Twino
Messaging Queue Server 3.0

Table of Contents

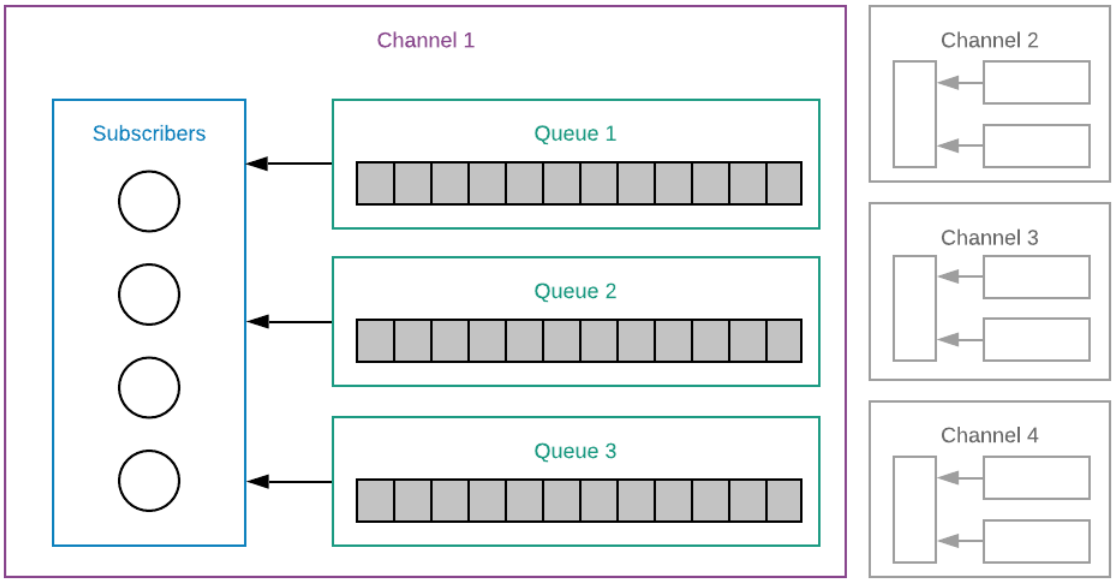
Overview

Twino Messaging Queue is a .NET Core Framework. It's not an executable application. You must create your own .NET Core project and implement interfaces of Twino MQ Server. Because of this, the limits of the Twino MQ server is just you. However, some unique features cannot be supported by Twino due to it's structure.

Implementing Twino MQ Server to your project is easy. You need to write about 10-lines of code and implement at least one delivery handler interface.

Twino has it's own aim and aspect about messaging queue operations:

- Twino has a good Broadcasting architecture. If you need to broadcast same message at same time to thousands of consumers and don't care about messages in past, you can do it by using about 300MB RAM and 2 vCPUs.
- Twino categorizes applications which are using same multiple type of messages with multiple type of queues. They join to same channel and consumes multiple queues at same time.<
- When things change often, some messages in queues should be in pull state in a time, but a few minutes later they should be in push state, or the queue must be paused for a while, Twino gives you that dynamism very well.
- Twino has tons of options. You can customize your channels and queues with a wide variety.



If a queue status is broadcast or push. When a message is received, it is sent to all subscribers of the channel. If queue status is pull or cache, consumers can pull messages from the queues. Server can has unlimited channels and queues.

Features and Limitations

Here are some features and differences from other messaging queue applications:

- Clients subscribes/unsubscribes to channels, not queues. A channel can include multiple queues and multiple consumers. Channel consumers can receive from all queues in the channel.
- Twino aims to use less resource. You can send thousands of messages in a second by using 1 CPU and 100MB RAM.
- Each queue has it's own status can be changed dynamically without any reset. Statuses are broadcast, push, round-robin, pull, cache, pause, stop. Each status has different publish and consume architecture.
- Multiple consumers can consume same queue messages at same time. Same message in same queue can be sent to unlimited clients without copy.
- Queues guarantee atomic FIFO. But in cases when three or more messages is received in same microsecond/10 from different producers, FIFO is not guaranteed. Actually nobody can know who sent it's message over network first.
- Queue features and options are; one delivery, acknowledge from server/consumer to producer, ack timeout, message timeout, hidden producer, unique message id, waiting for ack to continue, message limit, message size limit, high-low priority messages, persistent and durable messages, authentication and authorization for each event.
- Channel features and options are; single/multiple queues, allowing specific queues, consumer limit, queue limit, auto destroy, authentication and authorization for each event.
- It's possible to send direct message to a client which is connected to the server and wait for response. Twino supports parallel messages at same time. More messages can be sent before receive previous responses asynchoronously.
- Sending Acknowledge, saving message to disk, removing message from queue decisions up to users. In lifecycle of a message, there are many steps and you can decide when.
- Twino MQ can run as clusters. Each node can be master, slave or master and slave. Slave nodes sends the messages to masters. Sending messages can be filtered. But a single queue cannot be scaled horizontaly.

Protocol Limitations

- Message Id, Message Source, Target, Channel Name values max length is 255.
- Each channel can has 65535 (max value of unsigned short) queues.
- Channel names can't include " " or ";" characters.
- Header max length is 65535 (max value of unsigned short)
- Message content length is max value of unsigned Int64.

Protocol Frame

Default (0) OneDelivery (1)	Default (0) High Priority (1)	Message Type (6 bits)	
Pending Response	Pending Acknowledge	NoHeader (0) Header (1)	Message TTL (5 bits)
Message Id Length			
Message Source (Producer) Id Length			
Message Target (Channel or Client) Id Length			
Content Type First Byte (16-bit unsigned)			
Content Type First Byte (16-bit unsigned)			
[1 to 9 Bytes Length] Payload Length			
Message Id			
Message Source			
Message Target			
[If Header Exists] Header Length First Byte (16-bit unsigned)			
[If Header Exists] Header Length Second Byte (16-bit unsigned)			
[If Header Exists] Header Data			
Message Payload			

- Octet 1 – Bit 1, FALSE is default. TRUE, message will be delivered by only one consumer.
- Octet 1 – Bit 2, FALSE is normal priority message. TRUE is high priority message.
- Octet 1 – Next 6 Bits, Message type is described below.
- Octet 2 – Bit 1, TRUE means message should be respond, sender is waiting a response.
- Octet 2 – Bit 2, TRUE means sender is interested ack or nack as response.
- Octet 2 – Bit 3, FALSE mean message does not have header data. TRUE means it has.
- Octet 2 – Next 5 Bits, Message TTL value.
- Next each row corresponds to 1 octet.
- Lines with yellow background size value of previous lines.
- Lines with bold If conditions exists if the condition is TRUE in first 2 octets. If the condition is FALSE, these lines are not expected.
- Last 4 lines; Message Id Length, Message Id, Payload Length and Payload.

