A High-Throughput Low-Latency Queuing Streaming API to Complement OpenStack Zaqar REST Interface

**By:** Jim Hoagland, Keith Newstadt, Karthik Karuppaiya ([cpe.symantec.com](http://cpe.symantec.com))

**Status:** work in progress

**Last Modified:** 2014-09-25

This document describes a streaming API (really more like a protocol) to complement the OpenStack Zaqar (Marconi) REST interface. This allows messages to be stored or received in queues with high-throughput and low-latency.

# Streaming Interface

[need a URI]

A WebSocket connection is initiated by the client and is set up in accordance with RFC 6455. The resulting WebSocket connection can be used for sending messages to different queues and for retrieving messages from different queues, though at a given time only type of operation is selected and only one queue is involved.

Sec-WebSocket-Protocol=TBD /\* TODO, supposed to eventually registered with IANA \*/

[should we have a distinct wsmessages separate connection for each queue to produce to or receive from as seems to be the case in Narconi? does this scale as # queues grows? Does the HTTP limit of 2 connections apply? the multiplex WebSocket extension will help, but that internet draft expired]

In the interest of efficiency, the protocol between the client and server is binary and optimized for space and parsing. To that end we use Google Protocol Buffers (protobufs) to represent the set of fields we need to communicate between the client and the server. Protobufs consist of possible nested structures called messages; to reduce confusion with “message” elsewhere in the this doc, we’ll refer to these as *pbmessages*.

The X-Auth-Token header is used for authentication for the streaming interface in the same way as it is used for OpenStack REST APIs such as Zaqar’s REST interface.  The header appears in the WebSocket protocol setup.

## Common elements of producing and consuming

A client that wants to get or post messages to a queue initiates a client request. Client requests are contained in a single binary WebSocket message (hereafter called *wsmessage* to reduce confusion with other uses of the word message), as are the corresponding responses. A binary request wsmessage starts with of a setup request and the response wsmessage from the server starts with a response to the setup request.

### Delimiting pbmessage boundaries

If you put two pbmessages adjacent to each other, you can’t reliably distinguish where one pbmessage ends and the other begins. For this reason, we need some framing to provide this. We have chosen length prefixing of pbmessages for this purpose. Each top-level pbmessage send by the client or server is prefixed with some number of bytes in which the integer length is represented. We chose to encode the length in an identical way to how Google protobuf encodes a uint32 in the wire format, as a [base 128 varint](https://developers.google.com/protocol-buffers/docs/encoding#varints). For example a 64 byte pbmessage would be prefixed with 0x40, a 16383 byte pbmessage would b prefixed with 0xFF 0x7F, and a 1,000,000 byte pbmessage would be prefixed with 0xC0 0x84 0x3D.

Note that this puts a cap of 2^32 -1 on the size of a protobuf, which seems reasonable given resource constraints that are likely to be in place anyway (you aren’t going to want to send a divisible 4GB set of data in one shot and you would need to have enough memory on both the client and server side to work with that size protobuf).

### Status pbmessage

The status pbmessages are used in some server responses and is equivalent to a HTTP status line except that the reason phrase is optional. Response codes have the same meaning as in HTTP. This is the pbmessage:

// represents the status of the server in the same way as HTTP but with the reason phrase being optional

message Status {

// status\_code: same as status code in RFC 2616 (required)

optional uint32 status\_code = 1;

// reason\_phrase: reason for the status (optional)

optional string reason\_phrase = 2;

}

### Requested operation

We currently have two operations that the client can make: get and append. Which operation is requested is indicated at the start of every new client to server wsmessage; here is the protobuf that is the first protobuf from the client to a server in a wsmessage:

// a setup request from the consumer; could be either for a get operation or an append operation

message SetupRequest {

/// consuming direction

// the get field is present if and only if a get request is being requested

optional ConsumeSetupRequest get = 1;

/// producing direction

// the get field is present if and only if a get request is being requested

optional ProduceSetupRequest append = 8;

// exactly one of the above fields should be present; instead of having an operation type enum, we can tell the operation type being requested by which field is present

// (as it turns out bits 2-5 of the first byte of the protobuf essentially act as an operation code)

}

## Message numbers

When a client is operating as a producer, messages are streaming from the client to the server and when a client is operating as a consumer, messages are streaming from the server to the client.

