Design notes for PoC implementation of streaming API server for Queuing Streaming Interface

This document captures some notes related to the Python implementation of streaming API server for Queuing Streaming Interface.

Table of Contents

Design notes for PoC implementation of streaming API server for Queuing Streaming Interface 1

Libraries that could be used 1

WebSocket Library 1

Event framework 2

ProtoBuf Library 2

Low Level Design 2

Local State 2

Code flow 3

Interface routines 3

Interface routines (for consume direction only) 5

Interface routines (for produce direction only) 6

Setup/teardown for queue operation 6

Requests (client to server) 7

Other events 7

Classes 7

ProtoBufStream 7

ProtoBufStream implementation suggested changes 8

Future work 8

# Libraries that could be used for server implementation

## WebSocket Library

After reviewing the Python WebSocket libraries listed in “Notes from Research Into Queuing Protocols for Symantec CPE Stream Processing Solution“, [Autobahn|Python](http://autobahn.ws/python/) was the only one with clear support for a frame-level API. It is also reportedly high performance.

Stand-alone server script available. Other options discussed here: <https://groups.google.com/forum/#!topic/autobahnws/djyQVnFxRPA>

## Event framework

<http://autobahn.ws/python/>

[Autobahn|Python](http://autobahn.ws/python/) supports both asyncio and Twisted. Ashish reports that Twisted is complicated and will take weeks to learn. asyncio has a back-port to Python2: [Trollius](https://pypi.python.org/pypi/trollius).

## ProtoBuf Library

Unfortunately the Python ProtoBuf library provided by Google [[tutorial]](https://developers.google.com/protocol-buffers/docs/pythontutorial) is much slower than the C++ version due to being purely in Python. If you compile protobuf from source in a special way you get a version with an experimental C extension wrapping Google's C++ protobuf implementation, which is much faster but there are reports of bugs [[1]](http://blog.amvtek.com/posts/2014/Jul/12/comparing-python-performance-of-protobufthrift-serialization/) [[2]](https://github.com/rethinkdb/rethinkdb/issues/1026). As of late June 2014 a solid version (2.6.0) is [supposed to be available](http://code.google.com/p/protobuf/issues/detail?id=286#c10) in 2014 Q3 (this date seems a bit more solid than earlier plans for such a release over the last couple years).

even faster by using bindings directly to the generated C++ classes: <http://yz.mit.edu/wp/fast-native-c-protocol-buffers-from-python/>

There is also Palm, by Bump but that is [reportedly](https://github.com/rethinkdb/rethinkdb/issues/1026) very very slow for encoding.

<https://github.com/connexio/cypb>

[CyPB](https://github.com/connexio/cypb) is in alpha, but is reported to be twice as fast as even the experimental C extension wrapping Google's C++ protobuf implementation. Doesn’t support bytes type (which we need) and no updates to code since 2011.  [Fast-python-pb](https://github.com/Cue/fast-python-pb) is at a similar stage.

Plan: use slow Python library for development. If in benchmarking this seems to be a bottleneck, we can decide whether to wait for 2.6.0, to try the experimental extension, or try the “even faster” approach.

# Low Level Design

## Locally maintained state

in interface per-wsmessage for any direction

* + inbuffer: buffer of incoming data
  + state
  + response wsmessage handle or similar
  + driver handle

in interface per-wsmessage for consume-direction

* + last\_sent\_msgnum: msg sent counter (message num)
  + max\_msgnum\_to\_send: highest message number allowed to send (most recently received)
  + outqueue: small buffer of messages to send
  + send\_a\_marker
  + last message sent marker
  + last message sent ID

in interface per-wsmessage for consume-direction

* + last\_rcvd\_msgnum: last received message number
  + last\_acked\_msgnum: the last message number that we sent a status for
  + max\_messages\_to\_request: max number of messages we want to get at a time (probably based on memory or driver limitations)

## Code flow

### Interface routines

send\_a\_pbmessage:

create bytes for base 127 varint encoding of length (can cache)

invoke ws api to send a partial wsmessage immediately, sending length bytes + serialized pbmessage

client receives a partial wsmessage

...

interface’s new-data-in-wsmessage handler (onMessageFrameData):

adds new data to inbuffer (a [ProtoBufStream](#_ProtoBufStream))

loop while inbuffer is not empty

tries to parse a pbmessage length prefix off start of inbuffer

if length prefix is cut off, break out of loop

if length prefix exceeds 5 bytes, break out of loop and close wsmessage with an error

if don’t have <pbmessage length> amount of additional data in inbuffer, break out of loop

discard pbmessage length prefix

remove <pbmessage length> bytes of inbufffer and save as our current pbmessage (prefer zero-copy)

switch(state)

state=NEW:

parse pbmessage as a SetupRequest as setup-request

if parse fails, close wsmessage with an error

validate setup-request

if validation fails, close wsmessage with an error

if setup-request contains a “get” element:

get that element as get-setup-request

validate get-setup-request (a ConsumeSetupRequest)

if validation fails, close wsmessage with an error

verify user has access to named queue

…

tell driver to get ready for get request based on queue name, starting marker, echo requested, and include claimed