Message Types

0x00	**Custom**	These type of messages are not proceed in messaging queue server. Use when you need to process some custom messages in server side.
0x08	**Terminate**	Client or server notifies that the connection is closing.
0x09	**Ping**	Twino Messaging Queue Protocol PING Message.
0x0A	**Pong**	Twino Messaging Queue Protocol PONG Message.
0x10	**Server Operation**	Clients sends an message for an operation to server. These messages are RPC style. They usually expect acknowledge or response.
0x11	**Queue Message**	If that type of message is sent from a client to a server, message is pushed into a queue. If it's sent from a server to a client, it means client is consuming the message from the queue.
0x12	**Direct Message**	Direct messages between clients. They can request acknowledge or response. Messaging Queue server just transmits these messages between them without any process.
0x13	**Acknowledge**	Acknowledge or negative acknowledge message of a message. Acknowledge messages have same message Id with their original messages and additional their types are 0x15. So, every message can has an acknowledge.
0x14	**Response**	Response message of any message. Response messages have same message Id with their request messages and additional their types are 0x16. So, every message can has a response.
0x15	**Queue Pull Request**	Consumer sends a pull request to a queue to receive messages. Pull, Paused or Cache queues don't send messages directly. Consumers should send a pull request to consume messages.
0x16	**Event**	If message is from client to server, it means client subscribes or unsubscribes from an event. If it's from server to a client, an event is occurred and server sends the information about it.
0x17	**Router**	Message is routed to a custom exchange in a server. That exchange implementation can push the message into multiple queues or send it directly to a client.

Version and Handshaking

In this version of TMQ Server (3.), Twino Message Queue Protocol version 2.0 is used. In this protocol, when a client is connected to server, It should send 8 bytes protocol message first. If server accepts the protocol and version, as response server send same 8 bytes protocol message to client. Last, client sends connection information message as first message. After these 3 messages the connection is ready to communicate.

Client can send connection data before receive 8 bytes protocol message from the server. It does not have to wait for protocol reply to send information message. But, if the connection is refused, protocol is not accepted, this message will not be proceed. Maybe the server uses different protocol and can miss-read the connection data message. In some cases, if you are sure the server is TMQ Server, you can feel safe to send connection data message before reading 8 bytes protocol message. This can lower your handshaking duration.
8 Bytes protocol message **TMQP/2.0** string message. Message in bytes:

```
0x54 0x4D 0x51 0x50 0x2F 0x32 0x2E 0x30
```

Connection Info message is a message in protocol frame. Message content is HTTP Request like. First line is two words. First word is remote path, which remove host name client use. Second word is a method. If you specify a message to tell server, you can type something. Otherwise method is sent as NONE. Other lines are HTTP Header like key and values. Properties can be defined by clients to some send information to server such as Client Id, Client Name, Client Type, Authorization Token, or some other. Here is a sample to connection info.

```
tmq://remote-host CONNECT
Client-Id: unique-client-id
Client-Name: E-Mail-Sender
Client-Type: Worker-Application
Client-Token: Bearer ey..abc..123..xyz
```

These 4 properties are used by Twino Messaging Queue Server. If you don't send these properties, they will be null or auto-defined. If you don't specify client id, server will generate a unique id for the client. You can add your own properties in header. They are stored as connection data in server-side.

PING and PONG Messages

In Twino, PING and PONG messages are sent if really necessary. If server keeps sending messages to clients and clients keep sending messages to server, PING and PONG messages are not necessary. So, server does not send PING message and does not wait for PONG response.

But when connection is established but there is no message sending and receiving, server sends PING message and waits PONG from client.

Twino Messaging Queue Protocol 3.0 PING and PONG messages are same with version 2.0. Each message must be 8 bytes. In Twino, each frame must have at least 8 bytes, so do PING and PONG messages. But we need only first two bytes to recognize if the message is PING or PONG. Other bytes must be 0x00 and they are required.

PING Message
`0x89 0xFF 0x00 0x00 0x00 0x00 0x00 0x00`

PONG Message
`0x8A 0xFF 0x00 0x00 0x00 0x00 0x00 0x00`

Queue Statuses

Queue statuses decides how messages will be proceed and sent. Queue status can be changed after it's created.

Broadcast

Broadcast status does not keep messages in memory or in database file. Broadcast status just sends messages to consumers when it's received from producer. If there is no available consumers, message is lost. Broadcast status guarantees that each consumer receives same message at same time. It's like watching live video from a source.

- Messages are not saved to database file.
- Messages are immediately sent to all available consumers when they received.
- If there are no available consumers, messages are lost.
- They have the best performance. They can deliver hundred thousands messages per second.

Push

Push status works like Broadcast status. When a producer pushes a message, it is sent to available consumers immediately. But if there are no consumers, message is stored in the queue unlike Broadcast. Each message is sent at least one consumer before it's removed. Messages can be saved to database file.

- Messages are saved to database file as an option.
- Messages are immediately sent to all available consumers when they received.
- If there are no available consumers, messages are stored. They sent right after a consumer subscribes.
- At least one consumer receives each message.

Round Robin

Round Robin works like push status. The only difference is, if there are many consumers, push status sends same message to all of them. Round robin status sends to only one of them. A message is delivered only once. Messages can be saved to database file.

- Messages are saved to database file as an option.
- Messages are immediately sent to one available consumers when they received.
- If there are no available consumers, messages are stored. They sent right after a consumer subscribes.
- Only one consumer receives same message.

Pull

In Pull status, messages are not sent to consumers when producer pushes. Consumers need to send a pull request to consume messages. Messages can be saved to database file.

- Messages are saved to database file as an option.
- Messages are stored in queue when producer pushes them. They aren't sent to consumers.
- Consumers must send a pull request to receive messages.
- Only one consumer receives same message.

Cache

Cache status works like pull status. Consumers must send a pull request to receive messages. But messages are not removed from the queue. Consumers consume same message when they request a pull. The only way to remove a message from the queue is sending new message. Queue keeps only one message in same time: The latest message. That's how cache mechanism works like, that's why the name of the status is cache. Messages can be saved to database file.

- Messages are saved to database file as an option.
- Producers can push new messages. New message removes old message in queue.
- Only one message is stored in queue. Consuming will not remove it from the queue.

Pause

Paused queues accepts new messages from producers. But a queue in paused status does not send messages to consumers. All messages are stored in it. When queue status changed, messages will be able to consume. If you want to stop consuming from a queue for a short time, but you do not want to lose any message, still accept new messages from producer, you can just pause it. Messages can be saved to database file.

- Messages are saved to database file as an option.
- All messages are stored in queue when a queue is paused.
- Producers can push messages into the queue. They will be stored.
- Consumers can't pull or receive any message from the queue.