In both directions, there is an implicit numbering of messages that have been sent or that are anticipated within a wsmessage. The message numbers start with 1. In the (quite remarkable) case that 263 -1 messages are sent in a wsmessage, a new wsmessage will need to started; this allows us to safely use a signed 64 bit int to represent the message number.

This numbering is only used within the protocol itself.

## Message flow control

The rate at which a message sender can sustainably send messages is limited by the rate at which the message recipient can retrieve the messages and do whatever it needs to do with it (store it and/or process it). To try to prevent the sender from sending messages that are likely to get dropped due to limited buffers, the recipient explicitly communicates to the sender how many additional messages it is currently willing to accept. This is communicated in a way that is clear and doesn’t require additional synchronization by the recipient indicating the highest [message number](#h.pmfjo6d66tph) that it is willing to receive. The sender must not send a message that would be numbered higher than the most recently received accepted message number from the receiving side.

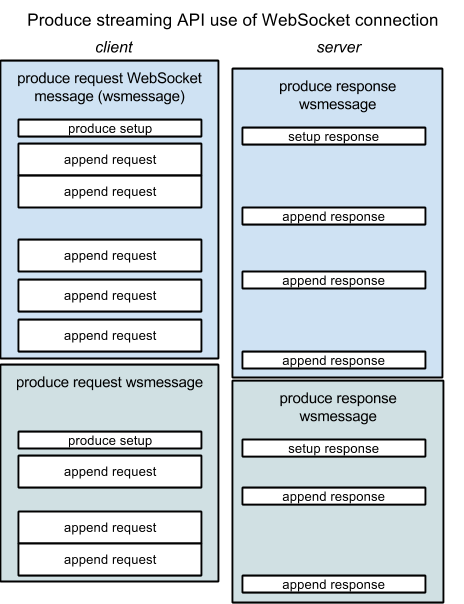
~~To allow a faster start to streaming for a new wsmessage, the sender can assume that the upon starting that the recipient is always willing to accept up at least to message number 100. This way 100 messages can be sent right away.~~

## Interface for producing messages using streaming interface

This interface describes how messages may be be sent, over time, to the queuing interface for addition to a queue. A single wsmessage sends messages to a particular queue and a single wsmessage is used for the corresponding responses. WebSocket frames allow for chunks of the wsmessage to be sent when available and needed. Communication in both directions consists of sequence of length-prefixed protobuf messages.

We are optimized for the cases where there is a sequence of messages going to the same queue; to start receiving from another queue any current wsmessage will need to be closed out and a new wsmessage would be need to be initiated.

Here is an overview of producing:



### Produce operation setup

Produce operations are initiated by the client sending a SetupRequest pbmessage containing an populated append field in a new wsmessage. The append field contains a ProduceSetupRequest pbmessage, defined as follows:

message ProduceSetupRequest {

// queue\_name: the name of the queue to add the messages to (required)

optional string queue\_name = 1;

}

The queue name is the name of the queue to produce to. Upon receiving the produce setup request, the interface will check if the user (as determined by keystone) is authorized to receive from the requested queue.

The server responds by starting a new wsmessage with a ProduceSetupResponse pbmessage:

message ProduceSetupResponse {

// queue\_name: the name of the queue to add the messages to (required)

optional Status status = 1;

// max\_msgnum\_to\_send: the initial highest numbered message that the server wants sent to it (required)

optional int64 max\_msgnum\_to\_send = 3;

}

The server will respond with [status](#h.feaqxoe2cnk9) of 100 (continue) if client is cleared to proceed. If the client is allowed to send message, and appropriate status code will be sent and the max\_msgnum\_to\_send will be set to 0.

