distributed transaction.

Exception: explicit-accepts

Error:	`not-allowed`

On a session on which tx.select has been issued, a client MUST NOT issue a message.subscribe command with the accept-mode property set to any value other than explicit. Similarly a tx.select MUST NOT be issued on a session on which a there is a non cancelled subscriber with accept-mode of none.

Command: `tx.commit`

Name	`commit`
Code	`0x2`

An AMQP server MUST handle incoming tx.commit commands (if the tx class is implemented).

This command commits all messages published and accepted in the current transaction. A new transaction starts immediately after a commit.

In more detail, the commit acts on all messages which have been transferred from the Client to the Server, and on all acceptances of messages sent from Server to Client. Since the commit acts on commands sent in the same direction as the commit command itself, there is no ambiguity on the scope of the commands being committed. Further, the commit will not be completed until all preceding commands which it affects have been completed.

Since transactions act on explicit accept commands, the only valid accept-mode for message subscribers is explicit. For transferring messages from Client to Server (publishing) all accept-modes are permitted.

Exceptions

Exception: select-required

Error:	`illegal-state`

A client MUST NOT issue tx.commit on a session that has not been selected for standard transactions with tx.select.

Command: `tx.rollback`

Name	`rollback`
Code	`0x3`

An AMQP server MUST handle incoming tx.rollback commands (if the tx class is implemented).

This command abandons the current transaction. In particular the transfers from Client to Server (publishes) and accepts of transfers from Server to Client which occurred in the current transaction are discarded. A new transaction starts immediately after a rollback.

In more detail, when a rollback is issued, any the effects of transfers which occurred from Client to Server are discarded. The Server will issue completion notification for all such transfers prior to the completion of the rollback. Similarly the effects of any message.accept issued from Client to Server prior to the issuance of the tx.rollback will be discarded; and notification of completion for all such commands will be issued before the issuance of the completion for the rollback.

After the completion of the rollback, the client will still hold the messages which it has not yet accepted (including those for which accepts were previously issued within the transaction); i.e. the messages remain "acquired". If the Client wishes to release those messages back to the Server, then appropriate message.release commands must be issued.

Exceptions

Exception: select-required

Error:	`illegal-state`

A client MUST NOT issue tx.rollback on a session that has not been selected for standard transactions with tx.select.

Class: dtx

Code	Name	Description
0x6	dtx	Demarcates dtx branches

An AMQP server MAY implement the dtx class.

An AMQP client MAY implement the dtx class.

Methods

Code	Name
0x1	select()
	Select dtx mode
0x2	start(xid: *xid*, join: *bit*, resume: *bit*)
	Start a dtx branch
0x3	end(xid: *xid*, fail: *bit*, suspend: *bit*)
	End a dtx branch
0x4	commit(xid: *xid*, one-phase: *bit*)
	Commit work on dtx branch
0x5	forget(xid: *xid*)
	Discard dtx branch
0x6	get-timeout(xid: *xid*)
	Obtain dtx timeout in seconds
0x7	prepare(xid: *xid*)
	Prepare a dtx branch
0x8	recover()
	Get prepared or completed xids
0x9	rollback(xid: *xid*)
	Rollback a dtx branch
0xa	set-timeout(xid: *xid*, timeout: *uint32*)
	Set dtx timeout value

This provides the X-Open XA distributed transaction protocol support. It allows a session to be selected for use with distributed transactions, the transactional boundaries for work on that session to be demarcated and allows the transaction manager to coordinate transaction outcomes.

Grammar:

```
dtx-demarcation = C:SELECT *demarcation
demarcation     = C:START C:END
```

Grammar:

```
dtx-coordination    = *coordination
coordination        = command
                      / outcome
                      / recovery
command             = C:SET-TIMEOUT
                      / C:GET-TIMEOUT
outcome             = one-phase-commit
                      / one-phase-rollback
                      / two-phase-commit
                      / two-phase-rollback
one-phase-commit    = C:COMMIT
one-phase-rollback  = C:ROLLBACK
two-phase-commit    = C:PREPARE C:COMMIT
two-phase-rollback  = C:PREPARE C:ROLLBACK
recovery            = C:RECOVER *recovery-outcome
recovery-outcome    = one-phase-commit
                      / one-phase-rollback
                      / C:FORGET
```

Rules

Rule: transactionality

Enabling XA transaction support on a session requires that the server MUST manage transactions demarcated by start-end blocks. That is to say that on this XA-enabled session, work undergone within transactional blocks is performed on behalf a transaction branch whereas work performed outside of transactional blocks is NOT transactional.

Domain: `dtx.xa-result`

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`status`	`xa-status`		**required**

Domain: `dtx.xid`

An xid uniquely identifies a transaction branch.

Struct Type

Size	Packing
2	2

Fields

Name	Type	Description	
`format`	`uint32`	implementation specific format code	**required**
`global-id`	`vbin8`	global transaction id	**required**
`branch-id`	`vbin8`	branch qualifier	**required**

Domain: `dtx.xa-status`

Name	Type	Description
`xa-status`	`uint16`	XA return codes

Valid Values

Value	Name	Description
`0`	`xa-ok`	Normal execution completion (no error).
`1`	`xa-rbrollback`	The rollback was caused for an unspecified reason.
`2`	`xa-rbtimeout`	A transaction branch took too long.
`3`	`xa-heurhaz`	The transaction branch may have been heuristically completed.
`4`	`xa-heurcom`	The transaction branch has been heuristically committed.
`5`	`xa-heurrb`	The transaction branch has been heuristically rolled back.
`6`	`xa-heurmix`	The transaction branch has been heuristically committed and rolled back.
`7`	`xa-rdonly`	The transaction branch was read-only and has been committed.

Command: `dtx.select`

Name	`select`
Code	`0x1`

An AMQP server MAY handle incoming dtx.select commands (if the dtx class is implemented).

This command sets the session to use distributed transactions. The client must use this command at least once on a session before using XA demarcation operations.

Command: `dtx.start`

Name	`start`
Code	`0x2`

An AMQP server MAY handle incoming dtx.start commands (if the dtx class is implemented).

This command is called when messages should be produced and consumed on behalf a transaction branch identified by xid.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch to be started.		
`join`	`bit`	Join with existing xid flag	**optional**
	Indicate whether this is joining an already associated xid. Indicate that the start applies to joining a transaction previously seen.		
`resume`	`bit`	Resume flag	**optional**
	Indicate that the start applies to resuming a suspended transaction branch specified.		

Exceptions

Exception: illegal-state

Error:	`illegal-state`
Field:	`xid`

If the command is invoked in an improper context (see class grammar) then the server MUST send a session exception.

Exception: already-known

Error:	`not-allowed`
Field:	`xid`

If neither join nor resume is specified is specified and the transaction branch specified by xid has previously been seen then the server MUST raise an exception.

Exception: join-and-resume

Error:	`not-allowed`
Field:	`xid`

If join and resume are specified then the server MUST raise an exception.

Exception: unknown-xid

Error:	`not-allowed`

If xid is already known by the broker then the server MUST raise an exception.

Exception: unsupported

Error:	`not-implemented`

If the broker does not support join the server MUST raise an exception.

Result

Type:	xa-result
	See: Section 10.4.2, “ dtx.xa-result ”

This confirms to the client that the transaction branch is started or specify the error condition. The value of this field may be one of the following constants: xa-ok: Normal execution. xa-rbrollback: The broker marked the transaction branch rollback-only for an unspecified reason. xa-rbtimeout: The work represented by this transaction branch took too long.

Command: dtx.end

Name	end
Code	0x3

An AMQP server MAY handle incoming dtx.end commands (if the dtx class is implemented).

This command is called when the work done on behalf a transaction branch finishes or needs to be suspended.

Arguments

Name	Type	Description	
xid	xid	Transaction xid	**required**
	Specifies the xid of the transaction branch to be ended.		
fail	bit	Failure flag	**optional**
	If set, indicates that this portion of work has failed; otherwise this portion of work has completed successfully.		
suspend	bit	Temporary suspension flag	**optional**
	Indicates that the transaction branch is temporarily suspended in an incomplete state.		

Rules

Rule: success

If neither fail nor suspend are specified then the portion of work has completed successfully.

Rule: session-closed

When a session is closed then the currently associated transaction branches MUST be marked rollback-only.

Rule: failure

An implementation MAY elect to roll a transaction back if this failure notification is received. Should an implementation elect to implement this behavior, and this bit is set, then then the transaction branch SHOULD be marked as rollback-only and the end result SHOULD have the xa-rbrollback status set.

Rule: resume

The transaction context is in a suspended state and must be resumed via the start command with resume specified.

Exceptions

Exception: illegal-state

Error:	illegal-state
Field:	xid

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

Exception: suspend-and-fail

Error:	`not-allowed`
Field:	`xid`

If suspend and fail are specified then the server MUST raise an exception.

Exception: not-associated

Error:	`illegal-state`

The session MUST be currently associated with the given xid (through an earlier start call with the same xid).

Result

Type:	xa-result
	See: Section 10.4.2, “ dtx.xa-result ”

This command confirms to the client that the transaction branch is ended or specify the error condition. The value of this field may be one of the following constants: xa-ok: Normal execution. xa-rbrollback: The broker marked the transaction branch rollback-only for an unspecified reason. If an implementation chooses to implement rollback-on-failure behavior, then this value should be selected if the dtx.end.fail bit was set. xa-rbtimeout: The work represented by this transaction branch took too long.

Command: `dtx.commit`

Name	`commit`
Code	`0x4`

An AMQP server MAY handle incoming dtx.commit commands (if the dtx class is implemented).

Commit the work done on behalf a transaction branch. This command commits the work associated with xid. Any produced messages are made available and any consumed messages are discarded.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch to be committed.		
`one-phase`	`bit`	One-phase optimization flag	**optional**
	Used to indicate whether one-phase or two-phase commit is used.		

Exceptions

Exception: illegal-state

Error:	`illegal-state`
Field:	`xid`

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

Exception: unknown-xid

Error:	`not-found`

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Exception: not-disassociated

Error:	`illegal-state`

If this command is called when xid is still associated with a session then the server MUST raise an exception.

Exception: one-phase

Error:	`illegal-state`

The one-phase bit MUST be set if a commit is sent without a preceding prepare.

Exception: two-phase

Error:	`illegal-state`

The one-phase bit MUST NOT be set if the commit has been preceded by prepare.