Stop

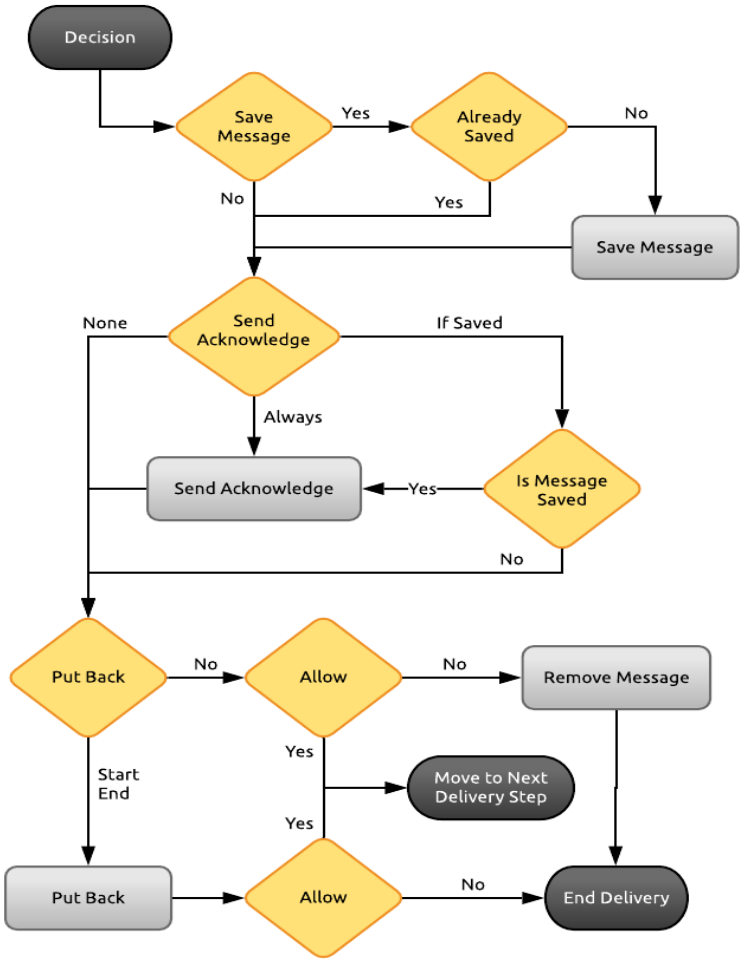
Queue is stopped. Producers can't push any messages into the queue, consumers can't consume any messages from the queue. It's possible re-run stopped queues. But when a queue is stopped, all messages in queue are removed. Only stop a queue when you need to reset it.

- All messages are removed when a queue is stopped.
- Producers can't push any message.
- Consumers can't pull or receive any message.

Decisions

Discussing about decisions right now may be early. You will understand what the Decision is really is when you see the workflow diagram in next section. But in order to understand how the delivery diagram flows, firstly, you need to know what's going on in Decision steps.

Decisions are used in nearly every step of delivery workflow. Each step has it's own decision. Each decision redirects the flow and give incredible flexibility. When a method of a step you implemented is returned the Decision object, the workflow at right side of the page is executed. Each decision object includes four properties: allow, save, put back and acknowledge.



Allow, decides if the flow can move to next step or the delivery flow is finished. The delivery workflow ends exactly when a decision's allow value is false.

Save, decides to save the message to database file. If the message is already saved, true value is ignored. If you need to save the message whenever you see and a step can't save because of some reason, you can set the save value always true in all decisions.

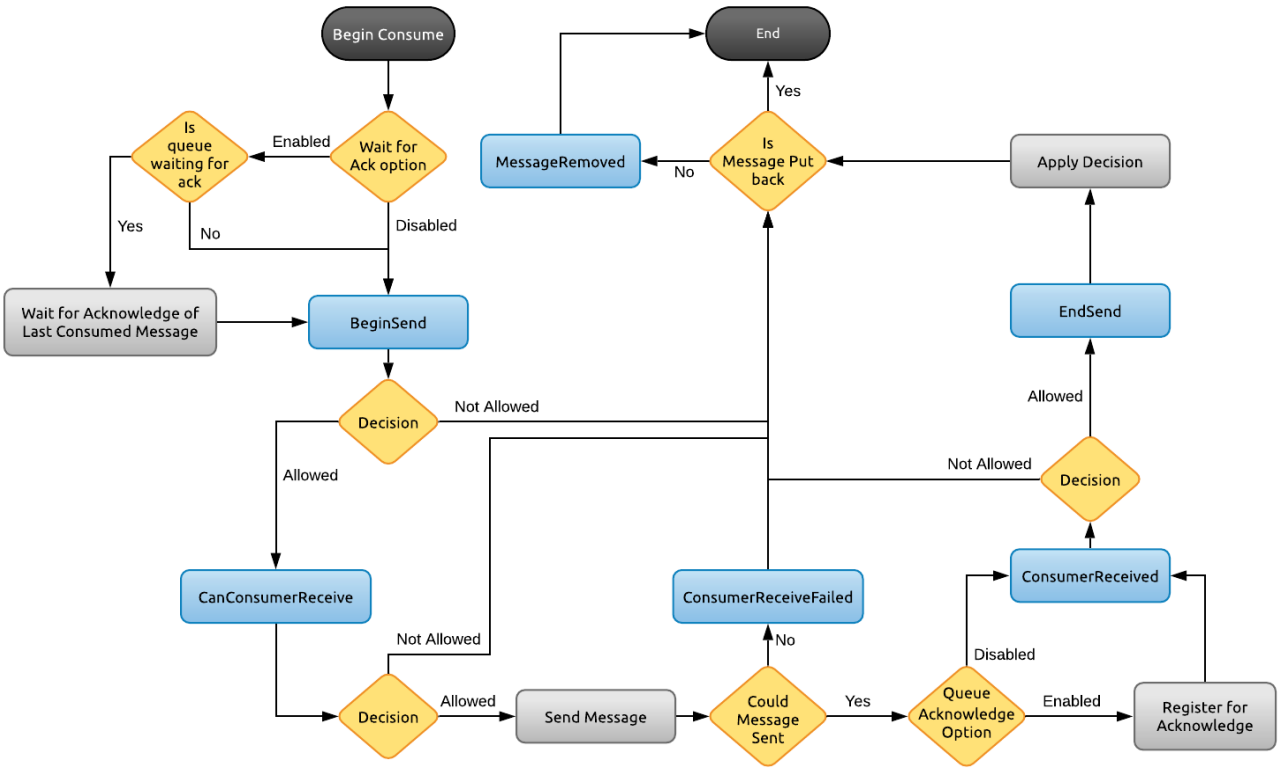
Put Back, marks the message will be put back to the queue or not. There are three choices. **No** doesn't put the message back. **Start** puts back the message at the beginning of the queue. The message will be consumed first. It can be used like immediate retry. **End** puts the message end of the queue. It will be consumed after all messages. To prevent endless put back re-consume loop, there is a put back limit. You can change this value in **ChannelQueueOptions**.

Send Acknowledge, decides when the acknowledge is sent to it's producer. **None** doesn't send. **Always** sends the acknowledge right after the decided step. **Negative**, sends negative acknowledge. **IfSaved** is sends the acknowledge right after the decided step if it's saved into database file. Send acknowledge it about the step it's decided. For example, if **IfSaved** chosen but the message is saved in next step (not in the decided current) the acknowledge will not be sent. **IfSaved** status is checked only in the step that decision made.

Delivery Workflow

When a message is received from a producer. First, it's channel and queue is found, if it's queue message. Each queue must has a delivery handler implementation. If you don't assign a specific handler to the queue, it uses channel's default delivery handler. If the channel does not have a delivery handler, it uses server's default delivery handler. If server does not have a delivery handler, an exception is thrown and application exits. So, at least one delivery handler must be implemented. Delivery handler implementation is an interface with name **IMessageDeliveryHandler**. There are several methods in it, their names describe what they do, when and why. All these methods are drawn as gray rectangles in the workflow diagram below. your determine the workflow with that implementation. **Apply Decisions** are described in previous section.

