Table of Contents

1. AMQP version 0-9-1 specification

This document was automatically generated from the AMQP XML specification. All edits to the content of this file should be directed to the XML file (for content) or the XSLT template (for layout and/or formatting).

1.1. AMQP-defined Domains

The following domains are defined in this specification:

Name	Type	[Label] Description
bit	bit	[single bit]
class-id	short	
consumer-tag	shortstr	[consumer tag] Identifier for the consumer, valid within the current channel.
delivery-tag	longlong	[server-assigned delivery tag] The server-assigned and channel-specific delivery tag
exchange-name	shortstr	[exchange name] The exchange name is a client-selected string that identifies the exchange for publish methods.
long	long	[32-bit integer]
longlong	longlong	[64-bit integer]
longstr	longstr	[long string]
message-count	long	[number of messages in queue] The number of messages in the queue, which will be zero for newly-declared queues. This is the number of messages present in the queue, and committed if the channel on which they were published is transacted, that are not waiting acknowledgement.
method-id	short	
no-ack	bit	[no acknowledgement needed] 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 assumes the delivery will succeed and immediately dequeues it. This functionality may increase performance but at the cost of reliability. Messages can get lost if a client dies before they are delivered to the application.
no-local	bit	[do not deliver own messages] If the no-local field is set the server will not send messages to the connection that published them.
no-wait	bit	[do not send reply method] If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.
octet	octet	[single octet]
path	shortstr	Unconstrained.
peer-properties	table	This table provides a set of peer properties, used for identification, debugging, and general information.
queue-name	shortstr	[queue name] The queue name identifies the queue within the vhost. In methods where the queue name may be blank, and that has no specific significance, this refers to the 'current' queue for the channel, meaning the last queue that the client declared on the channel. If the client did not declare a queue, and the method needs a queue name, this will result in a 502 (syntax error) channel exception.
redelivered	bit	[message is being redelivered] This indicates that the message has been

Name	Type	[Label] Description
		previously delivered to this or another client.
reply-code	short	[reply code from server] The reply code. The AMQ reply codes are defined as constants at the start of this formal specification.
reply-text	shortstr	[localised reply text] The localised reply text. This text can be logged as an aid to resolving issues.
short	short	[16-bit integer]
shortstr	shortstr	[short string]
table	table	[field table]
timestamp	timestamp	[64-bit timestamp]

1.2. AMQP-defined Constants

Many constants are error codes. Where this is so, they will fall into one of two categories:

- **Channel Errors:** These codes are all associated with failures that affect the current channel but not other channels in the same connection;
- **Connection Errors:** These codes are all associated with failures that preclude any further activity on the connection and require its closing.

The following constants are defined in the specification:

Name	Value	Error type	[Label] Description
frame-method	1		
frame-header	2		
frame-body	3		
frame-heartbeat	8		
frame-min-size	4096		
frame-end	206		
reply-success	200		Indicates that the method completed successfully. This reply code is reserved for future use - the current protocol design does not use positive confirmation and reply codes are sent only in case of an error.
content-too-large	311	channel	The client attempted to transfer content larger than the server could accept at the present time. The client may retry at a later time.
no-consumers	313	channel	When 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.
connection-forced	320	connection	An operator intervened to close the connection for some reason. The client may retry at some later date.
invalid-path	402	connection	The client tried to work with an unknown virtual host.
access-refused	403	channel	The client attempted to work with a server entity to which it has no access due to security settings.
not-found	404	channel	The client attempted to work with a server entity that does not exist.
resource-locked	405	channel	The client attempted to work with a server entity to which it has no access

Name	Value	Error type	[Label] Description
			because another client is working with it.
precondition-failed	406	channel	The client requested a method that was not allowed because some precondition failed.
frame-error	501	connection	The sender sent a malformed frame that the recipient could not decode. This strongly implies a programming error in the sending peer.
syntax-error	502	connection	The sender sent a frame that contained illegal values for one or more fields. This strongly implies a programming error in the sending peer.
command-invalid	503	connection	The client sent an invalid sequence of frames, attempting to perform an operation that was considered invalid by the server. This usually implies a programming error in the client.
channel-error	504	connection	The client attempted to work with a channel that had not been correctly opened. This most likely indicates a fault in the client layer.
unexpected-frame	505	connection	The peer sent a frame that was not expected, usually in the context of a content header and body. This strongly indicates a fault in the peer's content processing.
resource-error	506	connection	The server could not complete the method because it lacked sufficient resources. This may be due to the client creating too many of some type of entity.
not-allowed	530	connection	The client tried to work with some entity in a manner that is prohibited by the server, due to security settings or by some other criteria.
not-implemented	540	connection	The client tried to use functionality that is not implemented in the server.
internal-error	541	connection	The server could not complete the method because of an internal error. The server may require intervention by an operator in order to resume normal operations.

1.3. Class and Method ID Summaries

The following class and method IDs are defined in the specification:

Class	ID	Short class description	Method	ID	Short method description
connection	10	work with socket connections	start	10	start connection negotiation
			start-ok	11	select security mechanism and locale
			secure	20	security mechanism challenge
			secure-ok	21	security mechanism response
			tune	30	propose connection tuning parameters
			tune-ok	31	negotiate connection tuning parameters
			open	40	open connection to virtual host
			open-ok	41	signal that connection is ready
			close	50	request a connection close
			close-ok	51	confirm a connection close
channel	20	work with channels	open	10	open a channel for use
			open-ok	11	signal that the channel is ready

Class	ID	Short class description	Method	ID	Short method description
			flow	20	enable/disable flow from peer
			flow-ok	21	confirm a flow method
			close	40	request a channel close
			close-ok	41	confirm a channel close
exchange	40	work with exchanges	declare	10	verify exchange exists, create if needed
			declare-ok	11	confirm exchange declaration
			delete	20	delete an exchange
			delete-ok	21	confirm deletion of an exchange
queue	50	work with queues	declare	10	declare queue, create if needed
			declare-ok	11	confirms a queue definition
			bind	20	bind queue to an exchange
			bind-ok	21	confirm bind successful
			unbind	50	unbind a queue from an exchange
			unbind-ok	51	confirm unbind successful
			purge	30	purge a queue
			purge-ok	31	confirms a queue purge
			delete	40	delete a queue
			delete-ok	41	confirm deletion of a queue
basic	60	work with basic content	qos	10	specify quality of service
			qos-ok	11	confirm the requested qos
			consume	20	start a queue consumer
			consume-ok	21	confirm a new consumer
			cancel	30	end a queue consumer
			cancel-ok	31	confirm a cancelled consumer
			publish	40	publish a message
			return	50	return a failed message
			deliver	60	notify the client of a consumer message
			get	70	direct access to a queue
			get-ok	71	provide client with a message
			get-empty	72	indicate no messages available
			ack	80	acknowledge one or more messages
			reject	90	reject an incoming message
			recover-async	100	redeliver unacknowledged messages
			recover	110	redeliver unacknowledged messages
			recover-ok	111	confirm recovery
tx	90	work with transactions	select	10	select standard transaction mode
			select-ok	11	confirm transaction mode
			commit	20	commit the current transaction

Class	ID	Short class description	Method	ID	Short method description
			commit-ok	21	confirm a successful commit
			rollback	30	abandon the current transaction
			rollback-ok	31	confirm successful rollback