…

state=GETTING

init state for per-wsmessage for consume-direction

call send\_a\_pbmessage with a status message with 100

if setup-request contains a “append” element:

get that element as append-setup-request

validate append-setup-request (a ProduceSetupRequest)

if validation fails, close wsmessage with an error

verify user has access to named queue

…

tell driver to get ready for append request based on queue name

…

state=APPENDING

init state for per-wsmessage for produce-direction

create a ProduceSetupResponse with status message with 100 and with max\_msgnum\_to\_send set to 0+max\_messages\_to\_request

call send\_a\_pbmessage with the ProduceSetupResponse

state=GETTING:

parse pbmessage as a ConsumeRequest as consume-request

if parse fails, close wsmessage with an error

validate consume-request

if validation fails, close wsmessage with an error

if consume-request contains a send\_marker and it is true, set send\_a\_marker

update max\_msgnum\_to\_send to be allowed\_message\_num from consume-request

invoke send\_messages\_if\_avail

…

state=APPENDING:

parse pbmessage as a ProduceRequest as produce-request

if parse fails, close wsmessage with an error

validate produce-request

if validation fails, close wsmessage with an error (since we couldn’t parse, we don’t know what message number the client is on any more)

last\_rcvd\_msgnum += number of payloads in produce-request

driver-append-result = [ ]

for each message payload in produce-request:

tell driver to append that payload with the TTL from produce-request:

...

append driver’s response to driver-append-result

while (last\_acked\_msgnum <= last\_rcvd\_msgnum):

find the number of entries in driver-append-result that are identical (same status) to the first entry, left shifting off these entries

last-msgnum-with-same-status = last-acked-msgnum + this number

create a ProduceResponse with:

status=a corresponding status

msgnum\_status\_is\_thru=last-msgnum-with-same-status

max\_msgnum\_to\_send=last\_rcvd\_msgnum+max\_messages\_to\_request

last\_acked\_msgnum = last-msgnum-with-same-status

assert: driver\_append\_results is empty

assert: last\_acked\_msgnum == last\_rcvd\_msgnum

### Interface routines (for consume direction only)

get\_messages\_if\_avail: // non-blocking check for new message in queue

ask driver for any new messages up to size of our outqueue for the get request, along with marker metadata // non-blocking

add these messages to outqueue, with current time associated with each

buffered\_send\_messages: // send messages in outqueue (subject to buffering)

return if outqueue is empty

return if last\_sent\_msgnum >= max\_msgnum\_to\_send

do\_send = false

do\_send ||= size of outqueue > PREFERED\_MIN\_NUM\_MSGS\_TO\_SEND

do\_send ||= (now - enqueue time for first item in outqueue) > MAX\_WAIT\_TO\_SEND

if do\_send invoke send\_messages\_now

send\_messages\_now:

if not in a GETTING state return

start a ConsumeResponse instance

num\_mess\_to\_send=min(max\_msgnum\_to\_send-last\_sent\_msgnum,size of outqueue)

if num\_mess\_to\_send = 0 return

for i in 1 .. num\_mess\_to\_send

take off head of queue and create a MessageAndMetadata instance from it, filling in id, ttl, age, claim, payload

if i == num\_mess\_to\_send: # last message to send

if send\_a\_marker is set, fill in marker as well and clear send\_a\_marker

record last message sent marker to state

record last message sent ID to state

add the MessageAndMetadata to the ConsumeResponse

call send\_a\_pbmessage with ConsumeResponse

last\_send\_msgnum+= num\_mess\_to\_send

close out a consume response wsmessage:

call send\_messages\_now to send any much as can out of queue *# Q: should we really do this? don’t think so since client has closed our long running communication*

if send\_a\_marker is true

create a MessageAndMetadata with last message sent marker last message sent ID

call send\_a\_pbmessage with this

tell ws lib to close out the response wsmessage

clients gets the final frame in a message

...

reset/clear per-wsmessage for consume-direction state including contents of outqueue