The diagram above starts after the messages is pushed into the queue and it's being started to consume to queue's consumers. That consume operation could be triggered by queue itself of a consumer might sent a pull request to **Pull** status queue. The same workflow will be executed in both situations.



The diagram does not show how acknowledge is pending, what happens when acknowledge is received or timed out or what happens when the message is timed out. It also doesn't include when a message is pushed into the queue and decision about the push operation. There are more in delivery handler implementation. Other operations don't really need a workflow diagram like this, and explained in next section.

Message Lifecycle

Each message is received from the producer and decided in delivery handler if it will be accepted for enqueue or not in **ReceivedFromProducer** method. You can accept, decline, save or delete the message in this method. In return value of the method, If it's not allowed and there is no put back decision, the message will be deleted forever.

When the message is put into the queue (assume the queue status is not Broadcast) If there is a timeout for the message in queue, message timeout handler will remove it when it's duration expired before it's consumed. If there is no timeout defined, message will be stored in the queue forever. (If it's not persistent, it will be removed when the application exits)

When the message is consumed, there is two options. First, acknowledge option is disabled for the queue. In that situation message will be removed if all Decisions have **do not put back** option. If at least one Decision have a put back decision, it will be put back into the queue.

Second, acknowledge option is enabled for the queue. Message will be **dequeued,** but it does **not** mean it's removed forever. Each queue has an acknowledge timeout duration. If an acknowledge message (positive or negative) received from the consumer, delivery handler's **AcknowledgeReceived** will be called. You will be able to reach the message within it's parameter. So if you want to put the message back, save or delete it forever you can decide in **Decision** return value of the method. If an acknowledge message is not received from the consumer, delivery handler's **AcknowledgeTimedOut** method will be called. Parameters and return type are same with **AcknowledgeReceived** method. So you can do whatever you want in timeout method.

There is a critical situation about managing acknowledge messages if you do not want to lose the message. When the message is consumed, it will be dequeued. But it will come back to you in acknowledge methods. Server can shutdown, application can crash, exit or something could happen before one of delivery handler's acknowledge methods is called. If you delete the message from persistent database when it's consumed, you could lose it forever. So you can skip deleting message from the database when it's consumed and postpone that operation to **AcknowledgeReceived** method. When your started again, the message will be loaded from the database into the queue. But this time, consumer will receive the same message again (duplication). Because it received at first, tried to send a successful acknowledge. Maybe acknowledge message is received by your TMQ Server but could not be proceed (assume shutdown between receive and process). If duplication is important, your consumer can save the last proceed message's id and check it when reconnected.

Like this, there are many situations. It depend on your architecture, system and requirements. There is no guarantee when there are three applications and talk each other over network. Whatever you use as OS, application, server or client. It's fact. In critical situation you have to decide: *possibility of the losing the data* versus *possibility to consume the same data second time*.

Routers and Bindings

A router is similar to an Exchange in AMQP Protocol. Twino Framework's main goal is the connection between applications, not only queue messages. The same goal applies to routers. A router can bind a queue or a client itself. RPC functions can be executed over routers too. Request and response mechanism can work over routers.

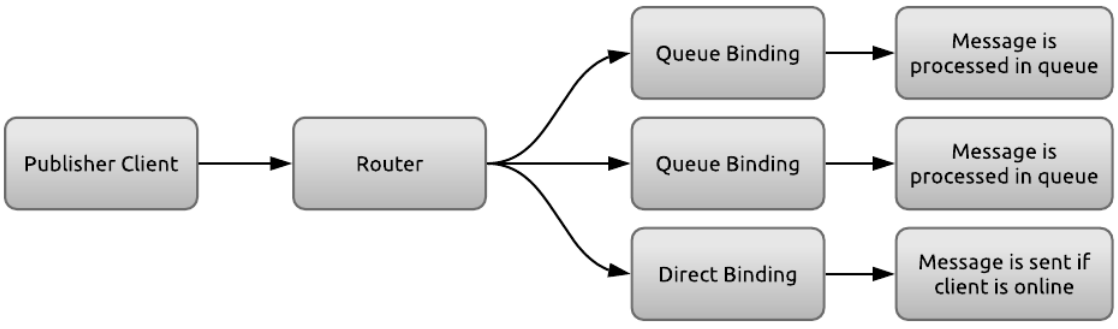
Twino has **Router** object in it. But router structure works with **IRouter** interfaces. So you can create your own routers. We will discuess Twino's default Router objects here. Each router can have unlimited bindings. **Routers** have unique names that used like their unique ids. Bindings are added routers with **Binding** objects. They have names too and used like unique ids. Twino has two kind of **Bindings** as default **QueueBinding** and **DirectBinding**. Queue bindings are for queues, direct bindings are for client bindings. Each binding has a priority. When a message is decided to be sent to only first binding, priority value is used. Highest value has the highest priority.

There are three router methods:

1. **Distribute:** Sends each message to all active bindings.
2. **Only First:** Only highest priority binding receives message. Others waits until the highest priority binding fail. If binding fails, next priority binding receives it.
3. **Round Robin:** Messages are sent to bindings with round robin algorithm. Each message is sent to only one binding.

Bindings have **Binding Interactions**; **None**, **Acknowledge** and **Response**. When a message is Published to a router, one or many of router bindings can send acknowledge or response. Binding interaction specifies that value. If interaction value is Ack, the message is sent to it's receiver with that information and tells the receiver; sender waits an acknowledge for that message.

When a message is published to a router, it's processed in multiple bindings. If a binding has an interaction (acknowledge or response) same message Id is used for that binding. If there is no binding required, new unique message is generated. So, messages might be same but message unique ids are not. You can track each message with it's unique id in queues or clients.



Acknowledge Messages

Acknowledge message are used for queue message confirmations. There are three cases that acknowledge messages are used.

1. Consumer acknowledges that the message is received. The server receives the message.
2. The server confirms the message is received from the producer and sends an acknowledge message to the producer.
3. When 1. step is occurred, if you choose to transmit that acknowledge message from the consumer to producer, server sends the message to producer.

As you see, in Twino, acknowledge messages are both acknowledges and publisher confirmations. And also, they are used for acknowledge + confirmation as third option.

As a producer, if you receive an acknowledge message, it should be sent from server or consumer. It's decided in server side in delivery handler implementation. There are many implementation methods, server chooses when the acknowledge will be sent to producer; after it received from producer, after it's sent to consumer after or acknowledge received from consumer. There are about ten options you can decide.