1.4. Class connection

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

Class 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
challenge            = S:SECURE C:SECURE-OK
use-connection       = *channel
close-connection     = C:CLOSE S:CLOSE-OK
                     / S:CLOSE C:CLOSE-OK
```

1.4.1. Property and Method Summary

Class ***connection*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
start	10	start-ok	start connection negotiation		Y	version-major	octet	protocol major version
						version-minor	octet	protocol minor version
						server-properties	peer-properties	server properties
						mechanisms	longstr	available security mechanisms
						locales	longstr	available message locales
start-ok	11		select security mechanism and locale	Y		client-properties	peer-properties	client properties
						mechanism	shortstr	selected security mechanism
						response	longstr	security response data
						locale	shortstr	selected message locale
secure	20	secure-ok	security mechanism challenge		Y	challenge	longstr	security challenge data
secure-ok	21		security mechanism	Y		response	longstr	security response data

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
			response					
tune	30	tune-ok	propose connection tuning parameters		Y	channel-max	short	proposed maximum channels
						frame-max	long	proposed maximum frame size
						heartbeat	short	desired heartbeat delay
tune-ok	31		negotiate connection tuning parameters	Y		channel-max	short	negotiated maximum channels
						frame-max	long	negotiated maximum frame size
						heartbeat	short	desired heartbeat delay
open	40	open-ok	open connection to virtual host	Y		virtual-host	path	virtual host name
						reserved-1		
						reserved-2		
open-ok	41		signal that connection is ready		Y	reserved-1		
close	50	close-ok	request a connection close	Y	Y	reply-code	reply-code	
						reply-text	reply-text	
						class-id	class-id	failing method class
						method-id	method-id	failing method ID
close-ok	51		confirm a connection close	Y	Y	[No parameters defined for this method]		

1.4.2. Methods

1.4.2.1. Method connection.start (ID 10)

ID: 10

Method accepted by: Client

Synchronous: Yes; expected response is from method(s) ***connection.start-ok***

Number of parameters: 5

Label: start connection negotiation

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	version-major	octet	protocol major version
2	version-minor	octet	protocol minor version
3	server-properties	peer-properties	server properties

4	mechanisms	longstr	available security mechanisms
5	locales	longstr	available message locales

This method starts the connection negotiation process by telling the client the protocol version that the server proposes, along with a list of security mechanisms which the client can use for authentication.

Guidelines for implementers:

- If the server cannot support the protocol specified in the protocol header, it MUST respond with a valid protocol header and then close the socket connection.

  Test scenario: The client sends a protocol header containing an invalid protocol name. The server MUST respond by sending a valid protocol header and then closing the connection.

- The server MUST provide a protocol version that is lower than or equal to that requested by the client in the protocol header.

  Test scenario: The client requests a protocol version that is higher than any valid implementation, e.g. 2.0. The server must respond with a protocol header indicating its supported protocol version, e.g. 1.0.

- If the client cannot handle the protocol version suggested by the server it MUST close the socket connection without sending any further data.

  Test 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 without sending any further data.

1.4.2.1.1. Parameter connection.start.version-major (octet)

Ordinal: 1

Domain: octet

Label: protocol major version

The major version number can take any value from 0 to 99 as defined in the AMQP specification.

1.4.2.1.2. Parameter connection.start.version-minor (octet)

Ordinal: 2

Domain: octet

Label: protocol minor version

The minor version number can take any value from 0 to 99 as defined in the AMQP specification.

1.4.2.1.3. Parameter connection.start.server-properties (peer-properties)

Ordinal: 3

Domain: peer-properties

Label: server properties

1.4.2.1.4. Parameter connection.start.mechanisms (longstr)

Ordinal: 4

Domain: longstr

Label: available security mechanisms

A list of the security mechanisms that the server supports, delimited by spaces.

1.4.2.1.5. Parameter connection.start.locales (longstr)

Ordinal: 5

Domain: longstr

Label: available message locales

A list of the message locales that the server supports, delimited by spaces. The locale defines the language in which the server will send reply texts.

1.4.2.2. Method connection.start-ok (ID 11)

ID: 11

Method accepted by: Server

Synchronous: No

Number of parameters: 4

Label: select security mechanism and locale

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	client-properties	peer-properties	client properties
2	mechanism	shortstr	selected security mechanism
3	response	longstr	security response data
4	locale	shortstr	selected message locale

This method selects a SASL security mechanism.

1.4.2.2.1. Parameter connection.start-ok.client-properties (peer-properties)

Ordinal: 1

Domain: peer-properties

Label: client properties

1.4.2.2.2. Parameter connection.start-ok.mechanism (shortstr)

Ordinal: 2

Domain: shortstr

Label: selected security mechanism

A single security mechanisms selected by the client, which must be one of those specified by the server.

1.4.2.2.3. Parameter connection.start-ok.response (longstr)

Ordinal: 3

Domain: longstr

Label: security response data

A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

1.4.2.2.4. Parameter connection.start-ok.locale (shortstr)

Ordinal: 4

Domain: shortstr

Label: selected message locale

A single message locale selected by the client, which must be one of those specified by the server.

1.4.2.3. Method connection.secure (ID 20)

ID: 20

Method accepted by: Client

Synchronous: Yes; expected response is from method(s) ***connection.secure-ok***

Number of parameters: 1

Label: security mechanism challenge

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	challenge	longstr	security challenge data

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

1.4.2.3.1. Parameter connection.secure.challenge (longstr)

Ordinal: 1

Domain: longstr

Label: security challenge data

Challenge information, a block of opaque binary data passed to the security mechanism.

1.4.2.4. Method connection.secure-ok (ID 21)

ID: 21

Method accepted by: Server

Synchronous: No

Number of parameters: 1

Label: security mechanism response

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	response	longstr	security response data

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

1.4.2.4.1. Parameter connection.secure-ok.response (longstr)

Ordinal: 1

Domain: longstr

Label: security response data

A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

1.4.2.5. Method connection.tune (ID 30)

ID: 30

Method accepted by: Client

Synchronous: Yes; expected response is from method(s) ***connection.tune-ok***

Number of parameters: 3

Label: propose connection tuning parameters

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	channel-max	short	proposed maximum channels
2	frame-max	long	proposed maximum frame size
3	heartbeat	short	desired heartbeat delay

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

1.4.2.5.1. Parameter connection.tune.channel-max (short)

Ordinal: 1

Domain: short

Label: proposed maximum channels

Specifies highest channel number that the server permits. Usable channel numbers are in the range 1..channel-max. Zero indicates no specified limit.

1.4.2.5.2. Parameter connection.tune.frame-max (long)

Ordinal: 2

Domain: long

Label: proposed maximum frame size

The largest frame size that the server proposes for the connection, including frame header and end-byte. The client can negotiate a lower value. Zero means that the server does not impose any specific limit but may reject very large frames if it cannot allocate resources for them.

1.4.2.5.3. Parameter connection.tune.heartbeat (short)

Ordinal: 3

Domain: short

Label: desired heartbeat delay

The delay, in seconds, of the connection heartbeat that the server wants. Zero means the server does not want a heartbeat.

1.4.2.6. Method connection.tune-ok (ID 31)

ID: 31

Method accepted by: Server

Synchronous: No

Number of parameters: 3

Label: negotiate connection tuning parameters

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	channel-max	short	negotiated maximum channels
2	frame-max	long	negotiated maximum frame size
3	heartbeat	short	desired heartbeat delay

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

1.4.2.6.1. Parameter connection.tune-ok.channel-max (short)

Ordinal: 1

Domain: short

Label: negotiated maximum channels

The maximum total number of channels that the client will use per connection.

1.4.2.6.2. Parameter connection.tune-ok.frame-max (long)

Ordinal: 2

Domain: long

Label: negotiated maximum frame size

The largest frame size that the client and server will use for the connection. Zero 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 frame-max limit applies principally to content frames, where large contents can be broken into frames of arbitrary size.

1.4.2.6.3. Parameter connection.tune-ok.heartbeat (short)

Ordinal: 3

Domain: short

Label: desired heartbeat delay

The delay, in seconds, of the connection heartbeat that the client wants. Zero means the client does not want a heartbeat.

1.4.2.7. Method connection.open (ID 40)

ID: 40

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***connection.open-ok***

Number of parameters: 3

Label: open connection to virtual host

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	virtual-host	path	virtual host name
2	reserved-1		
3	reserved-2		

This method 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.

1.4.2.7.1. Parameter connection.open.virtual-host (path)

Ordinal: 1

Domain: path

Label: virtual host name

The name of the virtual host to work with.

1.4.2.7.2. Parameter connection.open.reserved-1 ()

Ordinal: 2

Domain:

1.4.2.7.3. Parameter connection.open.reserved-2 ()

Ordinal: 3

Domain:

1.4.2.8. Method connection.open-ok (ID 41)

ID: 41

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: signal that connection is ready

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		

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

1.4.2.8.1. Parameter connection.open-ok.reserved-1 ()

Ordinal: 1

Domain:

1.4.2.9. Method connection.close (ID 50)

ID: 50

Method accepted by: Server, Client

Synchronous: Yes; expected response is from method(s) ***connection.close-ok***

Number of parameters: 4

Label: request a connection close

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reply-code	reply-code	
2	reply-text	reply-text	
3	class-id	class-id	failing method class
4	method-id	method-id	failing method ID

This method indicates that the sender wants to close the connection. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

Guidelines for implementers:

- After sending this method, any received methods except Close and Close-OK MUST be discarded. The response to receiving a Close after sending Close must be to send Close-Ok.

1.4.2.9.1. Parameter connection.close.reply-code (reply-code)

Ordinal: 1

Domain: reply-code

1.4.2.9.2. Parameter connection.close.reply-text (reply-text)

Ordinal: 2

Domain: reply-text

1.4.2.9.3. Parameter connection.close.class-id (class-id)

Ordinal: 3

Domain: class-id

Label: failing method class

When the close is provoked by a method exception, this is the class of the method.

1.4.2.9.4. Parameter connection.close.method-id (method-id)

Ordinal: 4

Domain: method-id

Label: failing method ID

When the close is provoked by a method exception, this is the ID of the method.

1.4.2.10. Method connection.close-ok (ID 51)

ID: 51

Method accepted by: Server, Client

Synchronous: No

Number of parameters: 0

Label: confirm a connection close

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

Guidelines for implementers:

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

1.5. Class channel

The channel class provides methods for a client to establish a channel to a server and for both peers to operate the channel thereafter.

Class Grammar:

```
channel              = open-channel *use-channel close-channel
open-channel         = C:OPEN S:OPEN-OK
use-channel          = C:FLOW S:FLOW-OK
                     / S:FLOW C:FLOW-OK
                     / functional-class
close-channel        = C:CLOSE S:CLOSE-OK
                     / S:CLOSE C:CLOSE-OK
```

1.5.1. Property and Method Summary

Class ***channel*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
open	10	open-ok	open a channel for use	Y		reserved-1		
open-ok	11		signal that the channel is ready		Y	reserved-1		
flow	20	flow-ok	enable/disable flow from peer	Y	Y	active	bit	start/stop content frames
flow-ok	21		confirm a flow method	Y	Y	active	bit	current flow setting
close	40	close-ok	request a channel close	Y	Y	reply-code	reply-code	
						reply-text	reply-text	

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
						class-id	class-id	failing method class
						method-id	method-id	failing method ID
close-ok	41		confirm a channel close	Y	Y	[No parameters defined for this method]		

1.5.2. Methods

1.5.2.1. Method channel.open (ID 10)

ID: 10

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***channel.open-ok***

Number of parameters: 1

Label: open a channel for use

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		

This method opens a channel to the server.

Guidelines for implementers:

- The client MUST NOT use this method on an already-opened channel.

  Test scenario: Client opens a channel and then reopens the same channel.

  On failure: Constant "channel-error" (See AMQP-defined Constants)

1.5.2.1.1. Parameter channel.open.reserved-1 ()

Ordinal: 1

Domain:

1.5.2.2. Method channel.open-ok (ID 11)

ID: 11

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: signal that the channel is ready

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		

This method signals to the client that the channel is ready for use.

1.5.2.2.1. Parameter channel.open-ok.reserved-1 ()

Ordinal: 1

Domain:

1.5.2.3. Method channel.flow (ID 20)

ID: 20

Method accepted by: Server, Client

Synchronous: Yes; expected response is from method(s) ***channel.flow-ok***

Number of parameters: 1

Label: enable/disable flow from peer

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	active	bit	start/stop content frames

This method asks the peer to pause or restart the flow of content data sent by a consumer. This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control. It does not affect contents returned by Basic.Get-Ok methods.

Guidelines for implementers:

- When a new channel is opened, it is active (flow is active). Some applications assume that channels are inactive until started. To emulate this behaviour a client MAY open the channel, then pause it.
- When sending content frames, a peer SHOULD monitor the channel for incoming methods and respond to a Channel.Flow as rapidly as possible.
- A peer MAY use the Channel.Flow method to throttle incoming content data for internal reasons, for example, when exchanging data over a slower connection.
- The peer that requests a Channel.Flow method MAY disconnect and/or ban a peer that does not respect the request. This is to prevent badly-behaved clients from overwhelming a server.

1.5.2.3.1. Parameter channel.flow.active (bit)

Ordinal: 1

Domain: bit

Label: start/stop content frames

If 1, the peer starts sending content frames. If 0, the peer stops sending content frames.

1.5.2.4. Method channel.flow-ok (ID 21)

ID: 21

Method accepted by: Server, Client

Synchronous: No

Number of parameters: 1

Label: confirm a flow method

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	active	bit	current flow setting

Confirms to the peer that a flow command was received and processed.

1.5.2.4.1. Parameter channel.flow-ok.active (bit)

Ordinal: 1

Domain: bit

Label: current flow setting

Confirms the setting of the processed flow method: 1 means the peer will start sending or continue to send content frames; 0 means it will not.

1.5.2.5. Method channel.close (ID 40)

ID: 40

Method accepted by: Server, Client

Synchronous: Yes; expected response is from method(s) ***channel.close-ok***

Number of parameters: 4

Label: request a channel close

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reply-code	reply-code	
2	reply-text	reply-text	
3	class-id	class-id	failing method class
4	method-id	method-id	failing method ID

This method indicates that the sender wants to close the channel. This may be due to internal conditions

(e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

Guidelines for implementers:

- After sending this method, any received methods except Close and Close-OK MUST be discarded. The response to receiving a Close after sending Close must be to send Close-Ok.

1.5.2.5.1. Parameter channel.close.reply-code (reply-code)

Ordinal: 1

Domain: reply-code

1.5.2.5.2. Parameter channel.close.reply-text (reply-text)

Ordinal: 2

Domain: reply-text

1.5.2.5.3. Parameter channel.close.class-id (class-id)

Ordinal: 3

Domain: class-id

Label: failing method class

When the close is provoked by a method exception, this is the class of the method.

1.5.2.5.4. Parameter channel.close.method-id (method-id)

Ordinal: 4

Domain: method-id

Label: failing method ID

When the close is provoked by a method exception, this is the ID of the method.

1.5.2.6. Method channel.close-ok (ID 41)

ID: 41

Method accepted by: Server, Client

Synchronous: No

Number of parameters: 0

Label: confirm a channel close

This method confirms a Channel.Close method and tells the recipient that it is safe to release resources for the channel.

Guidelines for implementers:

- A peer that detects a socket closure without having received a Channel.Close-Ok handshake

method SHOULD log the error.

1.6. Class exchange

Exchanges match and distribute messages across queues. Exchanges can be configured in the server or declared at runtime.

Class Grammar:

