RabbitMQ Stream Java Client

Version 0.1.0-SNAPSHOT: (db795f0)

Table of Contents

The RabbitMQ Stream Java Client is a Java library to communicate with the RabbitMQ Stream Plugin. It allows creating and deleting streams, as well as publishing to and consuming from these streams. Learn more in the the client overview.

What is a RabbitMQ Stream?

A RabbitMQ stream is a persistent and replicated data structure that models an append-only log. It differs from the classical RabbitMQ queue in the way message consumption works. In a classical RabbitMQ queue, consuming removes messages from the queue. In a RabbitMQ stream, consuming leaves the stream intact. So the content of a stream can be read and re-read without impact or destructive effect.

None of the stream or classical queue data structure is better than the other, they are usually suited for different use cases.

When to Use RabbitMQ Stream?

A stream abstraction is useful when one or several consumer applications require the whole history of data ("replay").

Streams can also be useful when a higher throughput than with classical RabbitMQ queues is required. RabbitMQ streams use various optimization techniques, and a custom network protocol to achieve performances that are not possible with the other protocols supported in RabbitMQ (AMQP, STOMP, MQTT). This does not make the stream protocol better than these protocols, they just all serve different purposes.

Other Way to Use Streams in RabbitMQ

It is also possible to use the stream abstraction in RabbitMQ with the AMQP 0-9-1 protocol. Instead of consuming from a stream with the stream protocol, one consumes from a "stream-powered" queue with the AMQP 0-9-1 protocol. A "stream-powered" queue is a special type of queue that is backed up with a stream infrastructure layer and adapted to provide the stream semantics (mainly non-destructive reading).

Using such a queue has the advantage to provide the features inherent to the stream abstraction (append-only structure, non-destructive reading) with any AMQP 0-9-1 client library. This is clearly interesting when considering the maturity of AMQP 0-9-1 client libraries and the ecosystem around AMQP 0-9-1.

But by using it, one does not benefit from the performance of the stream protocol, which has been designed for performance in mind, whereas AMQP 0-9-1 is a more general-purpose protocol.

It is not possible to use "stream-powered" queues with the stream Java client, you need to use an AMQP 0-9-1 client library.

Guarantees

RabbitMQ stream provides at-least-once guarantees thanks to the publisher confirm mechanism, which is supported by the stream Java client.

Stream Client Overview

The RabbitMQ Stream Java Client implements the RabbitMQ Stream protocol and avoids dealing with low-level concerns by providing high-level functionalities to build fast, efficient, and robust client applications.

- *administrate streams (creation/deletion) directly from applications.* This can also be useful for development and testing.
- *adapt publishing throughput* thanks to the configurable batch size and flow control.
- *consume asynchronously from streams and resume where left off* thanks to automatic or manual offset tracking.
- *use cluster nodes appropriately* by letting the client decide which node to connect to - to stream leaders for publishers to avoid network traffic and to stream replicas for consumers to offload leaders.
- *optimize resources* thanks to automatic growing and shrinking of connections depending on the number of publishers and consumers.
- *let the client handle network failure* thanks to automatic connection recovery and automatic re-subscription for consumers.

The Stream Java Client

The library requires Java 8 or more.

Setting up RabbitMQ

A RabbitMQ node with the stream plugin enabled is required. The easiest way to get up and running is to use Docker. It is also possible to use the generic Unix package.

With Docker

The following command creates a one-time Docker container to run RabbitMQ with the stream plugin enabled:

Running the stream plugin with Docker

```
docker run -it --rm --name rabbitmq -p 5555:5555 pivotalrabbitmq/rabbitmq-stream
```

The previous command exposes only the stream port (5555), you can expose ports for other protocols:

Exposing the AMQP 0.9.1 and management ports:

```
docker run -it --rm --name rabbitmq -p 5555:5555 -p 5672:5672 -p 15672:15672 \
    pivotalrabbitmq/rabbitmq-stream
```

Refer to the official RabbitMQ Docker image web page to find out more about its usage. Make sure to use the `pivotalrabbitmq/rabbitmq-stream` image in the command line.

The `pivotalrabbitmq/rabbitmq-stream` Docker image is meant for development usage only. It does not support all the features of the official Docker image, like TLS.

With the Generic Unix Package

The generic Unix package requires Erlang to be installed.