Acknowledge messages are processed in server when used for queue messages. If we are talking about direct messages, acknowledge messages must be sent as high priority flags. And server will not process these messages, just redirects to target clients. If you send a direct message to another client and message is waiting for acknowledge and with high priority flag, sending acknowledge is in receiver's responsibility. Server does not care what's going on between two clients. Acknowledge message type is **Acknowledge**. **Content Type** is Id value of the queue. **Target** is name of the channel. **Message Id** is same with the message which is being acknowledged. If consumer sends an acknowledge to server, server transmits the message to producer with Source value; Consumer

If an acknowledge message does not have any header data, it's successful (or positive) acknowledge. That means, receiver said everything ok about the message. If acknowledge message has a header with key **Nack-Reason**, the acknowledge message is a negative acknowledge. The value of the header tells why it's a negative acknowledge. Here is the table of values of negative acknowledge reasons:

Value	Description
none	There is no reason. It's just a negative acknowledge.
timeout	Acknowledge timed out. An example producer waits for acknowledge from server, and server waits it from consumer. But consumer doesn't send. In this case server might send a negative acknowledge when stops waiting it from consumer.

Or you can define other acknowledge reasons. Consumer sends it to server and server can transmit it to producer.

Response Messages

Response messages are used for responding a message from server or another client. Server can send response messages to clients for many operations. And clients send response messages to other clients' direct messages.

Queues messages can't be respond. They can only be acknowledged. Similar, direct messages can't be acknowledged. They can only be respond.

Response message type is Response. Target is Id of the receiver of the response message. Source is Id of the sender. If sender is server, Source will be null. Content type zero means, the message is successful. It can have a content or not. If content type greater than zero, it's used for status code. There are some predefined status codes similar to HTTP status codes. Predefined status codes are less than 1000. If you want to create your custom status codes, you should use numbers larger than 999. Otherwise, they can be processed incorrectly. Here are some status codes:

Value	Definition	Description
0	Successful	Successful response
1	Failed	Just failed, with no reason.
400	Bad Request	Request is invalid
401	Unauthorized	Client does not have permission to do that
404	Not Found	The data could not be found. (channel, queue etc)
406	Unacceptable	The request is unacceptable.
481	Already Created	The data is ready created. (channel, queue etc). Data is duplicate.
482	LimitExceeded	The limit is exceeded. Maybe client limit of channel or message limit of queue. Or something else that you used manually.
503	Busy	Process can't be done due to service is busy. Service might be server itself, or the application that received the request message.
581	SendError	The request could not sent over TCP connection.

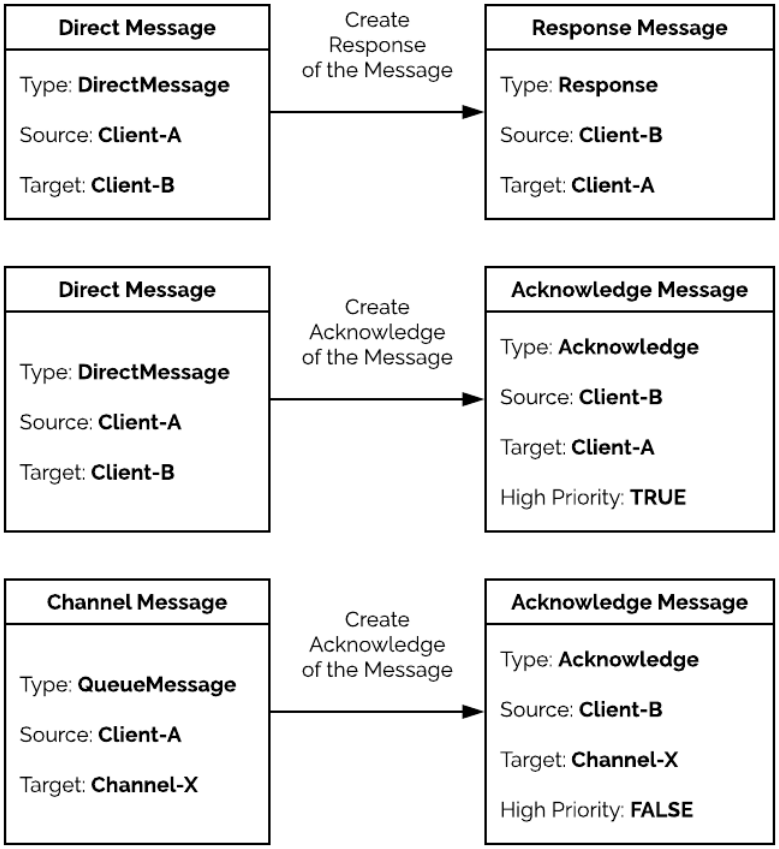
Message Source and Target

In this section, we will discuss what is for message Source and Target values. You can think, this section in continue of previous last two sections; Acknowledges and Responses.

Before talking about source and target, acknowledge and response, we need to clarify this: If a message will be respond or acknowledged it must have a unique message Id. TmqClient object creates the unique id's automatically. But if you connect and send messages to Twino MQ Server from your custom client software, you must do it manually.

Source is Id of the client who sent the message to the Twino MQ Server. Target is the name of the target of the message. If message is sent to a channel, target is the channel name. If message is sent to another client, target is Id of the client. But there are some situations such as when target is not defined well-known or queue does not allow consumers to see producer names.

For response messages, everything seems clear. Only a direct message can be respond. If a message is from A to B, response will be from B to A.



There are a few difference between acknowledge messages of direct messages and acknowledge messages of channel messages. When a client sends acknowledge for a direct message, it should know really who the sender is. If a message is sent to a target with "@type:rec-types" The message may be sent to many clients. However, acknowledge messages will be from receiver Id to sender Id.

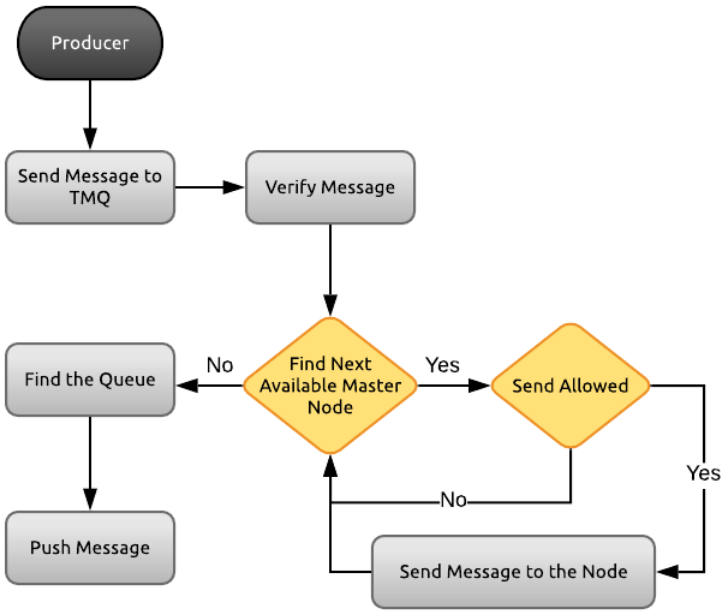
But in channel messages, consumer just sends acknowledge to a message in a queue. Maybe it's producer does not care the acknowledge, or even it's not connected. The server and queue are responsible for the acknowledge message, so the target must be the channel itself.

High Priority flag is used different for acknowledge messages. When you send an acknowledge message to server. If the message with High Priority flag, that means, the acknowledge is for a direct message. If not, the acknowledge message is for a message in a queue.