Result

Type:	xa-result
	See: Section 10.4.2, “ dtx.xa-result ”

This confirms to the client that the transaction branch is committed or specify the error condition. The value of this field may be one of the following constants: xa-ok: Normal execution xa-heurhaz: Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed. xa-heurcom: Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed. xa-heurrb: Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back. xa-heurmix: Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back. xa-rbrollback: The broker marked the transaction branch rollback-only for an unspecified reason. xa-rbtimeout: The work represented by this transaction branch took too long.

Command: `dtx.forget`

Name	`forget`
Code	`0x5`

An AMQP server MAY handle incoming dtx.forget commands (if the dtx class is implemented).

This command is called to forget about a heuristically completed transaction branch.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch to be forgotten.		

Exceptions

Exception: illegal-state

Error:	`illegal-state`
Field:	`xid`

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

Exception: unknown-xid

Error:	`not-found`

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Exception: not-disassociated

Error:	`illegal-state`

If this command is called when xid is still associated with a session then the server MUST raise an exception.

Command: `dtx.get-timeout`

Name	`get-timeout`
Code	`0x6`

An AMQP server MAY handle incoming dtx.get-timeout commands (if the dtx class is implemented).

This command obtains the current transaction timeout value in seconds. If set-timeout was not used prior to invoking this command, the return value is the default timeout; otherwise, the value used in the previous set-timeout call is returned.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch for getting the timeout.		

Exceptions

Exception: unknown-xid

Error:	`not-found`

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`timeout`	`uint32`	The current transaction timeout value	**required**
	The current transaction timeout value in seconds.		

Command: `dtx.prepare`

Name	`prepare`
Code	`0x7`

An AMQP server MAY handle incoming dtx.prepare commands (if the dtx class is implemented).

This command prepares for commitment any message produced or consumed on behalf of xid.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch that can be prepared.		

Rules

Rule: obligation-1

Once this command successfully returns it is guaranteed that the transaction branch may be either committed or rolled back regardless of failures.

Rule: obligation-2

The knowledge of xid cannot be erased before commit or rollback complete the branch.

Exceptions

Exception: illegal-state

Error:	`illegal-state`
Field:	`xid`

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

Exception: unknown-xid

Error:	`not-found`

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Exception: not-disassociated

Error:	`illegal-state`

If this command is called when xid is still associated with a session then the server MUST raise an exception.

Result

Type:	xa-result
	See: Section 10.4.2, “ dtx.xa-result ”

This command confirms to the client that the transaction branch is prepared or specify the error condition. The value of this field may be one of the following constants: xa-ok: Normal execution. xa-rdonly: The transaction branch was read-only and has been committed. xa-rbrollback: The broker marked the transaction branch rollback-only for an unspecified reason. xa-rbtimeout: The work represented by this transaction branch took too long.

Command: `dtx.recover`

Name	`recover`
Code	`0x8`

An AMQP server MAY handle incoming dtx.recover commands (if the dtx class is implemented).

This command is called to obtain a list of transaction branches that are in a prepared or heuristically completed state.

Result

Struct Type

Size	**Packing**
4	2

Fields

Name	**Type**	**Description**	
`in-doubt`	`array`	array of xids to be recovered	**required**
	Array containing the xids to be recovered (xids that are in a prepared or heuristically completed state).		

Command: dtx.rollback

Name	rollback
Code	0x9

An AMQP server MAY handle incoming dtx.rollback commands (if the dtx class is implemented).

This command rolls back the work associated with xid. Any produced messages are discarded and any consumed messages are re-enqueued.

Arguments

Name	Type	Description	
xid	xid	Transaction xid	**required**
	Specifies the xid of the transaction branch that can be rolled back.		

Exceptions

Exception: illegal-state

Error:	illegal-state
Field:	xid

If the command is invoked in an improper context (see class grammar) then the server MUST raise an exception.

Exception: unknown-xid

Error:	not-found

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Exception: not-disassociated

Error:	illegal-state

If this command is called when xid is still associated with a session then the server MUST raise an exception.

Result

Type:	xa-result
	See: Section 10.4.2, “ dtx.xa-result ”

This command confirms to the client that the transaction branch is rolled back or specify the error condition. The value of this field may be one of the following constants: xa-ok: Normal execution xa-heurhaz: Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed. xa-heurcom: Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed. xa-heurrb: Due to a

heuristic decision, the work done on behalf of the specified transaction branch was rolled back. xa-heurmix: Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back. xa-rbrollback: The broker marked the transaction branch rollback-only for an unspecified reason. xa-rbtimeout: The work represented by this transaction branch took too long.

Command: `dtx.set-timeout`

Name	`set-timeout`
Code	`0xa`

An AMQP server MAY handle incoming dtx.set-timeout commands (if the dtx class is implemented).

Sets the specified transaction branch timeout value in seconds.

Arguments

Name	Type	Description	
`xid`	`xid`	Transaction xid	**required**
	Specifies the xid of the transaction branch for setting the timeout.		
`timeout`	`uint32`	Dtx timeout in seconds	**required**
	The transaction timeout value in seconds.		

Rules

Rule: effective

Once set, this timeout value is effective until this command is reinvoked with a different value.

Rule: reset

A value of zero resets the timeout value to the default value.

Exceptions

Exception: unknown-xid

Error:	`not-found`

If xid is unknown (the transaction branch has not been started or has already been ended) then the server MUST raise an exception.

Class: exchange

Code	Name	Description
0x7	exchange	work with exchanges

An AMQP server MUST implement the exchange class.

An AMQP client MUST implement the exchange class.

Methods

Code	Name
0x1	declare(exchange: *name*, type: *str8*, alternate-exchange: *name*, passive: *bit*, durable: *bit*, auto-delete: *bit*, arguments: *map*)
	verify exchange exists, create if needed
0x2	delete(exchange: *name*, if-unused: *bit*)
	delete an exchange
0x3	query(name: *str8*)
	request information about an exchange
0x4	bind(queue: *queue.name*, exchange: *name*, binding-key: *str8*, arguments: *map*)
	bind queue to an exchange
0x5	unbind(queue: *queue.name*, exchange: *name*, binding-key: *str8*)
	unbind a queue from an exchange
0x6	bound(exchange: *str8*, queue: *str8*, binding-key: *str8*, arguments: *map*)
	request information about bindings to an exchange

Exchanges match and distribute messages across queues. Exchanges can be configured in the server or created at runtime.

Grammar:

```
exchange  = C:DECLARE
          / C:DELETE
          / C:QUERY
```

Rules

Rule: required-types

The server MUST implement these standard exchange types: fanout, direct.

Scenario: Client attempts to declare an exchange with each of these standard types.

Rule: recommended-types

The server SHOULD implement these standard exchange types: topic, headers.

Scenario: Client attempts to declare an exchange with each of these standard types.

Rule: required-instances

The server MUST, in each virtual host, pre-declare an exchange instance for each standard exchange type that it implements, where the name of the exchange instance, if defined, is "amq." followed by the exchange type name. The server MUST, in each virtual host, pre-declare at least two direct exchange instances: one named "amq.direct", the other with no public name that serves as a default exchange for publish commands (such as message.transfer).

Scenario: Client creates a temporary queue and attempts to bind to each required exchange instance ("amq.fanout", "amq.direct", "amq.topic", and "amq.headers" if those types are defined).

Rule: default-exchange

The server MUST pre-declare a direct exchange with no public name to act as the default exchange for content publish commands (such as message.transfer) and for default queue bindings.

Scenario: Client checks that the default exchange is active by publishing a message with a suitable routing key but without specifying the exchange name, then ensuring that the message arrives in the queue correctly.

Rule: default-access

The default exchange MUST NOT be accessible to the client except by specifying an empty exchange name in a content publish command (such as message.transfer). That is, the server must not let clients explicitly bind, unbind, delete, or make any other reference to this exchange.

Rule: extensions

The server MAY implement other exchange types as wanted.

Domain: exchange.name

Name	Type	Description
name	str8	exchange name

The exchange name is a client-selected string that identifies the exchange for publish commands. Exchange names may consist of any mixture of digits, letters, and underscores. Exchange names are scoped by the virtual host.

Command: exchange.declare

Name	declare
Code	0x1

An AMQP server MUST handle incoming exchange.declare commands.

This command creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

Arguments

Name	Type	Description	
exchange	name		**required**
type	str8	exchange type	**required**
	Each exchange belongs to one of a set of exchange types implemented by the server. The exchange types define the functionality of the exchange - i.e. how messages are routed through it. It is not valid or meaningful to attempt to change the type of an existing exchange.		
alternate-exchange	name	exchange name for unroutable messages	**optional**
	In the event that a message cannot be routed, this is the name of the exchange to which the message will be sent. Messages transferred using message.transfer will be routed to the alternate-exchange only if they are sent with the "none" accept-mode, and the discard-unroutable delivery property is set to false, and there is no queue to route to for the given message according to the bindings on this exchange.		
passive	bit	do not create exchange	**optional**
	If set, the server will not create the exchange. The client can use this to check whether an exchange exists without modifying the server state.		
durable	bit	request a durable exchange	**optional**
	If set when creating a new exchange, the exchange will be marked as durable. Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged if/when a server restarts.		
auto-delete	bit	auto-delete when unused	**optional**
	If set, the exchange is deleted automatically when there remain no bindings between the exchange and any queue. Such an exchange will not be automatically deleted until at least one binding has been made to prevent the immediate deletion of the exchange upon creation.		
arguments	map	arguments for declaration	**optional**
	A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is 1.		

Rules

Rule: minimum

The server SHOULD support a minimum of 16 exchanges per virtual host and ideally, impose no limit except as defined by available resources.

Scenario: The client creates as many exchanges as it can until the server reports an error; the number of exchanges successfully created must be at least sixteen.

Rule: empty-name

If alternate-exchange is not set (its name is an empty string), unroutable messages that would be sent to the alternate-exchange MUST be dropped silently.

Rule: double-failure

A message which is being routed to a alternate exchange, MUST NOT be re-routed to a secondary alternate exchange if it fails to route in the primary alternate exchange. After such a failure, the message MUST be dropped. This prevents looping.

Rule: support

The server MUST support both durable and transient exchanges.

Rule: sticky

The server MUST ignore the durable field if the exchange already exists.

Rule: sticky

The server MUST ignore the auto-delete field if the exchange already exists.

Exceptions

Exception: reserved-names

Error:	`not-allowed`

Exchange names starting with "amq." are reserved for pre-declared and standardized exchanges. The client MUST NOT attempt to create an exchange starting with "amq.".

