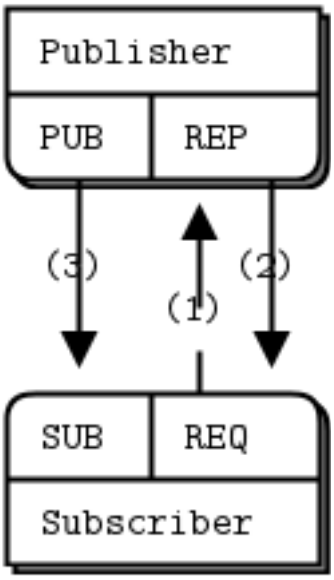
zenAptix Aqueduct
Aquifer Subscription Protocol

Prepared by: Ian de Beer
for zenAptix (Pty) Ltd

Date: 13 March 2015

Overview

The Gateway Publisher receives messages form a specific Aquifer source. The publisher must buffer messages or not consume messages from the source until it receives a subscription from a Subscriber. The decision to allow the Publisher to accept multiple Subscription with differing consumption rates lies with the implementer of the Publisher.



The REQ/REP channel is used for **control** messages and the PUB/SUB channel is used for **data** messages. The Control message structure is based on the Reactive Streams API.

The Aquifer service will send the Gateway a Subscription Request, requesting records to be sent over the PUB/SUB socket. The Publisher will respond with a onSubscription Response containing the socket over which the pub/sub session will proceed.

Once the Codec service has subscribed to the provided pub/sub socket it will send a Next message requesting a number of records it can handle. The Gateway server will will send a onNext which might be less or equal to the amount requested by the subscriber.

On receiving all the messages as declared by the Publisher the Subscriber can request the next batch of records. If the Subscriber has received less than the declared amount within a time-out period it must log the deficit and adjust the request rate downward to accommodate the network latency.

Unless we implement a broker like Kafka that supports reliable messaging, we'll use this best effort approach which will definitely be much better than our current approach

Reactive Streams

(http://www.reactive-streams.org
https://www.youtube.com/watch?v=khmVMvlP_QA&noredirect=1)

Publisher

1. The total number of onNext signals sent by a Publisher to a Subscriber MUST be less than or equal to the total number of elements requested by that Subscriber´s Subscription at all times.

2. A Publisher MAY signal less onNext than requested and terminate the Subscription by calling onComplete or onError.

3. onSubscribe, onNext, onError and onComplete signalled to a Subscriber MUST be signalled sequentially (no concurrent notifications).

4. If a Publisher fails it MUST signal an onError.

5. If a Publisher terminates successfully (finite stream) it MUST signal an onComplete.

6. If a Publisher signals either onError or onComplete on a Subscriber, that Subscriber's Subscription MUST be considered cancelled.

7. Once a terminal state has been signalled (onError, onComplete) it is REQUIRED that no further signals occur.

8. If a Subscription is cancelled its Subscriber MUST eventually stop being signalled.

9. Publisher.subscribe MUST call onSubscribe on the provided Subscriber prior to any other signals to that Subscriber and MUST return normally, except when the provided Subscriber is null in which case it MUST throw a java.lang.NullPointerException to the caller, for all other situations the only legal way to signal failure (or reject the Subscriber) is by calling onError (after calling onSubscribe).

10. Publisher.subscribe MAY be called as many times as wanted but MUST be with a different Subscriber each time

11. A Publisher MAY support multiple Subscribers and decides whether each Subscription is unicast or multicast.

12. A Publisher MUST produce the same elements, starting with the oldest element still available, in the same sequence for all its subscribers and MAY produce the stream elements at (temporarily) differing rates to different subscribers.

Subscriber

1. A Subscriber MUST signal demand via Subscription.request(long n) to receive onNext signals.

2. If a Subscriber suspects that its processing of signals will negatively impact its Publisher's responsiveness, it is RECOMMENDED that it asynchronously dispatches its signals.

3. Subscriber.onComplete() and Subscriber.onError(Throwable t) MUST NOT call any methods on the Subscription or the Publisher.

4. Subscriber.onComplete() and Subscriber.onError(Throwable t) MUST consider the Subscription cancelled after having received the signal.

5. A Subscriber MUST call Subscription.cancel() on the given Subscription after an onSubscribe signal if it already has an active Subscription.

6. A Subscriber MUST call Subscription.cancel() if it is no longer valid to the Publisher without the Publisher having signalled onError or onComplete.

7. A Subscriber MUST ensure that all calls on its Subscription take place from the same thread or provide for respective external synchronization.

8. A Subscriber MUST be prepared to receive one or more onNext signals after having called Subscription.cancel() if there are still requested elements pending. Subscription.cancel() does not guarantee to perform the underlying cleaning operations immediately.