Clusters and Message Workflow

Twino MQ Server has two type of cluster node clients: **Master** and **Slaves**. Connection direction is always from slaves to masters. Master nodes accept the connections, slave nodes connect. Each node can be master or/and slave at same time.

When a message is sent to a slave node by the producer. The slave sends that message to it's master nodes. There are some filters in two side (slave and master). If the message passes these filters, it will be proceed in master node. After sending the message to it's masters, slave node process the message and puts the message into the target queue.



You can create different architecture with master and slave connections.

You can choose a leader node like Kafka. If you make this node slave and all others master, all messages is sent to other nodes.

Another example, all nodes can be slave and master. Each node connect between send their messages each other. All nodes have same queues, same messages. Consumers can connect to different nodes and they consume messages. Or another architecture that you imagine.

There is an important point about sending messages to other nodes you should know. The message, received from a producer client, is sent to master nodes. But the message received from another node isn't sent to other nodes. If it's, messages will be duplicated many times (depends on time-to-live value of the TMQ and/or TCP package)

Server Operations

Server operation messages are between servers and requester clients. Some server operation messages may require some authorization rules. It depends on your server application. In this section, we are not going to discuss about authentication or authorization.

Server Operation messages usually don't use some values in protocol frame. One Delivery, High Priority, Multiple values are ignored. Some operation messages may have Header information. All other information they need included in message contents. All server operations responses with acknowledge or response. So the request messages must have a unique message id. The acknowledge responsing messages are fail, they send negative acknowledge as response.

Channel Operations

Create New Channel

Sends create channel request to server. Target name is the name of the channel which will be created. The definition of **creating channel** is **101** content type. If message does not have a content, default options, decided by server, will be used. The operation sends acknowledge as response. You can send **ChannelOptions** object as JSON format in content to create channel with custom options.

Message Type	Content Type	Target	Wait For
Server Operation	101	[Channel Name]	Acknowledge

Delete Existing Channel

Target name is the name of the channel which will be updated. Message content is ignored. The operation sends acknowledge as response.

Message Type	Content Type	Target	Wait For
Server Operation	103	[Channel Name]	Acknowledge

Get Channel List

Gets active channels in the server. If target is null, all channels will be returned. You can filter channels with target value. Filtering supports * character. The operation sends response as response.

Message Type	Content Type	Target	Wait For
Server Operation	104	NULL or channel name filter	Response

Get Channel Information

Gets detailed channel information. Target must be the target channel name. The operation sends response as response.

Message Type	Content Type	Target	Wait For
Server Operation	105	[Channel Name]	Response

Join to a Channel

Joins to a channel. Target is channel name. Message has no content. The operation sends acknowledge as response.

Message Type	Content Type	Target	Wait For
Server Operation	110	[Channel Name]	Acknowledge

Leave From a Channel

Leaves from a channel. Target is channel name. Message has no content. The operation sends acknowledge as response.

Message Type	Content Type	Target	Wait For
Server Operation	111	[Channel Name]	Acknowledge

Queue Operations

Create New Queue

Sends create new queue in a channel request to server. Target name is the name of the channel that queue will be created in. Queue Id must be send with **Queue-Id** key in header. If message does not have a content, default options, decided by server, will be used. The operation sends acknowledge as response. You can send **ChannelQueueOptions** object as JSON format in content to create queue with custom options. Default options will be used for missing properties of the object.

Message Type	Content Type	Target	Wait For
Server Operation	201	[Channel Name]	Acknowledge

Header Key is **Queue-Id** for queue content type.

Update Queue Options

Updates queue options in a channel. Target name is the name of the channel of the queue. Queue Id must be send with **Queue-Id** key in header. Message must has a content. Content must be JSON format of **ChannelQueueOptions** object. Default options will be used for missing properties of the object. The operation sends acknowledge as response.

Message Type	Content Type	Target	Wait For
Server Operation	202	[Channel Name]	Acknowledge

Header Key is **Queue-Id** for queue content type.

Delete Existing Queue

Target name is the name of the channel. Queue Id must be send with **Queue-Id** key in header. Message content is ignored. The operation sends acknowledge as response.

Message Type	Content Type	Target	Wait For
Server Operation	203	[Channel Name]	Acknowledge

Header Key is **Queue-Id** for queue content type.

Get Queue List of a Channel

Target name is the name of the channel. The operation sends response as response. Response is an array of **QueueSummary** object in JSON format.

Message Type	Content Type	Target	Wait For
Server Operation	204	[Channel Name]	Response

Get Queue Information

Target name is the name of the channel. Queue Id must be send with **Queue-Id** key in header. The operation sends response as response. Response is **QueueInformation** object in JSON format.

Message Type	Content Type	Target	Wait For
Server Operation	205	[Channel Name]	Response

Header Key is **Queue-Id** for queue content type.

Connection Operations

Get Online Nodes

That request should require administration authorization. Target name is empty. Response is an array of **NodeInformation** object in JSON format.

Message Type	Content Type	Target	Wait For
Server Operation	301	NULL	Response

Get Online Clients

That request should require administration authorization. If target name is empty, returns all clieents. If there is a target name, only clients with that **Client-Type** value will return. Response is an array of **ClientInformation** object in JSON format.

Message Type	Content Type	Target	Wait For
Server Operation	302	NULL or Client-Type filter	Response

Queues and Messaging

Pushing Messages Into Queues

Pushing a message into a single queue is simple. Message Type is Queue Message, Target is channel name, Content Type is Queue Id. Message content is message itself. If producers wants to receive acknowledge message for the message, producer should specify it in protocol frame. If acknowledge is pending, server will reply as acknowledge depend on decision. (There are many ack decisions, when, how, why)

Message Type	Content Type	Target	Wait For
Queue Message	Queue Id	Channel Name	None / Acknowledge

If a message should sent to multiple queues, message will be copied for each. Each copy must have a unique id. Original message id will stored for target channel and queue. For cc queues, message id values can be specified, or they will be generated. If producer let server to generate a message id for the copy, producer can't know what the generated value is. A header key – value should be added for each copy. Key is **cc** (like sending e-mail) value is consists of two or three words seperated with **";"**

If Queue Id is not specified, same Queue Id will be used for cc channels. If Message Id is not specified, new unique message id will be generated. Format is **Channel-Name ;Queue-Id;Message-Id**. Here are some cc examples:

Key	Value
cc	DemoChannel ;1001;ehfx-2893-k2xe-9x01
cc	OtherChannel;758
cc	AnotherChannel

Pulling Messages From Cache Queues

Messages are not consumed automatically by their consumers when the queue is pull or cache status. In order to consume a message, consumer must send a pull request to the server. If the queue is in Cache status, server sends a response message to pull request message. Response message content type is not Queue Id. It will be zero if the pull request is successful and there are incoming messages. You see content type list in **Acknowledge and Response Messages** section.

Message Type	Content Type	Target	Wait For
Queue Pull Request	Queue Id	Channel Name	Response

Pulling Messages From Pull Queues

Pull queues works similar to cache queues. But there are some concept changes. Cache queues are for just caching the same message. Their message Id values are not important. The key is calling a method and receiving the message. Response message type is the best way to do this.