### Interface routines (for produce direction only)

close out a produce response wsmessage: (it’s time to wrap up current append work)

tell driver done with current append to queue

reset/clear per-wsmessage for produce-direction state

### Setup/teardown for queue operation

=> client wants to start getting messages from a queue or client wants to start appending messages to a queue

if no existing ws connection, establishes one

ws lib receives connection

ws lib completes ws connection setup

ws lib tells interface [ws handler] of the new connection (creates new instance of interface?) *# this is not actually available from AutoBahn; instead we can do these the first time that onMessageBegin() is called*

interface initializes local state

interface sets up driver

driver initializes state (perhaps connecting to storage)

=> client sends SetupRequest with ConsumeSetupRequest or ProduceSetupRequest on ws connection

ws lib receives some data on ws connection

ws lib sees this is a new wsmessage

ws lib invokes new-wsmessage handler (onMessageBegin) in interface

close out existing response wsmessage (see above) if there is one *# shouldn’t be needed, that is closed out with the end of the previous inbound wsmessage*

interface initializes some per-wsmessage state including state=NEW

ws lib invokes new-data-in-wsmessage handler (onMessageFrameData) in interface (see above)

=> client finishes wsmessage

ws lib sees wsmessage wrapped up

ws lib invokes new-data-in-wsmessage handler (onMessageFrameData) in interface (see above) if there is any content in final frame

ws lib invokes terminated-wsmessage handler (onMessageEnd) in interface

if state=GETTING:

close out a consume response wsmessage (see above)

if state=APPENDING:

close out a produce response wsmessage (see above)

state=CLOSED

### Requests (client to server)

=> client sends a ConsumeRequest or ProduceRequest on ws connection

ws lib receives some data on ws connection

ws lib invokes new-data-in-wsmessage handler (onMessageFrameData) in interface (see above)

### Other events

=> every N ms run get\_messages\_if\_avail() and buffered\_send\_messages() [even better if we can be directly invoked when queue messages are available

=> (can be combined with the above) every N ms check if we have been waiting long enough to send [only needed if messages in outqueue]

=> client closes ws connection

ws lib closes on server side

## Classes

need a message class to represent messages between driver and interface and within interface (e.g., inside outqueue); consider directly using class corresponding to MessageAndMetadata pbmessage

we’ll need to have a WebSocket protocol class that AutoBahn will call. This will follow the [IWebSocketChannelFrameApi](https://github.com/tavendo/AutobahnPython/blob/master/autobahn/autobahn/websocket/interfaces.py) interface.

…

# ProtoBufStream

A ProtoBufStream object accepts an incrementally arriving protobufs with length prefix (such as seen in the sequence of fragments that arrive in a wsmessage in Queuing Streaming API) and provides routines for getting the leading protobufs off the stream.  At minimum operations like this is needed because Websocket does not guarantee that the Websocket fragments that arrive at a destinaton will be divided up in the same way at the source.  We try to minimize unnecessary memory copies.

Constructor: \_\_init\_\_(self)

Initializes the buffer to hold the messages.

append(self,data)

Appends the incoming data to the buffer

shift\_first\_pb(self)

returns bytes if pb\_message is available and parsed successfully - None if no pb\_message is available in the buffer and raises a MessageParsingException in case of parsing error

parse\_off\_first\_pb(self, message\_name)

like shift\_first\_pb() but what is returned in a pbmessage parsed as the pbmessage with the given name

encode\_length\_prefix(self, integer\_length)

encodes the provided uint32 length to Base 128 Varints

## ProtoBufStream implementation suggested changes

* add comments describing the class and describing the methods
* add comments describing how data is stored in the class, optimizations in place, etc
* for the append methods, rename the “inbuffer” parameter to “new\_data” to clarity and to reduce confusion
* handle error cases more completely
* see if we can reduce memory copies further, at least in the common case where what is being added exactly corresponds to a protobuf

# Future work

These are some things in the server that are not yet in place what we would probably need to add before we start using the code in production:

* run over SSL (wss instead of ws)
* implement Keystone authentication
* add authorization check
* fix responding to send\_a\_marker
* handle error cases
  + server currently just does an exit(1) for all error cases, but we should follow protocol guidelines for how to handle error cases (e.g., set a non-100 status code and mb end the wsmessage)
* switch to using new parse\_off\_first\_pb() method from ProtoBufStream class which reduces copies by decoding next message as the specified pbmessage
* add buffering of outgoing messages based on design above (preferred min batch size but with a max wait time; both params being configurable)
* make the buffer of outgoing messages a configurable size
* handle errors from protobuf library
* remove some of debugging code
* add a mechanism to allow the driver know precisely which messages were received by (or at least sent to) a consuming client
* possibly: switch to produce-direction interface with driver to be based on MessageStream and the driver needing to check that regularly
* make use of ability to cancel call\_later() rather than just checking state when get run
* for upgrade scenarios support suspending access from remote producer
* support live migration to Kafka in a different SPaaS cluster (support upgrade scenario)