Exception: exchange-name-required

Error:	`invalid-argument`

The name of the exchange MUST NOT be a blank or empty string.

Exception: typed

Error:	`not-allowed`

Exchanges cannot be redeclared with different types. The client MUST NOT attempt to redeclare an existing exchange with a different type than used in the original exchange.declare command.

Exception: exchange-type-not-found

Error:	`not-found`

If the client attempts to create an exchange which the server does not recognize, an exception MUST be sent.

Exception: pre-existing-exchange

Error:	`not-allowed`

If the alternate-exchange is not empty and if the exchange already exists with a different alternate-exchange, then the declaration MUST result in an exception.

Exception: not-found

Error:	`not-found`

If set, and the exchange does not already exist, the server MUST raise an exception.

Exception: unknown-argument

Error:	`not-implemented`

If the arguments field contains arguments which are not understood by the server, it MUST raise an exception.

Command: exchange.delete

Name	delete
Code	0x2

An AMQP server MUST handle incoming exchange.delete commands.

This command deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

Arguments

Name	Type	Description	
exchange	name		**required**
if-unused	bit	delete only if unused	**optional**
	If set, the server will only delete the exchange if it has no queue bindings. If the exchange has queue bindings the server does not delete it but raises an exception instead.		

Exceptions

Exception: exists

Error:	not-found

The client MUST NOT attempt to delete an exchange that does not exist.

Exception: exchange-name-required

Error:	invalid-argument

The name of the exchange MUST NOT be a missing or empty string.

Exception: used-as-alternate

Error:	not-allowed

An exchange MUST NOT be deleted if it is in use as an alternate-exchange by a queue or by another exchange.

Exception: exchange-in-use

Error:	precondition-failed

If the exchange has queue bindings, and the if-unused flag is set, the server MUST NOT delete the exchange, but MUST raise and exception.

Command: exchange.query

Name	query
Code	0x3

An AMQP server MUST handle incoming exchange.query commands.

This command is used to request information on a particular exchange.

Arguments

Name	Type	Description	
name	str8	the exchange name	**optional**
	The name of the exchange for which information is requested. If not specified explicitly the default exchange is implied.		

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
type	str8	indicate the exchange type	**optional**
	The type of the exchange. Will be empty if the exchange is not found.		
durable	bit	indicate the durability	**optional**
	The durability of the exchange, i.e. if set the exchange is durable. Will not be set if the exchange is not found.		
not-found	bit	indicate an unknown exchange	**optional**
	If set, the exchange for which information was requested is not known.		
arguments	map	other unspecified exchange properties	**optional**
	A set of properties of the exchange whose syntax and semantics depends on the server implementation. Will be empty if the exchange is not found.		

Command: exchange.bind

Name	bind
Code	0x4

An AMQP server MUST handle incoming exchange.bind commands.

This command binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and-forward queues are bound to a direct exchange and subscription queues are bound to a topic exchange.

Arguments

Name	Type	Description	
queue	queue.name		**required**
	Specifies the name of the queue to bind.		
exchange	name	name of the exchange to bind to	**required**
binding-key	str8	identifies a binding between a given exchange and queue	**required**
	The binding-key uniquely identifies a binding between a given (exchange, queue) pair. Depending on the exchange configuration, the binding key may be matched against the message routing key in order to make routing decisions. The match algorithm depends on the exchange type. Some exchange types may ignore the binding key when making routing decisions. Refer to the specific exchange type documentation. The meaning of an empty binding key depends on the exchange implementation.		
arguments	map	arguments for binding	**optional**
	A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.		

Rules

Rule: duplicates

A server MUST ignore duplicate bindings - that is, two or more bind commands with the same exchange, queue, and binding-key - without treating these as an error. The value of the arguments used for the binding MUST NOT be altered by subsequent binding requests.

Scenario: A client binds a named queue to an exchange. The client then repeats the bind (with identical exchange, queue, and binding-key). The second binding should use a different value for the arguments field.

Rule: durable-exchange

Bindings between durable queues and durable exchanges are automatically durable and the server MUST restore such bindings after a server restart.

Scenario: A server creates a named durable queue and binds it to a durable exchange. The server is restarted. The client then attempts to use the queue/exchange combination.

Rule: binding-count

The server SHOULD support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.

Scenario: A client creates a named queue and attempts to bind it to 4 different exchanges.

Rule: multiple-bindings

Where more than one binding exists between a particular exchange instance and a particular queue instance any given message published to that exchange should be delivered to that queue at most once, regardless of how many distinct bindings match.

Scenario: A client creates a named queue and binds it to the same topic exchange at least three times using intersecting binding-keys (for example, "animals.*", "animals.dogs.*", "animal.dogs.chihuahua"). Verify that a message matching all the bindings (using previous example, routing key = "animal.dogs.chihuahua") is delivered once only.

Exceptions

Exception: empty-queue

Error:	`invalid-argument`

A client MUST NOT be allowed to bind a non-existent and unnamed queue (i.e. empty queue name) to an exchange.

Scenario: A client attempts to bind with an unnamed (empty) queue name to an exchange.

Exception: queue-existence

Error:	`not-found`

A client MUST NOT be allowed to bind a non-existent queue (i.e. not previously declared) to an exchange.

Scenario: A client attempts to bind an undeclared queue name to an exchange.

Exception: exchange-existence

Error:	`not-found`

A client MUST NOT be allowed to bind a queue to a non-existent exchange.

Scenario: A client attempts to bind a named queue to a undeclared exchange.

Exception: exchange-name-required

Error:	`invalid-argument`

The name of the exchange MUST NOT be a blank or empty string.

Exception: unknown-argument

Error:	`not-implemented`

If the arguments field contains arguments which are not understood by the server, it MUST raise an exception.

Command: exchange.unbind

Name	unbind
Code	0x5

An AMQP server MUST handle incoming exchange.unbind commands.

This command unbinds a queue from an exchange.

Arguments

Name	Type	Description	
queue	queue.name		**required**
	Specifies the name of the queue to unbind.		
exchange	name		**required**
	The name of the exchange to unbind from.		
binding-key	str8	the key of the binding	**required**
	Specifies the binding-key of the binding to unbind.		

Exceptions

Exception: non-existent-queue

Error:	not-found

If the queue does not exist the server MUST raise an exception.

Exception: non-existent-exchange

Error:	not-found

If the exchange does not exist the server MUST raise an exception.

Exception: exchange-name-required

Error:	invalid-argument

The name of the exchange MUST NOT be a blank or empty string.

Exception: non-existent-binding-key

Error:	not-found

If there is no matching binding-key the server MUST raise an exception.

Command: exchange.bound

Name	bound
Code	0x6

An AMQP server MUST handle incoming exchange.bound commands.

This command is used to request information on the bindings to a particular exchange.

Arguments

Name	Type	Description	
exchange	str8	the exchange name	**optional**
	The name of the exchange for which binding information is being requested. If not specified explicitly the default exchange is implied.		
queue	str8	a queue name	**required**
	If populated then determine whether the given queue is bound to the exchange.		
binding-key	str8	a binding-key	**optional**
	If populated defines the binding-key of the binding of interest, if not populated the request will ignore the binding-key on bindings when searching for a match.		
arguments	map	a set of binding arguments	**optional**
	If populated defines the arguments of the binding of interest if not populated the request will ignore the arguments on bindings when searching for a match		

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
exchange-not-found	bit	indicate an unknown exchange	**optional**
	If set, the exchange for which information was requested is not known.		
queue-not-found	bit	indicate an unknown queue	**optional**
	If set, the queue specified is not known.		
queue-not-matched	bit	indicate no matching queue	**optional**
	A bit which if set indicates that no binding was found from the specified exchange to the specified queue.		
key-not-matched	bit	indicate no matching binding-key	**optional**
	A bit which if set indicates that no binding was found from the specified exchange with the specified binding-key.		
args-not-matched	bit	indicate no matching arguments	**optional**
	A bit which if set indicates that no binding was found from the specified exchange with the specified arguments.		

Class: queue

Code	Name	Description
0x8	queue	work with queues

An AMQP server MUST implement the queue class.

An AMQP client MUST implement the queue class.

Methods

Code	Name
0x1	declare(queue: *name*, alternate-exchange: *exchange.name*, passive: *bit*, durable: *bit*, exclusive: *bit*, auto-delete: *bit*, arguments: *map*)
	declare queue
0x2	delete(queue: *name*, if-unused: *bit*, if-empty: *bit*)
	delete a queue
0x3	purge(queue: *name*)
	purge a queue
0x4	query(queue: *name*)
	request information about a queue

Queues store and forward messages. Queues can be configured in the server or created at runtime. Queues must be attached to at least one exchange in order to receive messages from publishers.

Grammar:

```
queue = C:DECLARE
      / C:BIND
      / C:PURGE
      / C:DELETE
      / C:QUERY
      / C:UNBIND
```

Rules

Rule: any-content

A server MUST allow any content class to be sent to any queue, in any mix, and queue and deliver these content classes independently. Note that all commands that fetch content off queues are specific to a given content class.

Scenario: Client creates an exchange of each standard type and several queues that it binds to each exchange. It must then successfully send each of the standard content types to each of the available queues.

Domain: queue.name

Name	Type	Description
name	str8	queue name

The queue name identifies the queue within the virtual host. Queue names must have a length of between 1 and 255 characters inclusive, must start with a digit, letter or underscores ('_') character, and must be otherwise encoded in UTF-8.

Command: queue.declare

Name	declare
Code	0x1

An AMQP server MUST handle incoming queue.declare commands.

This command creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

Arguments

Name	Type	Description	
queue	name		**required**
alternate-exchange	exchange.name	exchange name for messages with exceptions	**optional**
	The alternate-exchange field specifies how messages on this queue should be treated when they are rejected by a subscriber, or when they are orphaned by queue deletion. When present, rejected or orphaned messages MUST be routed to the alternate-exchange. In all cases the messages MUST be removed from the queue.		
passive	bit	do not create queue	**optional**
	If set, the server will not create the queue. This field allows the client to assert the presence of a queue without modifying the server state.		
durable	bit	request a durable queue	**optional**
	If set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.		
exclusive	bit	request an exclusive queue	**optional**
	Exclusive queues can only be used from one session at a time. Once a session declares an exclusive queue, that queue cannot be used by any other session until the declaring session closes.		
auto-delete	bit	auto-delete queue when unused	**optional**
	If this field is set and the exclusive field is also set, then the queue MUST be deleted when the session closes. If this field is set and the exclusive field is not set the queue is deleted when all the consumers have finished using it. Last consumer can be cancelled either explicitly or because its session is closed. If there was no consumer ever on the queue, it won't be deleted.		
arguments	map	arguments for declaration	**optional**
	A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is 1.		