But Push status concept has some differences. First, you might need unique message Id values for each message. So the message must cannot be Response. Second, you might send a pull request for multiple messages and expecting more than one message. Each message content might be in megabytes. You should not wait to download all messages to start to process them. You need to receive one message, while you are processing it, other is being downloaded.

These requirements and concept changes forces pull operation to be fully asynchronous. Because of these changes, pulled messages have their unique Id values, their types are Queue Message. They are tracked by header key and values. And even there is no message in the queue, or pulling messages from the queue is not authorized, there will be at least one Queue Message received. With no content and a description about it. We will discuss receiving Queue Messages with their all header values in next section. Now let's look which header keys can be attached to pull request message.

Key	Value
Count	Count of the maximum messages that are pulled. If queue has that count of message, they will be sent as count. If queue has less messages than count, all messages are consumed. Each message will have a header data about index and count.
Clear	Clear is used when you want to remove all left messages in queue after pull operation. There are 4 possible values: 1. None, does nothing. It's default option. 2. All, clears all left messages in queue after response sent. 3. High-Priority, clears all high priority messages in queue. 4. Default-Priority, clears all default priority messages in queue.
Order	Default is FIFO. In a queue first in first outs. If you want to pull messages as LIFO, you can send this value: **LIFO**
Info	There are 2 possible values: **yes** and **no**. Default option is no. If yes, each message will have a header value that includes left message count values in queue. Header keys are **Priority-Messages** and **Messages**

Receiving Messages From Queues

Messages are consumed automatically if queue status is broadcast, push or round robin. When a message is sent to it's consumer; target is channel name, content type is queue id and message content is the message itself. If queue requests acknowledge, message should be acknowledged. That kind of queue messages don't have header data if you don't add any header in server-side.

Message Type	Content Type	Target	Wait For
Queue Message	Queue Id	Channel Name	None / Acknowledge

But messages have some header data if messages are sent after a pull request to a pull queue. Each pull request has it's count value. That values must be tracked by the client. If client requested 10 messages but the queue has 8 messages, 8 messages will be sent. Each message will have it's index and count such as 1/8, 2/8 ... 8/8.

And sometimes, queue might be empty. Or pull request might be unauthorized or maybe queue was deleted. To answer the pull request, a message will be sent with No-Content information.

Key	Value
Request-Id	Message Id value of the pull request.
Index	One-based index number for the message. Index of the last message equals to count.
Count	Total messages that are ready to sent to consumer. These messages are dequeued and from the queue after the pull request. Other clients can't consume them.
Priority-Messages	If info requested, left high priority message count of the queue.
Messages	If info requested, left message count of the queue.
No-Content	If a message has that header, it means, the message is not a real message. It's just a information about the pull request. Here are some used values for this header: • **Empty:** The queue is empty, there is no message to pull. • **Unacceptable:** The request is not valid. • **Unauthorized:** Client does not have permission to pull. • **No-Channel:** Channel not found. • **No-Queue:** Queue not found. • **Id-Required:** Pull request must include a unique message Id. • **End:** Always last message of the pull request. It's a signal for the client, tells the pull request operation is completed.

Send Direct Message

Direct messages are sent directly clients. Here are limitations and features of direct messages:

- Direct messages are able to sent from producer to producer, from consumer to consumer, from producer to consumer and from consumer to producer.
- Direct messages are able to sent by client id, client name or client type.
- It's possible to send a direct message multiple clients with same name or same type.
- It's possible to send a direct message to only one receiver when there are multiple available clients with same name or type. Always the oldest (first connected) client receives the message.
- Direct messages can be acknowledged or respond.
- Content Type is not used by server in direct messages. Sender and receiver and choose a content type to specify their message type.

Message Type	Content Type	Target	Wait For
Direct Message	User's Choice	There are 3 choices: 1. Receiver Client-Id 2. @name:"Client-Name" 3. @type:"Client-Type"	None Acknowledge Response

Direct messages must have a unique Message Id. Response will be sent to that Id. Response messages are explained in next section.

Receive Direct Message Response

Response messages have Response message type. Their target is Id of another message (request message). Content type value is zero if the response is successful. Response messages can wait for acknowledge or response.

Message Type	Content Type	Target	Wait For
Response	=0 Successful >0 Failed	Message Id of Request	None Acknowledge Response

Receive Acknowledge Message

Acknowledge messages are same with response messages. The only difference is Message Type. Acknowledge messages message type is Acknowledge. All other options, parameters and usages are same with response messages.

Events

Events are used for monitoring Twino MQ Server and getting notified when something happens. Events have two type of message structure. If an event message is sent to the server from a client, that means, client subscribes or unsubscribes to the event. If an event message is sent to client from the server, that means, something happened and server sends a notify message to the client.

Subscribe and Unsubscribe

In order to subscribe or unsubscribe to event, event name must be sent to target name. Subscribe and unsubscribe difference is in Content Type.

Message Type	Content Type	Target	Wait For
Event	1 Subscribe 0 Unsubscribe	Name of the event	Acknowledge

If the event is about server itself such as client connected, node disconnected, the subscription message does not require more data in it. But if subscription is to a specific channel and a specific queue, for example when a message is produced into a queue. The subscription message requires a channel name and a queue id.

There are some channel and queue events. If an event requires a channel **Channel-Name** header key is used. If an event requires a queue too, in addition, **Queue-Id** header key is used. These keys are specifies the channel and queue of the event.

Receive Messages of Events

When something happened about an event, server sends all event subscribes that information. This message is sent as Event Message Type. Target is the name of the event. Content Type in unused.

Message Type	Source	Content Type	Target	Wait For
Event	Event Name	Queue Id (if exists)	Channel name (If exists)	None

Some event messages requires channel and queue information like subscription and unsubscription messages. An example, if a message is produced in **DemoChannel** in a queue with **1500** Id value. The event message includes two headers. First key is **Channel-Name** and value is **DemoChannel**. Second key is **Queue-Id** and value is **1500**.

Event Types

All events have a name and an event model. Models are serialized into contents as JSON strings.

Event Name	Object Type	Description
ClientConnected	ClientEvent	When a client connected to Twino MQ Server.
ClientDisconnected	ClientEvent	When a client disconnected from Twino MQ Server.
ClientJoined	SubscriptionEvent	When a client joined to a channel.
ClientLeft	SubscriptionEvent	When a client left a channel.
ChannelCreated	ChannelEvent	When a channel is created.
ChannelRemoved	ChannelEvent	When a channel and it's queues are removed.
QueueCreated	QueueEvent	When a queue is created in a channel.
QueueUpdated	QueueEvent	When options of a queue is changed.
QueueRemoved	QueueEvent	When a queue and it's messages are removed.
MessageProduced	MessageEvent	When a message is produced into a queue.

Client Event

Client Event model is used for **ClientConnected** and **ClientDisconnected** events.

Name	Type	Description
Id	String	Client Unique Id
Name	String	Client Name
Type	String	Client Type

Channel Event

Client Event model is used for **ChannelCreated** and **ChannelRemoved** events.