### Produce requests and responses

After the produce direction is setup, messages to add to the queue are sent by the client in a ProduceRequest, which provides the payload of the message and the message TTL.

message ProduceRequest {

// ttl: the TTL for all the messages in this ProduceRequest (required)

optional uint32 ttl = 1;

// payloads: one or more message payloads (raw bytes) to add

repeated bytes payloads = 2;

}

The number of payloads in the ProduceRequest pbmessage is limited only by the rule of not exceeding the server’s communicated max\_msgnum\_to\_send.

Responses to client either consist of a max\_msgnum\_to\_send or a max\_msgnum\_to\_send with a report of the status of messages that were received by the client. In both cases a ProduceResponse pbmessage is used:

message ProduceResponse {

// status: the produce status being conveyed (optional; default is that no status is being conveyed in this protobuf message)

optional Status status = 1;

// msgnum\_status\_is\_thru: the above status is for this message number as well as any messages numbered prior to this but after the message number the previously reported status was for (required if status is present)

optional int64 msgnum\_status\_is\_thru = 2;

// max\_msgnum\_to\_send: update or re-affirm the highest numbered message that the server wants sent to it (required)

optional int64 max\_msgnum\_to\_send = 3;

}

These responses are not necessarily direct reactions to a particular ProduceRequest, but the server should send a status message shortly after it is known that a set of messages has been added to a queue or not (the client may be holding a copy of the messages until it hears the status). In addition, the server should give a reasonably timely notification to the client if it is able to accept additional messages beyond what was last communicated because otherwise it that could cause the client to delay sending additional messages, which may cause it to unnecessarily fall behind.

If there is a contiguous series of messages with the same status, a single ProduceResponse may communicate the status of all the messages. This is done by setting msgnum\_status\_is\_thru to the last message number in that range. The start of the range is implicitly the msgnum\_status\_is\_thru from the previously send ProduceResponse plus one.

In case of a status indicating an error, if the wsmessage or WebSocket connection is terminated at this point, the error should be considered fatal from the point of view of this production to the queue.

### Closing a produce operation

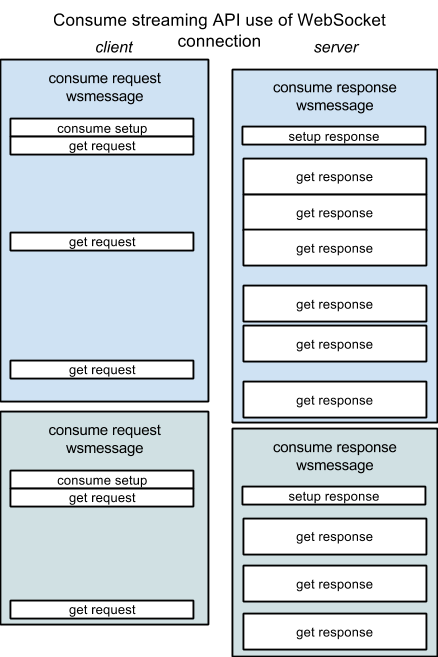
To close a streaming produce request, the client simply closes out the wsmessage. The server will finish responding to received messages then close its wsmessage. The server may close a wsmessage itself in which case the client would similarly close its wsmessage.

## Interface for consuming messages using streaming interface

This interface describes how messages may be retrieved from the queuing interface over time. A single wsmessage is used for requesting messages for a particular queue and a corresponding single wsmessage is used for the corresponding responses. WebSocket frames allow for chunks of the wsmessage to be sent when available and needed. Communication in both directions consists of sequence of length-prefixed protobuf messages.

We are optimized for the cases where there is a sequence of messages being received from the same queue; to start receiving from another queue any current wsmessage will need to be closed out and a new wsmessage would be need to be initiated.