```
exchange              = C:DECLARE    S:DECLARE-OK
                      / C:DELETE     S:DELETE-OK
```

Guidelines for implementers:

- The server MUST implement these standard exchange types: fanout, direct.

  Test scenario: Client attempts to declare an exchange with each of these standard types.

- The server SHOULD implement these standard exchange types: topic, headers.

  Test scenario: Client attempts to declare an exchange with each of these standard types.

- 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 methods.

  Test scenario: Client declares 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).

- The server MUST pre-declare a direct exchange with no public name to act as the default exchange for content Publish methods and for default queue bindings.

  Test scenario: Client checks that the default exchange is active by specifying a queue binding with no exchange name, and publishing a message with a suitable routing key but without specifying the exchange name, then ensuring that the message arrives in the queue correctly.

- The server MUST NOT allow clients to access the default exchange except by specifying an empty exchange name in the Queue.Bind and content Publish methods.

- The server MAY implement other exchange types as wanted.

1.6.1. Property and Method Summary

Class ***exchange*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
declare	10	declare-ok	verify exchange exists, create if needed	Y		reserved-1		
						exchange	exchange-name	

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
						type	shortstr	exchange type
						passive	bit	do not create exchange
						durable	bit	request a durable exchange
						reserved-2		
						reserved-3		
						no-wait	no-wait	
						arguments	table	arguments for declaration
declare-ok	11		confirm exchange declaration		Y	[No parameters defined for this method]		
delete	20	delete-ok	delete an exchange	Y		reserved-1		
						exchange	exchange-name	
						if-unused	bit	delete only if unused
						no-wait	no-wait	
delete-ok	21		confirm deletion of an exchange		Y	[No parameters defined for this method]		

1.6.2. Methods

1.6.2.1. Method exchange.declare (ID 10)

ID: 10

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***exchange.declare-ok***

Number of parameters: 9

Label: verify exchange exists, create if needed

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	exchange	exchange-name	
3	type	shortstr	exchange type
4	passive	bit	do not create exchange
5	durable	bit	request a durable exchange
6	reserved-2		
7	reserved-3		
8	no-wait	no-wait	
9	arguments	table	arguments for declaration

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

Guidelines for implementers:

- The server SHOULD support a minimum of 16 exchanges per virtual host and ideally, impose no limit except as defined by available resources.

  Test scenario: The client declares as many exchanges as it can until the server reports an error; the number of exchanges successfully declared must be at least sixteen.

1.6.2.1.1. Parameter exchange.declare.reserved-1 ()

Ordinal: 1

Domain:

1.6.2.1.2. Parameter exchange.declare.exchange (exchange-name)

Ordinal: 2

Domain: exchange-name

1.6.2.1.3. Parameter exchange.declare.type (shortstr)

Ordinal: 3

Domain: shortstr

Label: exchange type

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.

1.6.2.1.4. Parameter exchange.declare.passive (bit)

Ordinal: 4

Domain: bit

Label: do not create exchange

If set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not. The client can use this to check whether an exchange exists without modifying the server state. When set, all other method fields except name and no-wait are ignored. A declare with both passive and no-wait has no effect. Arguments are compared for semantic equivalence.

1.6.2.1.5. Parameter exchange.declare.durable (bit)

Ordinal: 5

Domain: bit

Label: request a durable exchange

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.

1.6.2.1.6. Parameter exchange.declare.reserved-2 ()

Ordinal: 6

Domain:

1.6.2.1.7. Parameter exchange.declare.reserved-3 ()

Ordinal: 7

Domain:

1.6.2.1.8. Parameter exchange.declare.no-wait (no-wait)

Ordinal: 8

Domain: no-wait

1.6.2.1.9. Parameter exchange.declare.arguments (table)

Ordinal: 9

Domain: table

Label: arguments for declaration

A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation.

1.6.2.2. Method exchange.declare-ok (ID 11)

ID: 11

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm exchange declaration

This method confirms a Declare method and confirms the name of the exchange, essential for automatically-named exchanges.

1.6.2.3. Method exchange.delete (ID 20)

ID: 20

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***exchange.delete-ok***

Number of parameters: 4

Label: delete an exchange

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	exchange	exchange-name	
3	if-unused	bit	delete only if unused
4	no-wait	no-wait	

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

1.6.2.3.1. Parameter exchange.delete.reserved-1 ()

Ordinal: 1

Domain:

1.6.2.3.2. Parameter exchange.delete.exchange (exchange-name)

Ordinal: 2

Domain: exchange-name

1.6.2.3.3. Parameter exchange.delete.if-unused (bit)

Ordinal: 3

Domain: bit

Label: delete only if unused

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 a channel exception instead.

1.6.2.3.4. Parameter exchange.delete.no-wait (no-wait)

Ordinal: 4

Domain: no-wait

1.6.2.4. Method exchange.delete-ok (ID 21)

ID: 21

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm deletion of an exchange

This method confirms the deletion of an exchange.

1.7. Class 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.

Class Grammar:

```
queue                = C:DECLARE   S:DECLARE-OK
                     / C:BIND      S:BIND-OK
                     / C:UNBIND    S:UNBIND-OK
                     / C:PURGE     S:PURGE-OK
                     / C:DELETE    S:DELETE-OK
```

1.7.1. Property and Method Summary

Class ***queue*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
declare	10	declare-ok	declare queue, create if needed	Y		reserved-1		
						queue	queue-name	
						passive	bit	do not create queue
						durable	bit	request a durable queue
						exclusive	bit	request an exclusive queue
						auto-delete	bit	auto-delete queue when unused
						no-wait	no-wait	
						arguments	table	arguments for declaration
declare-ok	11		confirms a queue definition		Y	queue	queue-name	
						message-count	message-count	
						consumer-count	long	number of consumers
bind	20	bind-ok	bind queue to an exchange	Y		reserved-1		
						queue	queue-name	
						exchange	exchange-name	name of the exchange to bind to
						routing-key	shortstr	message routing key
						no-wait	no-wait	
						arguments	table	arguments for binding
bind-ok	21		confirm bind successful		Y	[No parameters defined for this method]		
unbind	50	unbind-ok	unbind a queue from an	Y		reserved-1		

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
			exchange			queue	queue-name	
						exchange	exchange-name	
						routing-key	shortstr	routing key of binding
						arguments	table	arguments of binding
unbind-ok	51		confirm unbind successful		Y	[No parameters defined for this method]		
purge	30	purge-ok	purge a queue	Y		reserved-1		
						queue	queue-name	
						no-wait	no-wait	
purge-ok	31		confirms a queue purge		Y	message-count	message-count	
delete	40	delete-ok	delete a queue	Y		reserved-1		
						queue	queue-name	
						if-unused	bit	delete only if unused
						if-empty	bit	delete only if empty
						no-wait	no-wait	
delete-ok	41		confirm deletion of a queue		Y	message-count	message-count	

1.7.2. Methods

1.7.2.1. Method queue.declare (ID 10)

ID: 10

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***queue.declare-ok***

Number of parameters: 8

Label: declare queue, create if needed

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	passive	bit	do not create queue
4	durable	bit	request a durable queue
5	exclusive	bit	request an exclusive queue
6	auto-delete	bit	auto-delete queue when unused
7	no-wait	no-wait	

8	arguments	table	arguments for declaration

This method 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.

Guidelines for implementers:

- The server MUST create a default binding for a newly-declared queue to the default exchange, which is an exchange of type 'direct' and use the queue name as the routing key.

  Test scenario: Client declares 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 routing key.

- The server SHOULD support a minimum of 256 queues per virtual host and ideally, impose no limit except as defined by available resources.

  Test scenario: Client attempts to declare as many queues as it can until the server reports an error. The resulting count must at least be 256.

1.7.2.1.1. Parameter queue.declare.reserved-1 ()

Ordinal: 1

Domain:

1.7.2.1.2. Parameter queue.declare.queue (queue-name)

Ordinal: 2

Domain: queue-name

1.7.2.1.3. Parameter queue.declare.passive (bit)

Ordinal: 3

Domain: bit

Label: do not create queue

If set, the server will reply with Declare-Ok if the queue already exists with the same name, and raise an error if not. The client can use this to check whether a queue exists without modifying the server state. When set, all other method fields except name and no-wait are ignored. A declare with both passive and no-wait has no effect. Arguments are compared for semantic equivalence.

1.7.2.1.4. Parameter queue.declare.durable (bit)

Ordinal: 4

Domain: bit

Label: request a durable queue

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.

1.7.2.1.5. Parameter queue.declare.exclusive (bit)

Ordinal: 5

Domain: bit

Label: request an exclusive queue

Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes. Passive declaration of an exclusive queue by other connections are not allowed.

1.7.2.1.6. Parameter queue.declare.auto-delete (bit)

Ordinal: 6

Domain: bit

Label: auto-delete queue when unused

If set, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Applications can explicitly delete auto-delete queues using the Delete method as normal.

1.7.2.1.7. Parameter queue.declare.no-wait (no-wait)

Ordinal: 7

Domain: no-wait

1.7.2.1.8. Parameter queue.declare.arguments (table)

Ordinal: 8

Domain: table

Label: arguments for declaration

A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation.

1.7.2.2. Method queue.declare-ok (ID 11)

ID: 11

Method accepted by: Client

Synchronous: No

Number of parameters: 3

Label: confirms a queue definition

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description

1	queue	queue-name	
2	message-count	message-count	
3	consumer-count	long	number of consumers

This method confirms a Declare method and confirms the name of the queue, essential for automatically-named queues.

1.7.2.2.1. Parameter queue.declare-ok.queue (queue-name)

Ordinal: 1

Domain: queue-name

Reports the name of the queue. If the server generated a queue name, this field contains that name.

1.7.2.2.2. Parameter queue.declare-ok.message-count (message-count)

Ordinal: 2

Domain: message-count

1.7.2.2.3. Parameter queue.declare-ok.consumer-count (long)

Ordinal: 3

Domain: long

Label: number of consumers

Reports the number of active consumers for the queue. Note that consumers can suspend activity (Channel.Flow) in which case they do not appear in this count.

1.7.2.3. Method queue.bind (ID 20)

ID: 20

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***queue.bind-ok***

Number of parameters: 6

Label: bind queue to an exchange

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	exchange	exchange-name	name of the exchange to bind to
4	routing-key	shortstr	message routing key
5	no-wait	no-wait	
6	arguments	table	arguments for binding

This method 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.

Guidelines for implementers:

- A server MUST allow ignore duplicate bindings - that is, two or more bind methods for a specific queue, with identical arguments - without treating these as an error.

  Test scenario: A client binds a named queue to an exchange. The client then repeats the bind (with identical arguments).

- A server MUST not deliver the same message more than once to a queue, even if the queue has multiple bindings that match the message.

  Test scenario: A client declares a named queue and binds it using multiple bindings to the amq.topic exchange. The client then publishes a message that matches all its bindings.

- The server MUST allow a durable queue to bind to a transient exchange.

  Test scenario: A client declares a transient exchange. The client then declares a named durable queue and then attempts to bind the transient exchange to the durable queue.

- Bindings of durable queues to durable exchanges are automatically durable and the server MUST restore such bindings after a server restart.

  Test scenario: A server declares 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.

- The server SHOULD support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.

  Test scenario: A client declares a named queue and attempts to bind it to 4 different exchanges.

1.7.2.3.1. Parameter queue.bind.reserved-1 ()

Ordinal: 1

Domain:

1.7.2.3.2. Parameter queue.bind.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to bind.

1.7.2.3.3. Parameter queue.bind.exchange (exchange-name)

Ordinal: 3

Domain: exchange-name

Label: name of the exchange to bind to

1.7.2.3.4. Parameter queue.bind.routing-key (shortstr)

Ordinal: 4

Domain: shortstr

Label: message routing key

Specifies the routing key for the binding. The routing key is used for routing messages depending on the exchange configuration. Not all exchanges use a routing key - refer to the specific exchange documentation. If the queue name is empty, the server uses the last queue declared on the channel. If the routing key is also empty, the server uses this queue name for the routing key as well. If the queue name is provided but the routing key is empty, the server does the binding with that empty routing key. The meaning of empty routing keys depends on the exchange implementation.

1.7.2.3.5. Parameter queue.bind.no-wait (no-wait)

Ordinal: 5

Domain: no-wait

1.7.2.3.6. Parameter queue.bind.arguments (table)

Ordinal: 6

Domain: table

Label: arguments for binding

A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

1.7.2.4. Method queue.bind-ok (ID 21)

ID: 21

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm bind successful

This method confirms that the bind was successful.

1.7.2.5. Method queue.unbind (ID 50)

ID: 50

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***queue.unbind-ok***

Number of parameters: 5

Label: unbind a queue from an exchange

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	exchange	exchange-name	
4	routing-key	shortstr	routing key of binding
5	arguments	table	arguments of binding

This method unbinds a queue from an exchange.

Guidelines for implementers:

- If a unbind fails, the server MUST raise a connection exception.

1.7.2.5.1. Parameter queue.unbind.reserved-1 ()

Ordinal: 1

Domain:

1.7.2.5.2. Parameter queue.unbind.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to unbind.

1.7.2.5.3. Parameter queue.unbind.exchange (exchange-name)

Ordinal: 3

Domain: exchange-name

The name of the exchange to unbind from.

1.7.2.5.4. Parameter queue.unbind.routing-key (shortstr)

Ordinal: 4

Domain: shortstr

Label: routing key of binding

Specifies the routing key of the binding to unbind.

1.7.2.5.5. Parameter queue.unbind.arguments (table)

Ordinal: 5

Domain: table

Label: arguments of binding

Specifies the arguments of the binding to unbind.

1.7.2.6. Method queue.unbind-ok (ID 51)

ID: 51

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm unbind successful

This method confirms that the unbind was successful.

1.7.2.7. Method queue.purge (ID 30)

ID: 30

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***queue.purge-ok***

Number of parameters: 3

Label: purge a queue

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	no-wait	no-wait	

This method removes all messages from a queue which are not awaiting acknowledgment.

Guidelines for implementers:

- The server MUST NOT purge messages that have already been sent to a client but not yet acknowledged.
- 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.

1.7.2.7.1. Parameter queue.purge.reserved-1 ()

Ordinal: 1

Domain:

1.7.2.7.2. Parameter queue.purge.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to purge.

1.7.2.7.3. Parameter queue.purge.no-wait (no-wait)

Ordinal: 3

Domain: no-wait

1.7.2.8. Method queue.purge-ok (ID 31)

ID: 31

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: confirms a queue purge

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	message-count	message-count	

This method confirms the purge of a queue.

1.7.2.8.1. Parameter queue.purge-ok.message-count (message-count)

Ordinal: 1

Domain: message-count

Reports the number of messages purged.

1.7.2.9. Method queue.delete (ID 40)

ID: 40

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***queue.delete-ok***

Number of parameters: 5

Label: delete a queue

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	if-unused	bit	delete only if unused
4	if-empty	bit	delete only if empty
5	no-wait	no-wait	

This method deletes a queue. When a queue is deleted any pending messages are sent to a dead-letter

queue if this is defined in the server configuration, and all consumers on the queue are cancelled.

Guidelines for implementers:

- The server SHOULD use a dead-letter queue to hold messages that were pending on a deleted queue, and MAY provide facilities for a system administrator to move these messages back to an active queue.

1.7.2.9.1. Parameter queue.delete.reserved-1 ()

Ordinal: 1

Domain:

1.7.2.9.2. Parameter queue.delete.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to delete.

1.7.2.9.3. Parameter queue.delete.if-unused (bit)

Ordinal: 3

Domain: bit

Label: delete only if unused

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 a channel exception instead.

1.7.2.9.4. Parameter queue.delete.if-empty (bit)

Ordinal: 4

Domain: bit

Label: delete only if empty

If set, the server will only delete the queue if it has no messages.

1.7.2.9.5. Parameter queue.delete.no-wait (no-wait)

Ordinal: 5

Domain: no-wait

1.7.2.10. Method queue.delete-ok (ID 41)

ID: 41

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: confirm deletion of a queue

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	message-count	message-count	

This method confirms the deletion of a queue.

1.7.2.10.1. Parameter queue.delete-ok.message-count (message-count)

Ordinal: 1

Domain: message-count

Reports the number of messages deleted.

1.8. Class basic

The Basic class provides methods that support an industry-standard messaging model.

Class Grammar:

```
basic                = C:QOS S:QOS-OK
                     / C:CONSUME S:CONSUME-OK
                     / C:CANCEL S:CANCEL-OK
                     / C:PUBLISH content
                     / S:RETURN content
                     / S:DELIVER content
                     / C:GET ( S:GET-OK content / S:GET-EMPTY )
                     / C:ACK
                     / C:REJECT
                     / C:RECOVER-ASYNC
                     / C:RECOVER S:RECOVER-OK
```

Guidelines for implementers:

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

  Test 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.

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

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

- The server MAY use the Channel.Flow method to slow or stop a basic message publisher when necessary.

  Test scenario: Declare a queue overflow situation with non-persistent messages and verify

whether the server responds with Channel.Flow or not. Repeat with persistent messages.

- The server MAY overflow non-persistent basic messages to persistent storage.
- The server MAY discard or dead-letter non-persistent basic messages on a priority basis if the queue size exceeds some configured limit.
- The server MUST implement at least 2 priority levels for basic messages, where priorities 0-4 and 5-9 are treated as two distinct levels.

  Test scenario: Send a number of priority 0 messages to a queue. Send one priority 9 message. Consume messages from the queue and verify that the first message received was priority 9.
- The server MAY implement up to 10 priority levels.

  Test scenario: Send a number of messages with mixed priorities to a queue, so that all priority values from 0 to 9 are exercised. A good scenario would be ten messages in low-to-high priority. Consume from queue and verify how many priority levels emerge.
- The server MUST deliver messages of the same priority in order irrespective of their individual persistence.

  Test scenario: Send a set of messages with the same priority but different persistence settings to a queue. Consume and verify that messages arrive in same order as originally published.
- The server MUST support un-acknowledged delivery of Basic content, i.e. consumers with the no-ack field set to TRUE.
- The server MUST support explicitly acknowledged delivery of Basic content, i.e. consumers with the no-ack field set to FALSE.

  Test scenario: Declare a queue and a consumer using explicit acknowledgements. Publish a set of messages to the queue. Consume the messages but acknowledge only half of them. Disconnect and reconnect, and consume from the queue. Verify that the remaining messages are received.

1.8.1. Property and Method Summary

Class ***basic*** defines the following properties:

Name	Domain	Short Description
content-type	shortstr	MIME content type
content-encoding	shortstr	MIME content encoding
headers	table	message header field table
delivery-mode	octet	non-persistent (1) or persistent (2)
priority	octet	message priority, 0 to 9
correlation-id	shortstr	application correlation identifier
reply-to	shortstr	address to reply to
expiration	shortstr	message expiration specification
message-id	shortstr	application message identifier
timestamp	timestamp	message timestamp
type	shortstr	message type name
user-id	shortstr	creating user id

Name	Domain	Short Description
app-id	shortstr	creating application id
reserved	shortstr	reserved, must be empty

Class ***basic*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
qos	10	qos-ok	specify quality of service	Y		prefetch-size	long	prefetch window in octets
						prefetch-count	short	prefetch window in messages
						global	bit	apply to entire connection
qos-ok	11		confirm the requested qos		Y	[No parameters defined for this method]		
consume	20	consume-ok	start a queue consumer	Y		reserved-1		
						queue	queue-name	
						consumer-tag	consumer-tag	
						no-local	no-local	
						no-ack	no-ack	
						exclusive	bit	request exclusive access
						no-wait	no-wait	
						arguments	table	arguments for declaration
consume-ok	21		confirm a new consumer		Y	consumer-tag	consumer-tag	
cancel	30	cancel-ok	end a queue consumer	Y		consumer-tag	consumer-tag	
						no-wait	no-wait	
cancel-ok	31		confirm a cancelled consumer		Y	consumer-tag	consumer-tag	
publish	40		publish a message	Y		reserved-1		
						exchange	exchange-name	
						routing-key	shortstr	Message routing key
						mandatory	bit	indicate mandatory routing
						immediate	bit	request immediate delivery
return	50		return a failed message		Y	reply-code	reply-code	
						reply-text	reply-text	
						exchange	exchange-name	
						routing-key	shortstr	Message routing key
deliver	60		notify the client of a consumer message		Y	consumer-tag	consumer-tag	
						delivery-tag	delivery-tag	
						redelivered	redelivered	
						exchange	exchange-name	

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
						routing-key	shortstr	Message routing key
get	70	get-ok	direct access to a queue	Y		reserved-1		
						queue	queue-name	
						no-ack	no-ack	
get-ok	71		provide client with a message		Y	delivery-tag	delivery-tag	
						redelivered	redelivered	
						exchange	exchange-name	
						routing-key	shortstr	Message routing key
						message-count	message-count	
get-empty	72		indicate no messages available		Y	reserved-1		
ack	80		acknowledge one or more messages	Y		delivery-tag	delivery-tag	
						multiple	bit	acknowledge multiple messages
reject	90		reject an incoming message	Y		delivery-tag	delivery-tag	
						requeue	bit	requeue the message
recover-async	100		redeliver unacknowledged messages	Y		requeue	bit	requeue the message
recover	110		redeliver unacknowledged messages	Y		requeue	bit	requeue the message
recover-ok	111		confirm recovery		Y	[No parameters defined for this method]		

1.8.2. Properties

1.8.2.1. Property basic.content-type (shortstr)

Domain: shortstr

Label: MIME content type

1.8.2.2. Property basic.content-encoding (shortstr)

Domain: shortstr

Label: MIME content encoding

1.8.2.3. Property basic.headers (table)

Domain: table

Label: message header field table

1.8.2.4. Property basic.delivery-mode (octet)

Domain: octet

Label: non-persistent (1) or persistent (2)

1.8.2.5. Property basic.priority (octet)

Domain: octet

Label: message priority, 0 to 9

1.8.2.6. Property basic.correlation-id (shortstr)

Domain: shortstr

Label: application correlation identifier

1.8.2.7. Property basic.reply-to (shortstr)

Domain: shortstr

Label: address to reply to

1.8.2.8. Property basic.expiration (shortstr)

Domain: shortstr

Label: message expiration specification

1.8.2.9. Property basic.message-id (shortstr)

Domain: shortstr

Label: application message identifier

1.8.2.10. Property basic.timestamp (timestamp)

Domain: timestamp

Label: message timestamp

1.8.2.11. Property basic.type (shortstr)

Domain: shortstr

Label: message type name

1.8.2.12. Property basic.user-id (shortstr)

Domain: shortstr

Label: creating user id

1.8.2.13. Property basic.app-id (shortstr)

Domain: shortstr

Label: creating application id

1.8.2.14. Property basic.reserved (shortstr)

Domain: shortstr

Label: reserved, must be empty

1.8.3. Methods

1.8.3.1. Method basic.qos (ID 10)

ID: 10

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***basic.qos-ok***

Number of parameters: 3

Label: specify quality of service

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	prefetch-size	long	prefetch window in octets
2	prefetch-count	short	prefetch window in messages
3	global	bit	apply to entire connection

This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The particular properties and semantics of a qos method always depend on the content class semantics. Though the qos method could in principle apply to both peers, it is currently meaningful only for the server.

1.8.3.1.1. Parameter basic.qos.prefetch-size (long)

Ordinal: 1

Domain: long

Label: prefetch window in octets

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 down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored if the no-ack option is set.

1.8.3.1.2. Parameter basic.qos.prefetch-count (short)

Ordinal: 2

Domain: short

Label: prefetch window in messages

Specifies a prefetch 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 prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored if the no-ack option is set.

1.8.3.1.3. Parameter basic.qos.global (bit)

Ordinal: 3

Domain: bit

Label: apply to entire connection

By default the QoS settings apply to the current channel only. If this field is set, they are applied to the entire connection.

1.8.3.2. Method basic.qos-ok (ID 11)

ID: 11

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm the requested qos

This method 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.

1.8.3.3. Method basic.consume (ID 20)

ID: 20

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***basic.consume-ok***

Number of parameters: 8

Label: start a queue consumer

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	consumer-tag	consumer-tag	
4	no-local	no-local	
5	no-ack	no-ack	
6	exclusive	bit	request exclusive access
7	no-wait	no-wait	

8	arguments	table	arguments for declaration

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were declared on, or until the client cancels them.

Guidelines for implementers:

- The server SHOULD support at least 16 consumers per queue, and ideally, impose no limit except as defined by available resources.

  Test scenario: Declare a queue and create consumers on that queue until the server closes the connection. Verify that the number of consumers created was at least sixteen and report the total number.

1.8.3.3.1. Parameter basic.consume.reserved-1 ()

Ordinal: 1

Domain:

1.8.3.3.2. Parameter basic.consume.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to consume from.

1.8.3.3.3. Parameter basic.consume.consumer-tag (consumer-tag)

Ordinal: 3

Domain: consumer-tag

Specifies the identifier for the consumer. The consumer tag is local to a channel, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.

1.8.3.3.4. Parameter basic.consume.no-local (no-local)

Ordinal: 4

Domain: no-local

1.8.3.3.5. Parameter basic.consume.no-ack (no-ack)

Ordinal: 5

Domain: no-ack

1.8.3.3.6. Parameter basic.consume.exclusive (bit)

Ordinal: 6

Domain: bit

Label: request exclusive access

Request exclusive consumer access, meaning only this consumer can access the queue.

1.8.3.3.7. Parameter basic.consume.no-wait (no-wait)

Ordinal: 7

Domain: no-wait

1.8.3.3.8. Parameter basic.consume.arguments (table)

Ordinal: 8

Domain: table

Label: arguments for declaration

A set of arguments for the consume. The syntax and semantics of these arguments depends on the server implementation.

1.8.3.4. Method basic.consume-ok (ID 21)

ID: 21

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: confirm a new consumer

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	consumer-tag	consumer-tag	

The server provides the client with a consumer tag, which is used by the client for methods called on the consumer at a later stage.

1.8.3.4.1. Parameter basic.consume-ok.consumer-tag (consumer-tag)

Ordinal: 1

Domain: consumer-tag

Holds the consumer tag specified by the client or provided by the server.

1.8.3.5. Method basic.cancel (ID 30)

ID: 30