Name	Type	Description
Name	String	Channel Name
ActiveClient	Integer	Active clients in channel
ClientLimit	Integer	Channel's maximum client limit

Subscription Event

Subscription Event model is used for **ClientJoined** and **ClientLeft** events.

Name	Type	Description
Channel	String	Channel name
ClientId	String	Client Unique Id
ClientName	String	Client Name
ClientType	String	Client Type

Queue Event

Queue Event model is used for **QueueCreated**, **QueueUpdated** and **QueueRemoved** events.

Name	Type	Description
Channel	String	Channel name
Id	UInt16	Queue Id
Tag	String	Queue Tag
Status	String	Queue Status
PriorityMessages	Integer	High priority messages count in queue
Messages	Integer	Messages count in queue (doesn't include priority messages)

Message Event

Message Event model is used for **MessageProduced** event.

Name	Type	Description
Channel	String	Channel name
Queue	UInt16	Queue Id
Id	String	Message Id
Saved	Boolean	True, if message is saved to disk
ProducerId	String	Message Producer's Unique Client Id
ProducerName	String	Message Producer's Client Name
ProducerType	String	Message Producer's Client Type

Models

Documentation has some models for some operations. For example, updating channel options operation requires ChannelOptions object in message content. These objects are sent and received as JSON serialization format. They are explained in this section.

Channel Queue Options

Name	Type	Description
SendOnlyFirstAcquirer	Boolean	If true, message is sent to only one consumer: first subscribed acquirer. Even queue status is broadcast or push.
RequestAcknowledge	Boolean	Each messages are sent as acknowledge pending. They acknowledge messages are not respond, ack timeout operation will be executed.
AcknowledgeTimeout	Integer	In Milliseconds.
MessageTimeout	Integer	In Milliseconds. Starts when message is produced. It is reset when application restart.
UseMessageId	Boolean	If true, each message has it's unique id. Even it's producers does not send, server creates and assigns.
WaitForAcknowledge	Boolean	If true, after sending a message to it's consumer, queue stops to sending next message to any consumer. It waits the acknowledge (or timeout) for the message.
HideClientNames	Boolean	If true, source values of messages are hidden. Consumers can't see which producer pushed them.
Status	Integer	Queue status as integer value. 0 Broadcast, 1 Push, 2 RoundRobin, 3 Pull, 4 Cache,, 5 Paused, 6 Stopped.
MessageDeliveryHandler	String	The key for delivery handler. If server registers some delivery handler object types with unique keys. Clients can change handlers remotely.
MessageLimit	Integer	Maximum messages in a queue. If exceeds, server rejects push requests.

Channel Options

Channel options includes all properties of Channel Queue Options. When a new queue is created in a channel, if no options are set, channel's default options will be used. Because of this, each channel options should include channel queue options in it. Properties of Channel Queue Options are not listed below.

Name	Type	Description
AllowMultipleQueues	Boolean	If true, a channel can has multiple queues. False, only one queue can be defined in a channel.
AllowedQueues	UInt16 Array	If not null, list of allowed id list for queues.
EventHandler	String	The key for event handler. If server registers some event handler object types with unique keys. Clients can change handlers remotely.
Authenticator	String	The key for authenticator. If server registers some authenticator object types with unique keys. Clients can change handlers remotely.
ClientLimit	Integer	Maximum client limit for the channel. If exceeds, subscription requests are rejected. 0 is unlimited.
QueueLimit	Integer	Maximum queue limit for the channel. If exceeds, queue create requests are rejected. 0 is unlimited.
DestroyWhenEmpty	Boolean	If true, queue is destroyed when it has no consumer and no message.

Channel Information

Name	Type	Description
Name	String	Channel name
Queues	UInt16 Array	Active queues in the channel
AllowMultipleQueues	Boolean	If true, channel can have multiple queues with multiple content type. If false, each channel can only have one queue.
AllowedQueues	UInt16 Array	Null or empty value means all queues are allowed. Allowed queue id list for the channel. This property doesn't mean the all queues are created and active.
OnlyFirstAcquirer	Boolean	If true, messages will send to only first acquirers.

RequestAcknowledge	Boolean	If true, messages will request acknowledge from receivers.
AcknowledgeTimeout	Integer	When acknowledge is required, maximum duration for waiting acknowledge message. In milliseconds.
MessageTimeout	Integer	When message queuing is active, maximum time for a message wait. In milliseconds.
UseMessageId	Boolean	If true, server creates unique id for each message.
WaitForAcknowledge	Boolean	If true, queue does not send next message to receivers until acknowledge message received.
HideClientNames	Boolean	If true, server doesn't send client name to receivers in queueus.
ActiveClients	Integer	Online and subscribed clients in the channel.
ClientLimit	Integer	Maximum client limit of the channel. Zero is unlimited.
QueueLimit	Integer	Maximum queue limit of the channel. Zero is unlimited.
DestroyWhenEmpty	Boolean	If true, channel will be destroyed when there are no messages in queues and there are no consumers available.

Queue Information

Name	Type	Description
Id	UInt16	Queue id
Channel	String	Queue channel name
PriorityMessages	Integer	Pending high priority messages in the queue
Messages	Integer	Pending regular messages in the queue
Status	String	Current status of the queue
OnlyFirstAcquirer	Boolean	If true, messages will send to only first acquirers.
RequestAcknowledge	Boolean	If true, messages will request acknowledge from receivers.
AcknowledgeTimeout	Integer	When acknowledge is required, maximum duration for waiting acknowledge message. In milliseconds.
MessageTimeout	Integer	When message queuing is active, maximum time for a message wait. In milliseconds.
UseMessageId	Boolean	If true, server creates unique id for each message.
WaitForAcknowledge	Boolean	If true, queue does not send next message to receivers until acknowledge message received.

HideClientNames	Boolean	If true, server doesn't send client name to receivers in queueus.
ReceivedMessages	Integer	Total messages received from producers.
SentMessages	Integer	Total messages sent to consumers.
Deliveries	Integer	Total message send operation each message to each consumer.
NegativeAcks	Integer	Total negative acknowledged or not acknowledged messages.
Acks	Integer	Total acknowledged messages.
TimeoutMessages	Integer	Total timed out messages.
SavedMessages	Integer	Total saved messages.
RemovedMessages	Integer	Total removed messages.
Errors	Integer	Total error count.
LastMessageReceived	Int64	Last message receive date in UNIX milliseconds
LastMessageSent	Int64	Last message send date in UNIX milliseconds
MessageLimit	Integer	Maximum message limit of the queue. Zero is unlimited.
MessageSizeLimit	Integer	Maximum message size limit. Zero is unlimited.

Node Information

Name	Type	Description
Id	String	Node Instance Unique Id
Name	String	Node Name
Host	String	Node Hostname
Slave	Boolean	True, if node is slave.
Connected	Boolean	True, if node is connected.
Lifetime	Int64	Node connection lifetime in UNIX milliseconds.

Client Information

Name	Type	Description
Id	String	Client's unique Id
Name	String	Client name. It’s not unique.
Type	String	Client type. If different type of clients join your server, you can categorize them with this type value.
Online	Int64	Total online duration of client in milliseconds
IsAuthenticated	Boolean	If true, client authenticated by server's IClientAuthenticator implementation