Rules

Rule: default-binding

The server MUST create a default binding for a newly-created queue to the default exchange, which is an exchange of type 'direct' and use the queue name as the binding-key.

Scenario: Client creates a new queue, and then without explicitly binding it to an exchange, attempts to send a message through the default exchange binding, i.e. publish a message to the empty exchange, with the queue name as binding-key.

Rule: minimum-queues

The server SHOULD support a minimum of 256 queues per virtual host and ideally, impose no limit except as defined by available resources.

Scenario: Client attempts to create as many queues as it can until the server reports an error. The resulting count must at least be 256.

Rule: persistence

The queue definition MUST survive the server losing all transient memory, e.g. a machine restart.

Scenario: Client creates a durable queue; server is then restarted. Client then attempts to send message to the queue. The message should be successfully delivered.

Rule: types

The server MUST support both durable and transient queues.

Scenario: A client creates two named queues, one durable and one transient.

Rule: pre-existence

The server MUST ignore the durable field if the queue already exists.

Scenario: A client creates two named queues, one durable and one transient. The client then attempts to declare the two queues using the same names again, but reversing the value of the durable flag in each case. Verify that the queues still exist with the original durable flag values.

Rule: types

The server MUST support both exclusive (private) and non-exclusive (shared) queues.

Scenario: A client creates two named queues, one exclusive and one non-exclusive.

Rule: pre-existence

The server MUST ignore the auto-delete field if the queue already exists.

Scenario: A client creates two named queues, one as auto-delete and one explicit-delete. The client then attempts to declare the two queues using the same names again, but reversing the value of the auto-delete field in each case. Verify that the queues still exist with the original auto-delete flag values.

Exceptions

Exception: reserved-prefix

Error:	`not-allowed`

Queue names starting with "amq." are reserved for pre-declared and standardized server queues. A client MUST NOT attempt to declare a queue with a name that starts with "amq." and the passive option set to zero.

Scenario: A client attempts to create a queue with a name starting with "amq." and with the passive option set to zero.

Exception: pre-existing-exchange

Error:	`not-allowed`

If the alternate-exchange is not empty and if the queue already exists with a different alternate-exchange, then the declaration MUST result in an exception.

Exception: unknown-exchange

Error:	`not-found`

if the alternate-exchange does not match the name of any existing exchange on the server, then an exception must be raised.

Exception: passive

Error:	`not-found`

The client MAY ask the server to assert that a queue exists without creating the queue if not. If the queue does not exist, the server treats this as a failure.

Scenario: Client declares an existing queue with the passive option and expects the command to succeed. Client then attempts to declare a non-existent queue with the passive option, and the server must close the session with the correct exception.

Exception: in-use

Error:	`resource-locked`

If the server receives a declare, bind, consume or get request for a queue that has been declared as exclusive by an existing client session, it MUST raise an exception.

Scenario: A client declares an exclusive named queue. A second client on a different session attempts to declare a queue of the same name.

Exception: unknown-argument

Error:	`not-implemented`

If the arguments field contains arguments which are not understood by the server, it MUST raise an exception.

Command: queue.delete

Name	delete
Code	0x2

An AMQP server MUST handle incoming queue.delete commands.

This command deletes a queue. When a queue is deleted any pending messages are sent to the alternate-exchange if defined, or discarded if it is not.

Arguments

Name	Type	Description	
queue	name		**required**
	Specifies the name of the queue to delete.		
if-unused	bit	delete only if unused	**optional**
	If set, the server will only delete the queue if it has no consumers. If the queue has consumers the server does does not delete it but raises an exception instead.		
if-empty	bit	delete only if empty	**optional**
	If set, the server will only delete the queue if it has no messages.		

Exceptions

Exception: empty-name

Error:	invalid-argument

If the queue name in this command is empty, the server MUST raise an exception.

Exception: queue-exists

Error:	not-found

The queue must exist. If the client attempts to delete a non-existing queue the server MUST raise an exception.

Exception: if-unused-flag

Error:	precondition-failed

The server MUST respect the if-unused flag when deleting a queue.

Exception: not-empty

Error:	precondition-failed

If the queue is not empty the server MUST raise an exception.

Command: queue.purge

Name	purge
Code	0x3

An AMQP server MUST handle incoming queue.purge commands.

This command removes all messages from a queue. It does not cancel subscribers. Purged messages are deleted without any formal "undo" mechanism.

Arguments

Name	Type	Description	
queue	name		**required**
	Specifies the name of the queue to purge.		

Rules

Rule: empty

A call to purge MUST result in an empty queue.

Rule: pending-messages

The server MUST NOT purge messages that have already been sent to a client but not yet accepted.

Rule: purge-recovery

The server MAY implement a purge queue or log that allows system administrators to recover accidentally-purged messages. The server SHOULD NOT keep purged messages in the same storage spaces as the live messages since the volumes of purged messages may get very large.

Exceptions

Exception: empty-name

Error:	invalid-argument

If the the queue name in this command is empty, the server MUST raise an exception.

Exception: queue-exists

Error:	not-found

The queue MUST exist. Attempting to purge a non-existing queue MUST cause an exception.

Command: queue.query

Name	query
Code	0x4

An AMQP server MUST handle incoming queue.query commands.

This command requests information about a queue.

Arguments

Name	Type	Description	
queue	name	the queried queue	**required**

Result

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
queue	name		**required**
	Reports the name of the queue.		
alternate-exchange	exchange.name		**optional**
durable	bit		**optional**
exclusive	bit		**optional**
auto-delete	bit		**optional**
arguments	map		**optional**
message-count	uint32	number of messages in queue	**required**
	Reports the number of messages in the queue.		
subscriber-count	uint32	number of subscribers	**required**
	Reports the number of subscribers for the queue.		

Class: `file`

Code	Name	Description
`0x9`	`file`	work with file content

An AMQP server MAY implement the file class.

An AMQP client MAY implement the file class.

Methods

Code	Name
`0x1`	qos(prefetch-size: *uint32*, prefetch-count: *uint16*, global: *bit*)
	specify quality of service
`0x2`	qos-ok()
	confirm the requested qos
`0x3`	consume(queue: *queue.name*, consumer-tag: *str8*, no-local: *bit*, no-ack: *bit*, exclusive: *bit*, nowait: *bit*, arguments: *map*)
	start a queue consumer
`0x4`	consume-ok(consumer-tag: *str8*)
	confirm a new consumer
`0x5`	cancel(consumer-tag: *str8*)
	end a queue consumer
`0x6`	open(identifier: *str8*, content-size: *uint64*)
	request to start staging
`0x7`	open-ok(staged-size: *uint64*)
	confirm staging ready
`0x8`	stage()
	stage message content
`0x9`	publish(exchange: *exchange.name*, routing-key: *str8*, mandatory: *bit*, immediate: *bit*, identifier: *str8*)
	publish a message
`0xa`	return(reply-code: *return-code*, reply-text: *str8*, exchange: *exchange.name*, routing-key: *str8*)
	return a failed message
`0xb`	deliver(consumer-tag: *str8*, delivery-tag: *uint64*, redelivered: *bit*, exchange: *exchange.name*, routing-key: *str8*, identifier: *str8*)
	notify the client of a consumer message
`0xc`	ack(delivery-tag: *uint64*, multiple: *bit*)
	acknowledge one or more messages
`0xd`	reject(delivery-tag: *uint64*, requeue: *bit*)
	reject an incoming message

The file class provides commands that support reliable file transfer. File messages have a specific set of properties that are required for interoperability with file transfer applications. File messages and acknowledgements are subject to

session transactions. Note that the file class does not provide message browsing commands; these are not compatible with the staging model. Applications that need browsable file transfer should use Message content and the Message class.

Grammar:

```
file  = C:QOS S:QOS-OK
      / C:CONSUME S:CONSUME-OK
      / C:CANCEL
      / C:OPEN S:OPEN-OK C:STAGE content
      / S:OPEN C:OPEN-OK S:STAGE content
      / C:PUBLISH
      / S:DELIVER
      / S:RETURN
      / C:ACK
      / C:REJECT
```

Rules

Rule: reliable-storage

The server MUST make a best-effort to hold file messages on a reliable storage mechanism.

Rule: no-discard

The server MUST NOT discard a file message in case of a queue overflow. The server MUST use the Session.Flow command to slow or stop a file message publisher when necessary.

Rule: priority-levels

The server MUST implement at least 2 priority levels for file messages, where priorities 0-4 and 5-9 are treated as two distinct levels. The server MAY implement up to 10 priority levels.

Rule: acknowledgement-support

The server MUST support both automatic and explicit acknowledgements on file content.

Domain: `file.file-properties`

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`content-type`	`str8`	MIME content type	**optional**
`content-encoding`	`str8`	MIME content encoding	**optional**
`headers`	`map`	message header field table	**optional**
`priority`	`uint8`	message priority, 0 to 9	**optional**
`reply-to`	`str8`	destination to reply to	**optional**
`message-id`	`str8`	application message identifier	**optional**
`filename`	`str8`	message filename	**optional**
`timestamp`	`datetime`	message timestamp	**optional**
`cluster-id`	`str8`	intra-cluster routing identifier	**optional**

Domain: `file.return-code`

Name	Type	Description
`return-code`	`uint16`	return code from server

The return code. The AMQP return codes are defined by this enum.

Valid Values

Value	Name	Description
311	`content-too-large`	The client attempted to transfer content larger than the server could accept.
312	`no-route`	The exchange cannot route a message, most likely due to an invalid routing key. Only when the mandatory flag is set.
313	`no-consumers`	The exchange cannot deliver to a consumer when the immediate flag is set. As a result of pending data on the queue or the absence of any consumers of the queue.

Command: `file.qos`

Name	`qos`
Code	`0x1`
Response	qos-ok

An AMQP server MUST handle incoming file.qos commands (if the file class is implemented).

This command requests a specific quality of service. The QoS can be specified for the current session or for all sessions on the connection. The particular properties and semantics of a qos command always depend on the content class semantics. Though the qos command could in principle apply to both peers, it is currently meaningful only for the server.