- Download the latest generic Unix alpha from Bintray.
- Follow the instructions to install the generic Unix package.
- Enable the plugin `./rabbitmq-plugins enable rabbitmq_stream`.
- Start the broker `./rabbitmq-server -detached`. This starts the stream listener on port 5555.

Dependencies

Use your favorite build management tool to add the client dependencies to your project.

Note the client uses the Apache QPid Proton-J library for AMQP 1.0 message encoding and decoding.

Maven

pom.xml

```
<dependencies>

  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>stream-client</artifactId>
    <version>0.1.0-SNAPSHOT</version>
  </dependency>

  <dependency>
    <groupId>org.apache.qpid</groupId>
    <artifactId>proton-j</artifactId>
    <version>0.33.6</version>
  </dependency>

</dependencies>

<repositories>

  <repository>
    <id>ossrh</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots</url>
    <snapshots><enabled>true</enabled></snapshots>
    <releases><enabled>false</enabled></releases>
  </repository>

</repositories>
```

Gradle

build.gradle

```
dependencies {
  compile "com.rabbitmq:stream-client:0.1.0-SNAPSHOT"
  compile "org.apache.qpid:proton-j:0.33.6"
}

repositories {
  maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
  mavenCentral()
}
```

Sample Application

This section covers the basics of the RabbitMQ Stream Java API by building a small publish/consume application. This is a good way to get an overview of the API. If you want a more comprehensive introduction, you can go to the the reference documentation section.

The sample application publishes some messages and then registers a consumer to make some computations out of them. The source code is available on GitHub.

The sample class starts with a few imports:

Imports for the sample application

```
import com.rabbitmq.stream.*;
import java.util.UUID;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicLong;
import java.util.stream.IntStream;
```

The next step is to create the `Environment`. It is a management object used to manage streams and create producers as well as consumers. The next snippet shows how to create an `Environment` instance and create the stream used in the application:

Creating the environment

```
System.out.println("Connecting...");
Environment environment = Environment.builder().build();  ①
String stream = UUID.randomUUID().toString();
environment.streamCreator().stream(stream).create();  ②
```

① Use `Environment#builder` to create the environment

② Create the stream

Then comes the publishing part. The next snippet shows how to create a `Producer`, send messages, and handle publishing confirmations, to make sure the broker has taken outbound messages into account. The application uses a count down latch to move on once the messages have been confirmed.

Publishing messages

```
System.out.println("Starting publishing...");
int messageCount = 10000;
CountDownLatch publishConfirmLatch = new CountDownLatch(messageCount);
Producer producer = environment.producerBuilder()  ①
        .stream(stream)
        .build();
IntStream.range(0, messageCount)
        .forEach(i -> producer.send(  ②
                producer.messageBuilder()                    ③
                    .addData(String.valueOf(i).getBytes())   ③
                    .build(),                                ③
                confirmationStatus -> publishConfirmLatch.countDown()  ④
        ));
publishConfirmLatch.await(10, TimeUnit.SECONDS);  ⑤
producer.close();  ⑥
System.out.printf("Published %,d messages%n", messageCount);
```

① Create the `Producer` with `Environment#producerBuilder`

② Send messages with `Producer#send(Message, ConfirmationHandler)`

③ Create a message with `Producer#messageBuilder`

④ Count down on message publishing confirmation

⑤ Wait for all publishing confirmations to have arrived

⑥ Close the producer

It is now time to consume the messages. The `Environment` lets us create a `Consumer` and provide some logic on each incoming message by implementing a `MessageHandler`. The next snippet does this to calculate a sum and output it once all the messages have been received:

Consuming messages

```
System.out.println("Starting consuming...");
AtomicLong sum = new AtomicLong(0);
CountDownLatch consumeLatch = new CountDownLatch(messageCount);
Consumer consumer = environment.consumerBuilder()  ①
        .stream(stream)
        .messageHandler((offset, message) -> {  ②
            sum.addAndGet(Long.parseLong(new String(message.getBodyAsBinary())));  ③
            consumeLatch.countDown();  ④
        })
        .build();

consumeLatch.await(10, TimeUnit.SECONDS);  ⑤

System.out.println("Sum: " + sum.get());  ⑥

consumer.close();  ⑦
```

① Create the `Consumer` with `Environment#consumerBuilder`

② Set up the logic to handle messages

③ Add the value in the message body to the sum

④ Count down on each message

⑤ Wait for all messages to have arrived

⑥ Output the sum

⑦ Close the consumer

The application has some cleaning to do before terminating, that is deleting the stream and closing the environment:

Cleaning before terminating

```
environment.deleteStream(stream);  ①
environment.close();  ②
```

① Delete the stream

② Close the environment

You can run the sample application from the root of the project (you need a running local RabbitMQ node with the stream plugin enabled):

```
$ ./mvnw -q test-compile exec:java -Dexec.classpathScope="test" \
    -Dexec.mainClass="com.rabbitmq.stream.docs.SampleApplication"
Starting publishing...
Published 10000 messages
Starting consuming...
Sum: 49995000
```

You can remove the `-q` flag if you want more insight on the execution of the build.

RabbitMQ Stream Java API

Overview

This section describes the API to connect to the RabbitMQ Stream Plugin, publish messages, and consume messages. There are 3 main interfaces:

- `com.rabbitmq.stream.Environment` for connecting to a node and optionally managing streams.
- `com.rabbitmq.stream.Producer` to publish messages.
- `com.rabbitmq.stream.Consumer` to consume messages.

Environment

Creating the Environment

The environment is the main entry point to a node or a cluster of nodes. `Producer` and `Consumer` instances are created from an `Environment` instance. Here is the simplest way to create an `Environment` instance:

Creating an environment with all the defaults

```
Environment environment = Environment.builder().build();  ①
// ...
environment.close(); ②
```

① Create an environment that will connect to localhost:5555

② Close the environment after usage

Note the environment must be closed to release resources when it is no longer needed.

Consider the environment like a long-lived object. An application will usually create one `Environment` instance when it starts up and close it when it exits.

It is possible to use a URI to specify all the necessary information to connect to a node:

Creating an environment with a URI

```
Environment environment = Environment.builder()
        .uri("rabbitmq-stream://guest:guest@localhost:5555/%2f")  ①
        .build();
```

① Use the `uri` method to specify the URI to connect to

The previous snippet uses a URI that specifies the following information: host, port, username, password, and virtual host (`/`, which is encoded as `%2f`). The URI follows the same rules as the AMQP 0.9.1 URI, except the protocol must be `rabbitmq-stream` and TLS is not supported.

When using one URI, the corresponding node will be the main entry point to connect to. The `Environment` will then use the stream protocol to find out more about streams topology (leaders and replicas) when asked to create `Producer`s and `Consumer`s. The `Environment` may become blind if this node goes down though, so it may be more appropriate to specify several other URIs to try in case of failure of a node:

Creating an environment with several URIs

```
Environment environment = Environment.builder()
        .uris(Arrays.asList(                    ①
                "rabbitmq-stream://host1:5555",
                "rabbitmq-stream://host2:5555",
                "rabbitmq-stream://host3:5555")
        )
        .build();
```

① Use the `uris` method to specify several URIs

By specifying several URIs, the environment will try to connect to the first one, and will pick a new URI randomly in case of disconnection.

The following table sums up the main settings to create a `Environment`:

Parameter Name	Description	Default
`uri`	The URI of the node to connect to (single node).	`rabbitmq-stream://guest:guest@localhost:5555/%2f`
`uris`	The URI of the nodes to try to connect to (cluster).	`rabbitmq-stream://guest:guest@localhost:5555/%2f` singleton list
`host`	Host to connect to.	`localhost`
`port`	Port to use.	`5555`
`username`	Username to use to connect.	`guest`
`password`	Password to use to connect.	`guest`
`virtualHost`	Virtual host to connect to.	`/`
`recoveryBackOffDelayPolicy`	Delay policy to use for backoff on connection recovery.	Fixed delay of 5 seconds
`topologyUpdateBackOffDelayPolicy`	Delay policy to use for backoff on topology update, e.g. when a stream replica moves and a consumer needs to connect to another node.	Initial delay of 5 seconds then delay of 1 second.

Parameter Name	Description	Default
`scheduledExecutorService`	Executor used to schedule infrastructure tasks like background publishing, producers and consumers migration after disconnection or topology update. If a custom executor is provided, it is the developer's responsibility to close it once it is no longer necessary.	`Executors.newScheduledThreadPool(Runtime.getRuntime().availableProcessors());`

Managing Streams

Streams are usually long-lived, centrally-managed entities, that is, applications are not supposed to create and delete them. It is nevertheless possible to create and delete stream with the `Environment`. This comes in handy for development and testing purposes.