Here is an overview of consuming:



### Message metadata

Certain metadata is made available to the consumer about messages they are receiving. The metadata is identical to the GET message response metadata in the REST API. Specifically::

* message ID (unique identifier for a message): always provided
* TTL (corresponds to TTL provided by producer): always provided
* age (time since message was add to queue): always provided
* message marker (position in the queue): available upon request
* claim ID and claim client ID: always provided if relevant

### Consume operation setup

The client sends a consume setup request in a new wsmessage and the server responds in a new wsmessage, starting with a consume setup response. This is the corresponding request pbmessage:

// a consumer setup request

message ConsumeSetupRequest {

// queue\_name: the name of the queue to get message from (required)

optional string queue\_name = 1;

// echo\_requested: same as echo in REST API (required)

optional bool echo\_requested = 2;

// include\_claimed: same as include\_claimed in REST API (required)

optional bool include\_claimed = 3;

// starting\_marker: same as marker in REST API -- starting marker for consuming (required)

optional string starting\_marker = 4;

}

This requests messages from a queue and configures which ones it will receive. Here, echo requested, include\_claimed, and starting marker have the same meaning as in the Marconi REST interface.

The produce setup response consists simply of a Status pbmessage as defined above.

Upon receiving the consume setup request, the interface will check if the user (as determined by keystone) is authorized to receive from the requested queue. The server will respond with 100 (continue) if client is cleared to proceed. In the interest of reduced latency upon startup, the client need not wait for this response to start with consume requests.

### Consume requests and responses

After sending the consume setup request, the client sends a get request to ask for some messages to be send and thereafter occasionally sends additional consume requests. The get request is both used for flow control and to request that a marker be sent.

This is the pbmessage for a consume request:

// a consume response; used for flow control and for setting options for the retrieved messages that the server provides

message ConsumeRequest {

/// setting options to server

// send\_marker: if true, requests that the server provide a marker for a message, either for the most recently sent message or for the next one to be sent (optional, defaults to false)

optional bool send\_marker = 1 [default=false];

/// flow control

// allowed\_message\_num: update or re-affirm the highest numbered message that we want sent to us (required)

optional int64 allowed\_message\_num = 2;

}

Here, “send a marker” requests that the server send a message ID and its corresponding message marker. This can be either the most recent message it send or the next message it is going to send. In the later case, the marker is added as an additional field to the returned message. In the former case, just the message ID and marker would be sent (example use case: the client is going to close consuming from the queue and wants to know the marker location so it can resume from it later).

The allowed\_message\_num field is used to tell the server how many messages it is ready to receive as described in the [Message flow control](#h.uk70bao2l5w8) section.

These are the pbmessages for a consumer response:

// represents a claim (same as in Marconi REST API)

message Claim {

// id: claim ID (required)

optional string id = 1;

// client id: claim client ID (required)

optional string client\_id = 2;

}

// a message with the metadata that is available when sending the message from a queue to a client

message MessageAndMetadata {

// marker: marker indicating the position of this message in the queue (optional; may be omitted if not requested)

optional string marker = 1;

// id: message ID of this message (required)

optional string id = 2;

// ttl: message TTL (required, if a payload is being provided)

optional uint32 ttl = 3;

// age: current age of message (required, if a payload is being provided)

optional uint32 age = 4;

// claim: associate claim (optional; if a payload is being provided then omission indicates there is no associated claim)

optional Claim claim = 5;

// payload: the message payload as raw bytes (required if a payload is being provided)

optional bytes payload = 6;

}

// a consume response, either conveying a list of messages with associate metadata or an error

message ConsumeResponse {

// status: if there was an error encountered on the server side, conveys some details (optional; omission means there is no error to report)

optional Status status = 1;

// messages: one or more messages and associated metadata (optional)

repeated MessageAndMetadata messages = 2;

// either we should have a status or have at least one message in messages

}

In the case of an error on the server, the response is simply a Status pbmessage indicating the error. In the normal case, the pbmessage can contain one or more message with metadata structures.

You can see that when there is no message payload included but marker is present, then much of the fields are omitted; you just have the message marker and message ID.

You must have a marker, a message body, or both.

### Closing a consume operation

To close a streaming consume request, the client simply closes out the wsmessage. The server will shortly after cease sending then close its message. The server may close a message it message itself in which case the client would similarly close its message.