Arguments

Name	Type	Description	
`prefetch-size`	`uint32`	pre-fetch window in octets	**optional**
	The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent within the session. Pre-fetching gives a performance improvement. This field specifies the pre-fetch window size in octets. May be set to zero, meaning "no specific limit". Note that other pre-fetch limits may still apply. The prefetch-size is ignored if the no-ack option is set.		
`prefetch-count`	`uint16`	pre-fetch window in messages	**optional**
	Specifies a pre-fetch window in terms of whole messages. This is compatible with some file API implementations. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both pre-fetch windows (and those at the session and connection level) allow it. The prefetch-count is ignored if the no-ack option is set.		
`global`	`bit`	apply to entire connection	**optional**
	By default the QoS settings apply to the current session only. If this field is set, they are applied to the entire connection.		

Rules

Rule: prefetch-discretion

The server MAY send less data in advance than allowed by the client's specified pre-fetch windows but it MUST NOT send more.

Command: `file.qos-ok`

Name	`qos-ok`
Code	`0x2`

An AMQP client MUST handle incoming file.qos-ok commands (if the file class is implemented).

This command tells the client that the requested QoS levels could be handled by the server. The requested QoS applies to all active consumers until a new QoS is defined.

Command: `file.consume`

Name	`consume`
Code	`0x3`
Response	consume-ok

An AMQP server MUST handle incoming file.consume commands (if the file class is implemented).

This command asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the session they were created on, or until the client cancels them.

Arguments

Name	Type	Description	
`queue`	`queue.name`		**optional**
	Specifies the name of the queue to consume from.		
`consumer-tag`	`str8`		**optional**
	Specifies the identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags.		
`no-local`	`bit`		**optional**
	If the no-local field is set the server will not send messages to the connection that published them.		
`no-ack`	`bit`	no acknowledgement needed	**optional**
	If this field is set the server does not expect acknowledgements for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.		
`exclusive`	`bit`	request exclusive access	**optional**
	Request exclusive consumer access, meaning only this consumer can access the queue.		
`nowait`	`bit`	do not send a reply command	**optional**
	If set, the server will not respond to the command. The client should not wait for a reply command. If the server could not complete the command it will raise an exception.		
`arguments`	`map`	arguments for consuming	**optional**
	A set of arguments for the consume. The syntax and semantics of these arguments depends on the providers implementation.		

Rules

Rule: min-consumers

The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.

Exceptions

Exception: queue-exists-if-empty

Error:	`not-allowed`

If the queue name in this command is empty, the server MUST raise an exception.

Exception: not-existing-consumer

Error:	`not-allowed`

The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise an exception.

Exception: not-empty-consumer-tag

Error:	`not-allowed`

The client MUST NOT specify a tag that is empty or blank.

Scenario: Attempt to create a consumers with an empty tag.

Exception: in-use

Error:	`resource-locked`

If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise an exception.

Command: `file.consume-ok`

Name	`consume-ok`
Code	`0x4`

An AMQP client MUST handle incoming file.consume-ok commands (if the file class is implemented).

This command provides the client with a consumer tag which it MUST use in commands that work with the consumer.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**
	Holds the consumer tag specified by the client or provided by the server.		

Command: `file.cancel`

Name	`cancel`
Code	`0x5`

An AMQP server MUST handle incoming file.cancel commands (if the file class is implemented).

This command cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**
	the identifier of the consumer to be cancelled.		

Command: `file.open`

Name	`open`
Code	`0x6`
Response	open-ok

An AMQP server MUST handle incoming file.open commands (if the file class is implemented).

An AMQP client MUST handle incoming file.open commands (if the file class is implemented).

This command requests permission to start staging a message. Staging means sending the message into a temporary area at the recipient end and then delivering the message by referring to this temporary area. Staging is how the protocol handles partial file transfers - if a message is partially staged and the connection breaks, the next time the sender starts to stage it, it can restart from where it left off.

Arguments

Name	Type	Description	
`identifier`	`str8`	staging identifier	**optional**
	This is the staging identifier. This is an arbitrary string chosen by the sender. For staging to work correctly the sender must use the same staging identifier when staging the same message a second time after recovery from a failure. A good choice for the staging identifier would be the SHA1 hash of the message properties data (including the original filename, revised time, etc.).		
`content-size`	`uint64`	message content size	**optional**
	The size of the content in octets. The recipient may use this information to allocate or check available space in advance, to avoid "disk full" errors during staging of very large messages.		

Rules

Rule: content-size

The sender MUST accurately fill the content-size field. Zero-length content is permitted.

Command: `file.open-ok`

Name	`open-ok`
Code	`0x7`
Response	stage

An AMQP server MUST handle incoming file.open-ok commands (if the file class is implemented).

An AMQP client MUST handle incoming file.open-ok commands (if the file class is implemented).

This command confirms that the recipient is ready to accept staged data. If the message was already partially-staged at a previous time the recipient will report the number of octets already staged.

Arguments

Name	Type	Description	
`staged-size`	`uint64`	already staged amount	**optional**
	The amount of previously-staged content in octets. For a new message this will be zero.		

Rules

Rule: behavior

The sender MUST start sending data from this octet offset in the message, counting from zero.

Rule: staging

The recipient MAY decide how long to hold partially-staged content and MAY implement staging by always discarding partially-staged content. However if it uses the file content type it MUST support the staging commands.

Command: `file.stage`

Name	`stage`
Code	`0x8`

An AMQP server MUST handle incoming file.stage commands (if the file class is implemented).

An AMQP client MUST handle incoming file.stage commands (if the file class is implemented).

This command stages the message, sending the message content to the recipient from the octet offset specified in the Open-Ok command.

Segments

Following the command segment, the following segments may follow.

header

This segment MUST be present.

The header segment consists of at most one of each of the following entries:

- `file-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Command: `file.publish`

Name	`publish`
Code	`0x9`

An AMQP server MUST handle incoming file.publish commands (if the file class is implemented).

This command publishes a staged file message to a specific exchange. The file message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

Arguments

Name	Type	Description	
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange to publish to. The exchange name can be empty, meaning the default exchange. If the exchange name is specified, and that exchange does not exist, the server will raise an exception.		
`routing-key`	`str8`	Message routing key	**optional**
	Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.		
`mandatory`	`bit`	indicate mandatory routing	**optional**
	This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set, the server will return an unroutable message with a Return command. If this flag is zero, the server silently drops the message.		
`immediate`	`bit`	request immediate delivery	**optional**
	This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable message with a Return command. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed.		
`identifier`	`str8`	staging identifier	**optional**
	This is the staging identifier of the message to publish. The message must have been staged. Note that a client can send the Publish command asynchronously without waiting for staging to finish.		

Rules

Rule: default

The server MUST accept a blank exchange name to mean the default exchange.

Rule: implementation

The server SHOULD implement the mandatory flag.

Rule: implementation

The server SHOULD implement the immediate flag.

Exceptions

Exception: refusal

Error:	`not-implemented`

The exchange MAY refuse file content in which case it MUST send an exception.

Command: `file.return`

Name	`return`
Code	`0xa`

An AMQP client MUST handle incoming file.return commands (if the file class is implemented).

This command returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

Arguments

Name	Type	Description	
`reply-code`	`return-code`		**optional**
`reply-text`	`str8`	The localized reply text.	**optional**
	This text can be logged as an aid to resolving issues.		
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange that the message was originally published to.		
`routing-key`	`str8`	Message routing key	**optional**
	Specifies the routing key name specified when the message was published.		

Segments

Following the command segment, the following segments may follow.

header

This segment MUST be present.

The header segment consists of at most one of each of the following entries:

- `file-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Command: `file.deliver`

Name	`deliver`
Code	`0xb`

An AMQP client MUST handle incoming file.deliver commands (if the file class is implemented).

This command delivers a staged file message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the consume command, then the server responds with Deliver commands as and when messages arrive for that consumer.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**
`delivery-tag`	`uint64`		**optional**
	The server-assigned and session-specific delivery tag		
`redelivered`	`bit`	Indicate possible duplicate delivery	**optional**
	This boolean flag indicates that the message may have been previously delivered to this or another client.		
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange that the message was originally published to.		
`routing-key`	`str8`	Message routing key	**optional**
	Specifies the routing key name specified when the message was published.		
`identifier`	`str8`	staging identifier	**optional**
	This is the staging identifier of the message to deliver. The message must have been staged. Note that a server can send the Deliver command asynchronously without waiting for staging to finish.		

Rules

Rule: redelivery-tracking

The server SHOULD track the number of times a message has been delivered to clients and when a message is redelivered a certain number of times - e.g. 5 times - without being acknowledged, the server SHOULD consider the message to be non-processable (possibly causing client applications to abort), and move the message to a dead letter queue.

Rule: non-zero

The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning "all messages so far received".

Command: `file.ack`

Name	`ack`
Code	`0xc`

An AMQP server MUST handle incoming file.ack commands (if the file class is implemented).

This command acknowledges one or more messages delivered via the Deliver command. The client can ask to confirm a single message or a set of messages up to and including a specific message.

Arguments

Name	Type	Description	
`delivery-tag`	`uint64`		**optional**
	The identifier of the message being acknowledged		
`multiple`	`bit`	acknowledge multiple messages	**optional**
	If set to 1, the delivery tag is treated as "up to and including", so that the client can acknowledge multiple messages with a single command. If set to zero, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, tells the server to acknowledge all outstanding messages.		

Rules

Rule: session-local

The delivery tag is valid only within the session from which the message was received. i.e. A client MUST NOT receive a message on one session and then acknowledge it on another.

Rule: validation

The server MUST validate that a non-zero delivery-tag refers to an delivered message, and raise an exception if this is not the case.

Command: `file.reject`

Name	`reject`
Code	`0xd`

An AMQP server MUST handle incoming file.reject commands (if the file class is implemented).

This command allows a client to reject a message. It can be used to return untreatable messages to their original queue. Note that file content is staged before delivery, so the client will not use this command to interrupt delivery of a large message.

Arguments

Name	Type	Description	
`delivery-tag`	`uint64`		**optional**
	the identifier of the message to be rejected		
`requeue`	`bit`	requeue the message	**optional**
	If this field is zero, the message will be discarded. If this bit is 1, the server will attempt to requeue the message.		

Rules

Rule: server-interpretation

The server SHOULD interpret this command as meaning that the client is unable to process the message at this time.