9. A Subscriber MUST be prepared to receive an onComplete signal with or without a preceding Subscription.request(long n) call.

10. A Subscriber MUST be prepared to receive an onError signal with or without a preceding Subscription.request(long n) call.

11. A Subscriber MUST make sure that all calls on its onXXX methods happen-before [1] the processing of the respective signals. I.e. the Subscriber must take care of properly publishing the signal to its processing logic.

12. Subscriber.onSubscribe MUST be called at most once for a given Subscriber (based on object equality).

13. Calling onSubscribe, onNext, onError or onComplete MUST return normally except when any provided parameter is null in which case it MUST throw a java.lang.NullPointerException to the caller, for all other situations the only legal way for a Subscriber to signal failure is by cancelling its Subscription. In the case that this rule is violated, any associated Subscription to the Subscriber MUST be considered as cancelled, and the caller MUST raise this error condition in a fashion that is adequate for the runtime environment.

Subscription

1. Subscription.request and Subscription.cancel MUST only be called inside of its Subscriber context. A Subscription represents the unique relationship between a Subscriber and a Publisher [see 2.12].

2. The Subscription MUST allow the Subscriber to call Subscription.request synchronously from within onNext or onSubscribe.

3. Subscription.request MUST place an upper bound on possible synchronous recursion between Publisher and Subscriber

4. Subscription.request SHOULD respect the responsiveness of its caller by returning in a timely manner

5. Subscription.cancel MUST respect the responsiveness of its caller by returning in a timely manner[2], MUST be idempotent and MUST be thread-safe.

6. After the Subscription is cancelled, additional Subscription.request(long n) MUST be NOPs.

7. After the Subscription is cancelled, additional Subscription.cancel() MUST be NOPs.

8. While the Subscription is not cancelled, Subscription.request(long n) MUST register the given number of additional elements to be produced to the respective subscriber.

9. While the Subscription is not cancelled, Subscription.request(long n) MUST signal onError with a java.lang.IllegalArgumentException if the argument is <= 0. The cause message MUST include a reference to this rule and/or quote the full rule.

10. While the Subscription is not cancelled, Subscription.request(long n) MAY synchronously call onNext on this (or other) subscriber(s).

11. While the Subscription is not cancelled, Subscription.request(long n) MAY synchronously call onComplete or onError on this (or other) subscriber(s).

12. While the Subscription is not cancelled, Subscription.cancel() MUST request the Publisher to eventually stop signalling its Subscriber. The operation is NOT REQUIRED to affect the Subscription immediately.

13. While the Subscription is not cancelled, Subscription.cancel() MUST request the Publisher to eventually drop any references to the corresponding subscriber. Re-subscribing with the same Subscriber object is discouraged [see 2.12], but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.

14. While the Subscription is not cancelled, calling Subscription.cancel MAY cause the Publisher, if stateful, to transition into the shut-down state if no other Subscription exists at this point

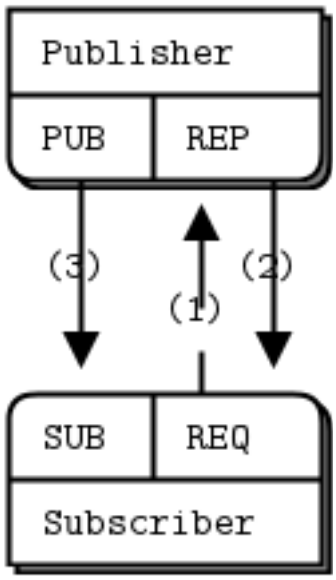
15. Calling Subscription.cancel MUST return normally. The only legal way to signal failure to a Subscriber is via the onError method.

16. Calling Subscription.request MUST return normally. The only legal way to signal failure to a Subscriber is via the onError method.

17. A Subscription MUST support an unbounded number of calls to request and MUST support a demand (sum requested - sum delivered) up to 2^63-1 (java.lang.Long.MAX_VALUE). A demand equal or greater than 2^63-1 (java.lang.Long.MAX_VALUE) MAY be considered by the Publisher as “effectively unbounded”[1].

ØMQ

1. Subscriber sends Subscription Request asking for a maximum number of messages to be sent over the Pub/Sub channel
2. Publisher sends Subscription Reply indicating the number of messages that will be sent
3. The Publisher send the indicated number of messages to the subscriber



JSON Messages:

Subscribe Request message

{"jsonClass":"Subscribe","cancel":false}

Subscribe Response message

{"jsonClass":"OnSubscribe","size":9223372036854775807,"port":5557}

Next Request message

{"jsonClass":"Next","count":1000}

Next Response message

{"jsonClass":"OnNext","count":1000}

Error Response message

{"jsonClass":"OnError","message":"stuffed"}

Complete Response message

{"jsonClass":"OnComplete","complete":true}