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***basic.cancel-ok***

Number of parameters: 2

Label: end a queue consumer

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	consumer-tag	consumer-tag	
2	no-wait	no-wait	

This method 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. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

Guidelines for implementers:

- If the queue does not exist the server MUST ignore the cancel method, so long as the consumer tag is valid for that channel.

  Test scenario: TODO.

1.8.3.5.1. Parameter basic.cancel.consumer-tag (consumer-tag)

Ordinal: 1

Domain: consumer-tag

1.8.3.5.2. Parameter basic.cancel.no-wait (no-wait)

Ordinal: 2

Domain: no-wait

1.8.3.6. Method basic.cancel-ok (ID 31)

ID: 31

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: confirm a cancelled consumer

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	consumer-tag	consumer-tag	

This method confirms that the cancellation was completed.

1.8.3.6.1. Parameter basic.cancel-ok.consumer-tag (consumer-tag)

Ordinal: 1

Domain: consumer-tag

1.8.3.7. Method basic.publish (ID 40)

ID: 40

Method accepted by: Server

Synchronous: No

Number of parameters: 5

Label: publish a message

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	exchange	exchange-name	
3	routing-key	shortstr	Message routing key
4	mandatory	bit	indicate mandatory routing
5	immediate	bit	request immediate delivery

This method 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 when the transaction, if any, is committed.

1.8.3.7.1. Parameter basic.publish.reserved-1 ()

Ordinal: 1

Domain:

1.8.3.7.2. Parameter basic.publish.exchange (exchange-name)

Ordinal: 2

Domain: exchange-name

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 a channel exception.

1.8.3.7.3. Parameter basic.publish.routing-key (shortstr)

Ordinal: 3

Domain: shortstr

Label: Message routing key

Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.

1.8.3.7.4. Parameter basic.publish.mandatory (bit)

Ordinal: 4

Domain: bit

Label: indicate mandatory routing

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 method. If this flag is zero, the server silently drops the message.

1.8.3.7.5. Parameter basic.publish.immediate (bit)

Ordinal: 5

Domain: bit

Label: request immediate delivery

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 method. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed.

1.8.3.8. Method basic.return (ID 50)

ID: 50

Method accepted by: Client

Synchronous: No

Number of parameters: 4

Label: return a failed message

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reply-code	reply-code	
2	reply-text	reply-text	
3	exchange	exchange-name	
4	routing-key	shortstr	Message routing key

This method 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.

1.8.3.8.1. Parameter basic.return.reply-code (reply-code)

Ordinal: 1

Domain: reply-code

1.8.3.8.2. Parameter basic.return.reply-text (reply-text)

Ordinal: 2

Domain: reply-text

1.8.3.8.3. Parameter basic.return.exchange (exchange-name)

Ordinal: 3

Domain: exchange-name

Specifies the name of the exchange that the message was originally published to. May be empty, meaning the default exchange.

1.8.3.8.4. Parameter basic.return.routing-key (shortstr)

Ordinal: 4

Domain: shortstr

Label: Message routing key

Specifies the routing key name specified when the message was published.

1.8.3.9. Method basic.deliver (ID 60)

ID: 60

Method accepted by: Client

Synchronous: No

Number of parameters: 5

Label: notify the client of a consumer message

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	consumer-tag	consumer-tag	
2	delivery-tag	delivery-tag	
3	redelivered	redelivered	
4	exchange	exchange-name	
5	routing-key	shortstr	Message routing key

This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

Guidelines for implementers:

- 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 unprocessable (possibly causing client applications to abort), and move the message to a dead letter queue.

Test scenario: TODO.

1.8.3.9.1. Parameter basic.deliver.consumer-tag (consumer-tag)

Ordinal: 1

Domain: consumer-tag

1.8.3.9.2. Parameter basic.deliver.delivery-tag (delivery-tag)

Ordinal: 2

Domain: delivery-tag

1.8.3.9.3. Parameter basic.deliver.redelivered (redelivered)

Ordinal: 3

Domain: redelivered

1.8.3.9.4. Parameter basic.deliver.exchange (exchange-name)

Ordinal: 4

Domain: exchange-name

Specifies the name of the exchange that the message was originally published to. May be empty, indicating the default exchange.

1.8.3.9.5. Parameter basic.deliver.routing-key (shortstr)

Ordinal: 5

Domain: shortstr

Label: Message routing key

Specifies the routing key name specified when the message was published.

1.8.3.10. Method basic.get (ID 70)

ID: 70

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***basic.get-ok, basic.get-empty***

Number of parameters: 3

Label: direct access to a queue

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		
2	queue	queue-name	
3	no-ack	no-ack	

This method provides a direct access to the messages in a queue using a synchronous dialogue that is designed for specific types of application where synchronous functionality is more important than performance.

1.8.3.10.1. Parameter basic.get.reserved-1 ()

Ordinal: 1

Domain:

1.8.3.10.2. Parameter basic.get.queue (queue-name)

Ordinal: 2

Domain: queue-name

Specifies the name of the queue to get a message from.

1.8.3.10.3. Parameter basic.get.no-ack (no-ack)

Ordinal: 3

Domain: no-ack

1.8.3.11. Method basic.get-ok (ID 71)

ID: 71

Method accepted by: Client

Synchronous: No

Number of parameters: 5

Label: provide client with a message

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	delivery-tag	delivery-tag	
2	redelivered	redelivered	
3	exchange	exchange-name	
4	routing-key	shortstr	Message routing key
5	message-count	message-count	

This method delivers a message to the client following a get method. A message delivered by 'get-ok' must

be acknowledged unless the no-ack option was set in the get method.

1.8.3.11.1. Parameter basic.get-ok.delivery-tag (delivery-tag)

Ordinal: 1

Domain: delivery-tag

1.8.3.11.2. Parameter basic.get-ok.redelivered (redelivered)

Ordinal: 2

Domain: redelivered

1.8.3.11.3. Parameter basic.get-ok.exchange (exchange-name)

Ordinal: 3

Domain: exchange-name

Specifies the name of the exchange that the message was originally published to. If empty, the message was published to the default exchange.

1.8.3.11.4. Parameter basic.get-ok.routing-key (shortstr)

Ordinal: 4

Domain: shortstr

Label: Message routing key

Specifies the routing key name specified when the message was published.

1.8.3.11.5. Parameter basic.get-ok.message-count (message-count)

Ordinal: 5

Domain: message-count

1.8.3.12. Method basic.get-empty (ID 72)

ID: 72

Method accepted by: Client

Synchronous: No

Number of parameters: 1

Label: indicate no messages available

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	reserved-1		

This method tells the client that the queue has no messages available for the client.

1.8.3.12.1. Parameter basic.get-empty.reserved-1 ()

Ordinal: 1

Domain:

1.8.3.13. Method basic.ack (ID 80)

ID: 80

Method accepted by: Server

Synchronous: No

Number of parameters: 2

Label: acknowledge one or more messages

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	delivery-tag	delivery-tag	
2	multiple	bit	acknowledge multiple messages

This method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

1.8.3.13.1. Parameter basic.ack.delivery-tag (delivery-tag)

Ordinal: 1

Domain: delivery-tag

1.8.3.13.2. Parameter basic.ack.multiple (bit)

Ordinal: 2

Domain: bit

Label: acknowledge multiple messages

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 method. 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.

1.8.3.14. Method basic.reject (ID 90)

ID: 90

Method accepted by: Server

Synchronous: No

Number of parameters: 2

Label: reject an incoming message

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	delivery-tag	delivery-tag	
2	requeue	bit	requeue the message

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

Guidelines for implementers:

- The server SHOULD be capable of accepting and process the Reject method while sending message content with a Deliver or Get-Ok method. I.e. the server should read and process incoming methods while sending output frames. To cancel a partially-send content, the server sends a content body frame of size 1 (i.e. with no data except the frame-end octet).
- The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.

  Test scenario: TODO.
- The client MUST NOT use this method as a means of selecting messages to process.

  Test scenario: TODO.

1.8.3.14.1. Parameter basic.reject.delivery-tag (delivery-tag)

Ordinal: 1

Domain: delivery-tag

1.8.3.14.2. Parameter basic.reject.requeue (bit)

Ordinal: 2

Domain: bit

Label: requeue the message

If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

1.8.3.15. Method basic.recover-async (ID 100)

ID: 100

Method accepted by: Server

Synchronous: No

Number of parameters: 1

Label: redeliver unacknowledged messages

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	requeue	bit	requeue the message

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method is deprecated in favour of the synchronous Recover/Recover-Ok.

Guidelines for implementers:

- The server MUST set the redelivered flag on all messages that are resent.

  Test scenario: TODO.

1.8.3.15.1. Parameter basic.recover-async.requeue (bit)

Ordinal: 1

Domain: bit

Label: requeue the message

If this field is zero, the message will be redelivered to the original recipient. If this bit is 1, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

1.8.3.16. Method basic.recover (ID 110)

ID: 110

Method accepted by: Server

Synchronous: No

Number of parameters: 1

Label: redeliver unacknowledged messages

Parameter Summary:

Ordinal	Parameter name	Domain	Short Description
1	requeue	bit	requeue the message

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

Guidelines for implementers:

- The server MUST set the redelivered flag on all messages that are resent.

  Test scenario: TODO.

1.8.3.16.1. Parameter basic.recover.requeue (bit)

Ordinal: 1

Domain: bit

Label: requeue the message

If this field is zero, the message will be redelivered to the original recipient. If this bit is 1, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

1.8.3.17. Method basic.recover-ok (ID 111)

ID: 111

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm recovery

This method acknowledges a Basic.Recover method.

1.9. Class tx

The Tx class allows publish and ack operations to be batched into atomic units of work. The intention is that all publish and ack requests issued within a transaction will complete successfully or none of them will. Servers SHOULD implement atomic transactions at least where all publish or ack requests affect a single queue. Transactions that cover multiple queues may be non-atomic, given that queues can be created and destroyed asynchronously, and such events do not form part of any transaction. Further, the behaviour of transactions with respect to the immediate and mandatory flags on Basic.Publish methods is not defined.

Class Grammar:

```
tx                    = C:SELECT S:SELECT-OK
                      / C:COMMIT S:COMMIT-OK
                      / C:ROLLBACK S:ROLLBACK-OK
```

Guidelines for implementers:

- Applications MUST NOT rely on the atomicity of transactions that affect more than one queue.
- Applications MUST NOT rely on the behaviour of transactions that include messages published with the immediate option.
- Applications MUST NOT rely on the behaviour of transactions that include messages published with the mandatory option.

1.9.1. Property and Method Summary

Class ***tx*** defines the following methods (S = received by server; C = received by client):

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
select	10	select-ok	select standard transaction mode	Y		[No parameters defined for this method]		

Name	ID	Sync. Response	Short description	S	C	Field Name	Domain	Short Description
select-ok	11		confirm transaction mode		Y	[No parameters defined for this method]		
commit	20	commit-ok	commit the current transaction	Y		[No parameters defined for this method]		
commit-ok	21		confirm a successful commit		Y	[No parameters defined for this method]		
rollback	30	rollback-ok	abandon the current transaction	Y		[No parameters defined for this method]		
rollback-ok	31		confirm successful rollback		Y	[No parameters defined for this method]		

1.9.2. Methods

1.9.2.1. Method tx.select (ID 10)

ID: 10

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***tx.select-ok***

Number of parameters: 0

Label: select standard transaction mode

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

1.9.2.2. Method tx.select-ok (ID 11)

ID: 11

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm transaction mode

This method confirms to the client that the channel was successfully set to use standard transactions.

1.9.2.3. Method tx.commit (ID 20)

ID: 20

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***tx.commit-ok***

Number of parameters: 0

Label: commit the current transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit.

Guidelines for implementers:

- The client MUST NOT use the Commit method on non-transacted channels.

  Test scenario: The client opens a channel and then uses Tx.Commit.

  On failure: Constant "precondition-failed" (See AMQP-defined Constants)

1.9.2.4. Method tx.commit-ok (ID 21)

ID: 21

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm a successful commit

This method confirms to the client that the commit succeeded. Note that if a commit fails, the server raises a channel exception.

1.9.2.5. Method tx.rollback (ID 30)

ID: 30

Method accepted by: Server

Synchronous: Yes; expected response is from method(s) ***tx.rollback-ok***

Number of parameters: 0

Label: abandon the current transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued.

Guidelines for implementers:

- The client MUST NOT use the Rollback method on non-transacted channels.

  Test scenario: The client opens a channel and then uses Tx.Rollback.

  On failure: Constant "precondition-failed" (See AMQP-defined Constants)

1.9.2.6. Method tx.rollback-ok (ID 31)

ID: 31

Method accepted by: Client

Synchronous: No

Number of parameters: 0

Label: confirm successful rollback

This method confirms to the client that the rollback succeeded. Note that if an rollback fails, the server raises a channel exception.