Rule: not-selection

A client MUST NOT use this command as a means of selecting messages to process. A rejected message MAY be discarded or dead-lettered, not necessarily passed to another client.

Rule: session-local

The delivery tag is valid only within the session from which the message was received. i.e. A client MUST NOT receive a message on one session and then reject it on another.

Rule: requeue-strategy

The server MUST NOT deliver the message to the same client within the context of the current session. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

Class: `stream`

Code	Name	Description
`0xa`	`stream`	work with streaming content

An AMQP server MAY implement the stream class.

An AMQP client MAY implement the stream class.

Methods

Code	Name
`0x1`	qos(prefetch-size: *uint32*, prefetch-count: *uint16*, consume-rate: *uint32*, global: *bit*)
	specify quality of service
`0x2`	qos-ok()
	confirm the requested qos
`0x3`	consume(queue: *queue.name*, consumer-tag: *str8*, no-local: *bit*, exclusive: *bit*, nowait: *bit*, arguments: *map*)
	start a queue consumer
`0x4`	consume-ok(consumer-tag: *str8*)
	confirm a new consumer
`0x5`	cancel(consumer-tag: *str8*)
	end a queue consumer
`0x6`	publish(exchange: *exchange.name*, routing-key: *str8*, mandatory: *bit*, immediate: *bit*)
	publish a message
`0x7`	return(reply-code: *return-code*, reply-text: *str8*, exchange: *exchange.name*, routing-key: *str8*)
	return a failed message
`0x8`	deliver(consumer-tag: *str8*, delivery-tag: *uint64*, exchange: *exchange.name*, queue: *queue.name*)
	notify the client of a consumer message

The stream class provides commands that support multimedia streaming. The stream class uses the following semantics: one message is one packet of data; delivery is unacknowledged and unreliable; the consumer can specify quality of service parameters that the server can try to adhere to; lower-priority messages may be discarded in favor of high priority messages.

Grammar:

```
stream  = C:QOS S:QOS-OK
        / C:CONSUME S:CONSUME-OK
        / C:CANCEL
        / C:PUBLISH content
        / S:RETURN
        / S:DELIVER content
```

Rules

Rule: overflow-discard

The server SHOULD discard stream messages on a priority basis if the queue size exceeds some configured limit.

Rule: priority-levels

The server MUST implement at least 2 priority levels for stream messages, where priorities 0-4 and 5-9 are treated as two distinct levels. The server MAY implement up to 10 priority levels.

Rule: acknowledgement-support

The server MUST implement automatic acknowledgements on stream content. That is, as soon as a message is delivered to a client via a Deliver command, the server must remove it from the queue.

Domain: `stream.stream-properties`

Struct Type

Size	Packing
4	2

Fields

Name	Type	Description	
`content-type`	`str8`	MIME content type	**optional**
`content-encoding`	`str8`	MIME content encoding	**optional**
`headers`	`map`	message header field table	**optional**
`priority`	`uint8`	message priority, 0 to 9	**optional**
`timestamp`	`datetime`	message timestamp	**optional**

Domain: `stream.return-code`

Name	Type	Description
`return-code`	`uint16`	return code from server

The return code. The AMQP return codes are defined by this enum.

Valid Values

Value	Name	Description
311	`content-too-large`	The client attempted to transfer content larger than the server could accept.
312	`no-route`	The exchange cannot route a message, most likely due to an invalid routing key. Only when the mandatory flag is set.
313	`no-consumers`	The exchange cannot deliver to a consumer when the immediate flag is set. As a result of pending data on the queue or the absence of any consumers of the queue.

Command: `stream.qos`

Name	`qos`
Code	`0x1`
Response	qos-ok

An AMQP server MUST handle incoming stream.qos commands (if the stream class is implemented).

This command requests a specific quality of service. The QoS can be specified for the current session or for all sessions on the connection. The particular properties and semantics of a qos command always depend on the content class semantics. Though the qos command could in principle apply to both peers, it is currently meaningful only for the server.

Arguments

Name	Type	Description	
`prefetch-size`	`uint32`	pre-fetch window in octets	**optional**
	The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent within the session. Pre-fetching gives a performance improvement. This field specifies the pre-fetch window size in octets. May be set to zero, meaning "no specific limit". Note that other pre-fetch limits may still apply.		
`prefetch-count`	`uint16`	pre-fetch window in messages	**optional**
	Specifies a pre-fetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both pre-fetch windows (and those at the session and connection level) allow it.		
`consume-rate`	`uint32`	transfer rate in octets/second	**optional**
	Specifies a desired transfer rate in octets per second. This is usually determined by the application that uses the streaming data. A value of zero means "no limit", i.e. as rapidly as possible.		
`global`	`bit`	apply to entire connection	**optional**
	By default the QoS settings apply to the current session only. If this field is set, they are applied to the entire connection.		

Rules

Rule: ignore-prefetch

The server MAY ignore the pre-fetch values and consume rates, depending on the type of stream and the ability of the server to queue and/or reply it.

Rule: drop-by-priority

The server MAY drop low-priority messages in favor of high-priority messages.

Command: `stream.qos-ok`

Name	`qos-ok`
Code	`0x2`

An AMQP client MUST handle incoming stream.qos-ok commands (if the stream class is implemented).

This command tells the client that the requested QoS levels could be handled by the server. The requested QoS applies to all active consumers until a new QoS is defined.

Command: `stream.consume`

Name	`consume`
Code	`0x3`
Response	consume-ok

An AMQP server MUST handle incoming stream.consume commands (if the stream class is implemented).

This command asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the session they were created on, or until the client cancels them.

Arguments

Name	Type	Description	
`queue`	`queue.name`		**optional**
	Specifies the name of the queue to consume from.		
`consumer-tag`	`str8`		**optional**
	Specifies the identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags.		
`no-local`	`bit`		**optional**
	If the no-local field is set the server will not send messages to the connection that published them.		
`exclusive`	`bit`	request exclusive access	**optional**
	Request exclusive consumer access, meaning only this consumer can access the queue.		
`nowait`	`bit`	do not send a reply command	**optional**
	If set, the server will not respond to the command. The client should not wait for a reply command. If the server could not complete the command it will raise an exception.		
`arguments`	`map`	arguments for consuming	**optional**
	A set of arguments for the consume. The syntax and semantics of these arguments depends on the providers implementation.		

Rules

Rule: min-consumers

The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.

Rule: priority-based-delivery

Streaming applications SHOULD use different sessions to select different streaming resolutions. AMQP makes no provision for filtering and/or transforming streams except on the basis of priority-based selective delivery of individual messages.

Exceptions

Exception: queue-exists-if-empty

Error:	`not-allowed`

If the queue name in this command is empty, the server MUST raise an exception.

Exception: not-existing-consumer

Error:	`not-allowed`

The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise an exception.

Exception: not-empty-consumer-tag

Error:	`not-allowed`

The client MUST NOT specify a tag that is empty or blank.

Scenario: Attempt to create a consumers with an empty tag.

Exception: in-use

Error:	`resource-locked`

If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise an exception with return code 405 (resource locked).

Command: `stream.consume-ok`

Name	`consume-ok`
Code	`0x4`

An AMQP client MUST handle incoming stream.consume-ok commands (if the stream class is implemented).

This command provides the client with a consumer tag which it may use in commands that work with the consumer.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**
	Holds the consumer tag specified by the client or provided by the server.		

Command: `stream.cancel`

Name	`cancel`
Code	`0x5`

An AMQP server MUST handle incoming stream.cancel commands (if the stream class is implemented).

This command cancels a consumer. Since message delivery is asynchronous the client may continue to receive messages for a short while after cancelling a consumer. It may process or discard these as appropriate.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**

Command: `stream.publish`

Name	`publish`
Code	`0x6`

An AMQP server MUST handle incoming stream.publish commands (if the stream class is implemented).

This command publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers as appropriate.

Arguments

Name	Type	Description	
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange to publish to. The exchange name can be empty, meaning the default exchange. If the exchange name is specified, and that exchange does not exist, the server will raise an exception.		
`routing-key`	`str8`	Message routing key	**optional**
	Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.		
`mandatory`	`bit`	indicate mandatory routing	**optional**
	This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set, the server will return an unroutable message with a Return command. If this flag is zero, the server silently drops the message.		
`immediate`	`bit`	request immediate delivery	**optional**
	This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable message with a Return command. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed.		

Segments

Following the command segment, the following segments may follow.

header

This segment MUST be present.

The header segment consists of at most one of each of the following entries:

- `stream-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Rules

Rule: default

The server MUST accept a blank exchange name to mean the default exchange.

Rule: implementation

The server SHOULD implement the mandatory flag.

Rule: implementation

The server SHOULD implement the immediate flag.

Exceptions

Exception: refusal

Error:	`not-implemented`

The exchange MAY refuse stream content in which case it MUST respond with an exception.

Command: `stream.return`

Name	`return`
Code	`0x7`

An AMQP client MUST handle incoming stream.return commands (if the stream class is implemented).

This command returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

Arguments

Name	Type	Description	
`reply-code`	`return-code`		**optional**
`reply-text`	`str8`	The localized reply text.	**optional**
	The localized reply text. This text can be logged as an aid to resolving issues.		
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange that the message was originally published to.		
`routing-key`	`str8`	Message routing key	**optional**
	Specifies the routing key name specified when the message was published.		

Segments

Following the command segment, the following segments may follow.

header

This segment MUST be present.

The header segment consists of at most one of each of the following entries:

- `stream-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Command: `stream.deliver`

Name	`deliver`
Code	`0x8`

An AMQP client MUST handle incoming stream.deliver commands (if the stream class is implemented).

This command delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume command, then the server responds with Deliver commands as and when messages arrive for that consumer.

Arguments

Name	Type	Description	
`consumer-tag`	`str8`		**optional**
`delivery-tag`	`uint64`		**optional**
	The server-assigned and session-specific delivery tag		
`exchange`	`exchange.name`		**optional**
	Specifies the name of the exchange that the message was originally published to.		
`queue`	`queue.name`		**required**
	Specifies the name of the queue that the message came from. Note that a single session can start many consumers on different queues.		

Segments

Following the command segment, the following segments may follow.

header

This segment MUST be present.

The header segment consists of at most one of each of the following entries:

- `stream-properties` [**optional**].

body

This segment is optional.

The body segment consists of opaque binary data (i.e. the message body).

Rules

Rule: session-local

The delivery tag is valid only within the session from which the message was received. i.e. A client MUST NOT receive a message on one session and then acknowledge it on another.

11. The Model

11.1. Exchanges

11.1.1. Mandatory Exchange Types

11.1.1.1. Direct Exchanges

Rule: exchange_type_direct

An AMQP Server MUST implement the *direct* exchange type.

Rule: exchange_type_direct_binding

If a message *M*, which has routing-key *R*, is sent to an exchange *E* of type *direct*; then *M* shall be delivered to a queue *Q* if and only if there is a binding between *E* and *Q* with binding key *K* such that *K* = *R* (excepting any rule which prevents this delivery).

Rule: default_exchange

An AMQP Server MUST implement an exchange of type *direct* with name equal to the empty string (a str8 value of length zero). Upon creation every queue MUST be bound automatically by the server to this exchange with a binding-key equal to the name of the queue created.

Rule: amq_direct_exchange

An AMQP Server MUST implement an exchange of type *direct* with name *amq.direct*.

11.1.1.2. Fanout Exchanges

Rule: exchange_type_fanout

An AMQP Server MUST implement the *fanout* exchange type.

Rule: exchange_type_fanout_binding

If a message *M*, which has routing-key *R*, is sent to an exchange *E* of type *fanout*; then *M* shall be delivered to a queue *Q* if and only if there is a binding between *E* and *Q* (the binding-key used is unimportant) (excepting any rule which prevents this delivery).

Rule: amq_fanout_exchange

An AMQP Server MUST implement an exchange of type *fanout* with name *amq.fanout*.

11.1.2. Optional Exchange Types

11.1.2.1. Headers Exchanges

Rule: exchange_type_headers

An AMQP Server SHOULD implement the *headers* exchange type.

Rule: headers_exchange_requires_match_arg

When creating a binding between an exchange *E*, of type *headers* and any queue *Q* the *arguments* field MUST contain a key "x-match" to a value of type *str8* which must equal either *"any"* or *"all"*. If the arguments field does not contain a key "x-match" then an exception of type invalid-argument MUST be raised.

Rule: headers_exchange_requires_match_all

Consider a message *M*, which has an application-headers map *P*, which is sent to an exchange *E* of type *headers*. If there exists a binding between *E* a queue *Q* with binding-key *K* and arguments map *A* containg the mapping `{ "x-match" -> (str8,"all") }`, then message *M* MUST route to *Q* because of binding *K* if and only if for every mapping *{key -> (type, value)}* in the binding arguments map *P* which does not have a key beginning with "x-"; there is a matching mapping in the application-headers. In this context "mtaching" means that either the same triplet of key, type, value exist in the application-headers map, or that the mapping in the binding arguments is of the form {key -> void, }, in which case any mapping with same key will match.
.

Rule: headers_exchange_requires_match_any

Consider a message *M*, which has an application-headers map *P*, which is sent to an exchange *E* of type *headers*. If there exists a binding between *E* a queue *Q* with binding-key *K* and arguments map *A* containg the mapping `{ "x-match" -> (str8,"all") }`, then message *M* MUST route to *Q* because of binding *K* if and only if there exists at least one mapping *{key -> (type, value)}* in the binding arguments map *P* which does not have a key beginning with "x-"; for which there is a matching mapping in the application-headers. In this context "mtaching" means that either the same triplet of key, type, value exist in the application-headers map, or that the mapping in the binding arguments is of the form {key -> void, }, in which case any mapping with same key will match.
.

Rule: amq_match_exchange

The server SHOULD implement the headers exchange type and in that case, the server MUST pre-declare within each virtual host at least one exchange of type *headers*, named *amq.match*.

11.1.2.2. Topic Exchanges

Rule: exchange_type_topic

An AMQP Server SHOULD implement the *topic* exchange type.

Rule: exchange_type_topic_binding

If a message *M*, which has routing-key *R*, is sent to an exchange *E* of type *topic*; then *M* shall be delivered to a queue *Q* if and only if there is a binding between *E* and *Q* with binding key *K* such that (excepting any rule which prevents this delivery) *K* matches *R* where matching in this context means the following:

- The *R* is treated as zero or more more words, delimited by the '.' character.
- The binding key MUST be specified in this form and additionally supports special wild-card characters: '*' matches a single word and '#' matches zero or more words.

Thus the routing pattern *.stock.# matches the routing keys usd.stock and eur.stock.db but not stock.nasdaq.

Rule: amq_topic_exchange

The server SHOULD implement the topic exchange type and in that case, the server MUST pre-declare within each virtual host at least one exchange of type *topic*, named *amq.topic*.

11.1.2.3. Failover Exchanges

Rule: exchange_type_failover

An AMQP Server SHOULD implement the *failover* exchange type.

Rule: amq_failover_exchange

If an AMQP Server implements the *failover* exchange, it MUST implement an exchange of type *failover* with name *amq.failover*.

Rule: only_one_failover_exchange

An attempt to declare an exchange of type *failover* except in passive mode should result in an exception of type not-allowed.

Rule: failover_exchange_allow_private_queues

Any AMQP client MAY bind a private queue to this exchange.

Rule: failover_exchange_disallow_shared_queues

It is an error to bind a non-private queue to this exchange. Attempting to bind a queue which is not private to an exchange of type *failover* MUST result in an exception of type not-allowed.

Rule: failover_exchange_behavior

Queues bound to an exchange of type *failover* receive messages with updated information about the set of available failover candidates.

The failover exchange MUST emit "failover update messages" under the following circumstances:

1. When a new queue is bound to the exchange, that queue immediately receives a failover update message.

2. When the set of failover candidates changes, queues bound to the failover exchange receive an update.

Rule: failover_exchange_messages

A failover update message MUST have an empty body. The application-headers field of the message-properties header MUST contain exactly one entry with name "amq.failover" and a value of type array (but of domain amqp-url-array) containing a list of AMQP URLs.

The URL provides a list of broker addresses that the client MAY fail over to in the event of a crash or disconnect.

Rule: failover_exchange_message_url_ordering

The broker SHOULD order the addresses in such a way that resources will be efficiently allocated if clients consistently connect to the first address on the list. Clients SHOULD try addresses in the order listed. However a client MAY choose any address on the list. The list MAY have a single entry or be empty depending on the configuration of the cluster. The client MAY attempt to reconnect to the original broker as well as the brokers listed in failover updates.

Rule: failover_exchange_message_differences

The failover exchange MAY give different failover lists to each connected queue - different order or entirely different addresses. For example each client might receive a different random ordering of available candidates to balance load. Another example, the broker might give different failvover lists to provide different quality of service guarantees to different clients, or might replicate information about each clients to a different set of backup brokers.

Rule: failover_exchange_spontaneous_disconnect

A healthy broker MAY abruptly disconnect clients without the normal connection closure protocol in order to force a fail over for load balancing purposes. It SHOULD wait till all queues bound to the failover exchange are empty (i.e. all clients have received the latest failover update message) but it MAY disconnect clients that are slow to respond after a broker determined timeout.

Rule: failover_exchange_usage

Brokers that do not support fail-over are not required to provide the "amq.failover" exchange. Clients that do not support fail-over are not required to use it.

11.1.3. System Exchanges

Rule: exchange_type_system

An AMQP Server MAY implement the *system* exchange type.

Rule: system_exchange_behavior

The system exchange type works as follows:

1. A publisher sends the exchange a message with the routing key S.

2. The system exchange passes this to a system service S.

System services starting with "amq." are reserved for AMQP usage. All other names may be used freely on by server implementations.

Rule: system_exchange_binding_forbidden

An attempt to bind any queue to an exchange of type *system* MUST result in an exception of type not-allowed.

11.1.4. Implementation-defined Exchange Types

Rule: exchange_type_naming

All non-normative exchange types MUST be named starting with "x-". Exchange types that do not start with "x-" are reserved for future use in the AMQP standard.

11.1.5. Exchange Naming

Rule: exchange_naming

Exchange names beginning "amq." are reserved for AMQP standard exchanges. An attempt to declare an exchange with name beginning "amq." except in passive mode MUST result in an exception of type not-allowed.

11.2. Queues

Rule: queue_naming

Queue names beginning "amq." are reserved for AMQP standard queues. An attempt to declare a queue with name beginning "amq." except in passive mode MUST result in an exception of type not-allowed.

12. Protocol Grammar

12.1. Augmented BNF Rules

We use the Augmented BNF syntax defined in IETF RFC 2234. In summary,

1. The name of a rule is simply the name itself.
2. Terminals are specified by one or more numeric characters with the base interpretation of those characters indicated as 'b', 'd' or 'x'.
3. A rule can define a simple, ordered string of values by listing a sequence of rule names.
4. A range of alternative numeric values can be specified compactly, using dash ("-") to indicate the range of alternative values.
5. Elements enclosed in parentheses are treated as a single element, whose contents are strictly ordered.
6. Elements separated by forward slash ("/") are alternatives.
7. The operator "*" preceding an element indicates repetition. The full form is: "<a>*<b>element", where <a> and <b> are optional decimal values, indicating at least <a> and at most <b> occurrences of element.
8. A rule of the form: "<n>element" is equivalent to <n>*<n>element.
9. Square brackets enclose an optional element sequence.

12.2. Grammar

We provide a complete grammar for AMQP:

Framing

```
             amqp = protocol-header *frame

  protocol-header = AMQP class instance major minor
             AMQP = "AMQP"
   protocol-class = %x01
protocol-instance = %x01                      ; AMQP over TCP
                  / %x02                      ; AMQP over SCTP
            major = OCTET                     ; major version
            minor = OCTET                     ; minor version

            frame = frame-header frame-body

     frame-header = flags type size %x00 %b0.0.0.0 track channel %x00.00.00.00

            flags = frame-version %b0.0 first-segment last-segment first-frame last-frame
    frame-version = 2 BIT
    first-segment = 1 BIT
     last-segment = 1 BIT
      first-frame = 1 BIT
       last-frame = 1 BIT

             type = 1 OCTET
             size = uint16
            track = 4 BIT
          channel = 2 OCTET

       frame-body = *OCTET
```

Assemblies

```
        assembly = control-segment
                 / command-segment [header-segment] [body-segment]

  control-segment = class-code control-code arguments
  command-segment = class-code command-code session-header arguments
   header-segment = *struct32 ; further restricted by amqp.xml
     body-segment = *OCTET

       class-code = OCTET
     control-code = OCTET
     command-code = OCTET

   session-header = ssn-hdr-flags ssn-hdr-fields
    ssn-hdr-flags = 2 OCTET   ; packing flags for fields
   ssn-hdr-fields = *OCTET    ; defined by session.header struct in amqp.xml

        arguments = *OCTET    ; defined by amqp.xml
```

Types

```
            bin8 = OCTET
```

```
            int8 = OCTET
```

```
           uint8 = OCTET
```

```
            char = OCTET
```

```
         boolean = OCTET
```

```
           bin16 = 2 OCTET
```

```
           int16 = high-byte low-byte
       high-byte = OCTET
        low-byte = OCTET
```

```
          uint16 = high-byte low-byte
       high-byte = OCTET
        low-byte = OCTET
```

```
bin32 = 4 OCTET
```

```
int32 = byte-four byte-three byte-two byte-one
byte-four = OCTET ; most significant byte (MSB)
byte-three = OCTET
byte-two = OCTET
byte-one = OCTET ; least significant byte (LSB)
```

```
uint32 = byte-four byte-three byte-two byte-one
byte-four = OCTET ; most significant byte (MSB)
byte-three = OCTET
byte-two = OCTET
byte-one = OCTET ; least significant byte (LSB)
```

```
float = 4 OCTET ; IEEE 754 32-bit floating point number
```

```
char-utf32 = 4 OCTET ; single UTF-32 character
```

```
sequence-no = 4 OCTET ; RFC-1982 serial number
```

```
bin64 = 8 OCTET
```

```
int64 = byte-eight byte-seven byte-six byte-five
        byte-four byte-three byte-two byte-one
byte-eight = 1 OCTET ; most significant byte (MSB)
byte-seven = 1 OCTET
byte-six = 1 OCTET
byte-five = 1 OCTET
byte-four = 1 OCTET
byte-three = 1 OCTET
byte-two = 1 OCTET
byte-one = 1 OCTET ; least significant byte (LSB)
```

```
uint64 = byte-eight byte-seven byte-six byte-five
         byte-four byte-three byte-two byte-one
byte-eight = 1 OCTET ; most significant byte (MSB)
byte-seven = 1 OCTET
byte-six = 1 OCTET
byte-five = 1 OCTET
byte-four = 1 OCTET
byte-three = 1 OCTET
byte-two = 1 OCTET
byte-one = 1 OCTET ; least significant byte (LSB)
```

```
double = 8 OCTET ; double precision IEEE 754 floating point number
```

```
datetime = 8 OCTET ; 64 bit posix time_t format
```

```
bin128 = 16 OCTET
```

```
uuid = 16 OCTET ; RFC-4122 section 4.1.2
```

```
bin256 = 32 OCTET
```

```
bin512 = 64 OCTET
```

```
bin1024 = 128 OCTET
```

```
vbin8 = size octets
 size = uint8
octets = 0*255 OCTET ; size OCTETs
```

```
str8-latin = size characters
      size = uint8
characters = 0*255 OCTET ; size OCTETs
```

```
      str8 = size utf8-unicode
      size = uint8
utf8-unicode = 0*255 OCTET ; size OCTETs
```

```
str8-utf16 = size utf16-unicode
      size = uint8
utf16-unicode = 0*255 OCTET ; size OCTETs
```

```
vbin16 = size octets
  size = uint16
octets = 0*65535 OCTET ; size OCTETs
```

```
str16-latin = size characters
       size = uint16
 characters = 0*65535 OCTET ; size OCTETs
```

```
        str16 = size utf8-unicode
         size = uint16
 utf8-unicode = 0*65535 OCTET ; size OCTETs
```

```
  str16-utf16 = size utf16-unicode
         size = uint16
utf16-unicode = 0*65535 OCTET ; size OCTETs
```

```
  byte-ranges = size *range
         size = uint16
        range = lower upper
        lower = uint64
        upper = uint64
```

```
  sequence-set = size *range
         size = uint16         ; length of variable portion in bytes

        range = lower upper    ; inclusive
        lower = sequence-no
        upper = sequence-no
```

```
      vbin32 = size octets
        size = uint32
      octets = 0*4294967295 OCTET ; size OCTETs
```

```
         map = size count *entry

        size = uint32             ; size of count and entries in octets
       count = uint32             ; number of entries in the map

       entry = key type value
         key = str8
        type = OCTET              ; type code of the value
       value = *OCTET             ; the encoded value
```

```
        list = size count *item

        size = uint32            ; size of count and items in octets
       count = uint32            ; number of items in the list

        item = type value
        type = OCTET             ; type code of the value
       value = *OCTET            ; the encoded value
```

```
       array = size type count values

        size = uint32            ; size of type, count, and values in octets
        type = OCTET             ; the type of the encoded values
       count = uint32            ; number of items in the array

      values = 0*4294967290 OCTET ; (size - 5) OCTETs
```

```
         struct32 = size class-code struct-code packing-flags field-data

             size = uint32

       class-code = OCTET     ; zero for top-level structs
      struct-code = OCTET     ; together with class-code identifies the struct
                              ; definition which determines the pack-width and
                              ; fields

    packing-flags = 0*4 OCTET ; pack-width OCTETs

       field-data = *OCTET    ; (size - 2 - pack-width) OCTETs
```

```
            bin40 = 5 OCTET
```

```
            dec32 = exponent mantissa
         exponent = uint8
         mantissa = int32
```

```
            bin64 = 9 OCTET
```

```
            dec64 = exponent mantissa
         exponent = uint8
         mantissa = int64
```

Appendix A. Conformance Tests

A.1. Introduction

The AMQP conformance tests are designed to verify how far an AMQP server actually conforms to the specifications laid out in this document. In principle, every "guideline for implementers", or "RULE" in the protocol's XML specification has a specific test that verifies whether the server conforms or not. In practice, some of the guidelines are intended for clients, and some are not testable without excessive cost.

The protocol itself cross references test by a logical label from within the protocol XML description, but the Test Sets will be documented elsewhere as developed and ratified by the AMQ Protocol governing body.

Note that tests do not test performance, stability, or scalability. The scope of the conformance tests is to measure how far an AMQP server is compatible with the protocol specifications, not how well it is built.

Appendix B. Implementation Guide

It is the intent of the authors to include a full implementation guide in a future release of the specification. The material included here will form the starting point for the implementation guide.

B.1. AMQP Client Architecture

It is possible to read and write AMQP frames directly from an application but this would be bad design. Even the simplest AMQP dialogue is rather more complex than, say HTTP, and application developers should not need to understand such things as binary framing formats in order to send a message to a message queue.

The recommended AMQP client architecture consists of several layers of abstraction:

1. A **framing layer**. This layer takes AMQP protocol commands or controls, in some language-specific format (structures, classes, etc.) and serializes them as wire-level frames. The framing layer can be mechanically generated from the AMQP specification (which is defined in a protocol modelling language, implemented in XML and specifically designed for AMQP).

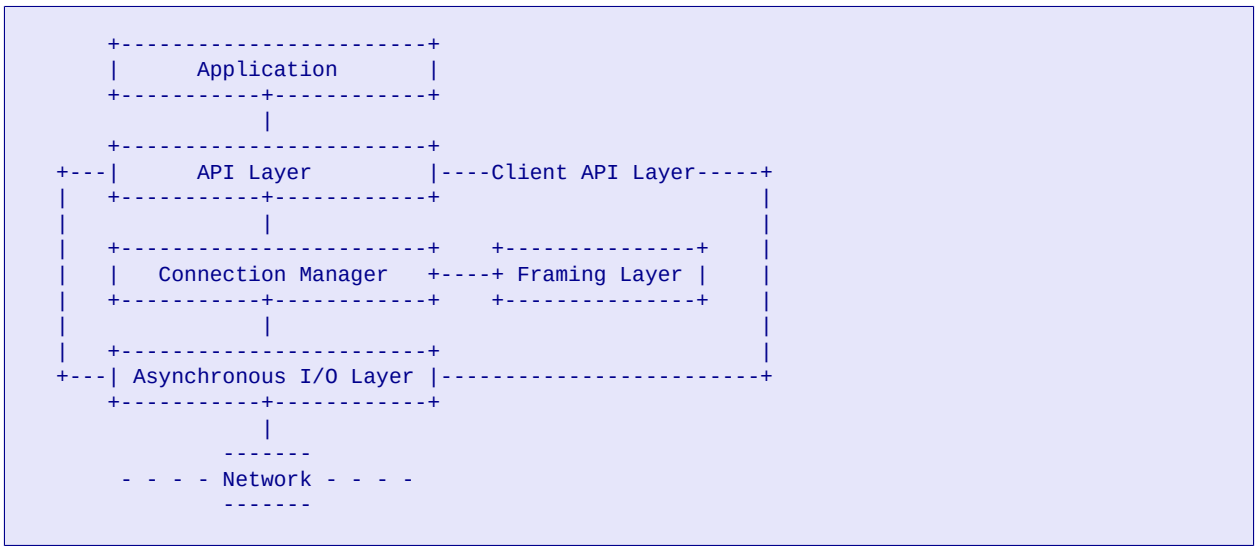
2. A **connection manager layer**. This layer reads and writes AMQP frames and manages the overall connection and session logic. In this layer we can encapsulate the full logic of opening a connection and session, error handling, content transmission and reception, and so on. Large parts of this layer can be produced automatically from the AMQP specifications. For instance, the specifications define which commands carry content, so the logic "send command and then optionally send content" can be produced mechanically.

3. An **API layer**. This layer exposes a specific API for applications to work with. The API layer may reflect some existing standard, or may expose the high-level AMQP commands, making a mapping as described earlier in this section. The AMQP commands are designed to make this mapping both simple and useful. The API layer may itself be composed of several layers, e.g. a higher-level API constructed on top of the AMQP command API.

4. A **transaction processing layer**. This layer drives the application by delivering it transactions to process, where the transactions are middleware messages. Using a transaction layer can be very powerful because the middleware becomes entirely hidden, making applications easier to build, test, and maintain.

Additionally, there is usually some kind of I/O layer, which can be very simple (synchronous socket reads and writes) or sophisticated (fully asynchronous multi-threaded i/o).

This diagram shows the overall recommended architecture (without layer 4, which is a different story):



In this document, when we speak of the "client API", we mean all the layers below the application (i/o, framing, connection manager, and API layers. We will usually speak of "the client API" and "the application" as two separate things, where the application uses the client API to talk to the middleware server.