Streams are created with the `Environment#streamCreator()` method:

Creating a stream

```
environment.streamCreator().stream("my-stream").create();  ①
```

① Create the `my-stream` stream

Streams can be deleted with the `Environment#delete(String)` method:

Deleting a stream

```
environment.deleteStream("my-stream");  ①
```

① Delete the `my-stream` stream

Note you should avoid stream churn (creating and deleting streams repetitively) as their creation and deletion imply some significant housekeeping on the server side (interactions with the file system, communication between nodes of the cluster).

It is also possible to limit the size of a stream when creating it. A stream is an append-only data structure and reading from it does not remove data. This means a stream can grow indefinitely. RabbitMQ Stream supports a size-based retention policy: once the stream reaches a given size, it is truncated (starting from the beginning).

IMPORTANT

Limit the size of streams if appropriate!

Make sure to set up a retention policy on potentially large streams if you don't want to saturate the storage devices of your servers. Keep in mind that this means some data will be erased!

It is possible to set up the retention policy when creating the stream:

Setting the retention policy when creating a stream

```
environment.streamCreator()
        .stream("my-stream")
        .maxLengthBytes(ByteCapacity.GB(10))  ①
        .maxSegmentSizeBytes(ByteCapacity.MB(500))  ②
        .create();
```

① Set the maximum size to 10 GB

② Set the segment size to 500 MB

The previous snippet mentions a segment size. RabbitMQ Stream does not store a stream in a big, single file, it uses segment files for technical reasons. A stream is truncated by deleting whole segment files (and not part of them), so the maximum size of a stream is usually significantly higher than the size of segment files. 500 MB is a reasonable segment file size to begin with.

Producer

Creating a Producer

A `Producer` instance is created from the `Environment`. The only mandatory setting to specify is the stream to publish to:

Creating a producer from the environment

```
Producer producer = environment.producerBuilder()  ①
        .stream("my-stream")  ②
        .build();  ③
// ...
producer.close();  ④
```

① Use `Environment#producerBuilder()` to define the producer

② Specify the stream to publish to

③ Create the producer instance with `build()`

④ Close the producer after usage

Consider a `Producer` instance like a long-lived object, do not create one to send just one message.

Internally, the `Environment` will query the broker to find out about the topology of the stream and will create or re-use a connection to publish to the leader node of the stream.

The following table sums up the main settings to create a `Producer`:

Parameter Name	Description	Default
`batchSize`	The maximum number of messages to accumulate before sending them to the broker.	100

Parameter Name	Description	Default
`subEntrySize`	The number of messages to put in a sub-entry. A sub-entry is one "slot" in a publishing frame, meaning outbound messages are not only batched in publishing frames, but in sub-entries as well. Use this feature to increase throughput at the cost of increased latency.	1 (meaning no use of sub-entry batching)
`maxUnconfirmedMessages`	The maximum number of unconfirmed outbound messages. `Producer#send` will start blocking when the limit is reached.	10,000
`batchPublishingDelay`	Period to send a batch of messages.	100 ms

Sending Messages

Once a `Producer` has been created, it is possible to send a message with the `Producer#send(Message, ConfirmationHandler)` method. The following snippet shows how to publish a message with a byte array payload:

Sending a message

```
byte[] messagePayload = "hello".getBytes(StandardCharsets.UTF_8);  ①
producer.send(
        producer.messageBuilder().addData(messagePayload).build(),  ②
        confirmationStatus -> {  ③
            if (confirmationStatus.isConfirmed()) {
                // the message made it to the broker
            } else {
                // the message did not make it to the broker
            }
        });
```

① The payload of a message is an array of bytes

② Create the message with `Producer#messageBuilder()`

③ Define the behavior on publish confirmation

Messages are not only made of a `byte[]` payload, we will see in the next section they can also carry pre-defined and application properties.

The `ConfirmationHandler` defines an asynchronous callback invoked when the client received from the broker the confirmation the message has been taken into account. The `ConfirmationHandler` is the place for any logic on publishing confirmation, including re-publishing the message if it is

negatively acknowledged.

Working with Complex Messages

The publishing example above showed that messages are made of a byte array payload, but it did not go much further. Messages in RabbitMQ Stream can actually be more sophisticated, as they comply to the AMQP 1.0 message format.

In a nutshell, a message in RabbitMQ Stream has the following structure:

- properties: *a defined set of standard properties of the message* (e.g. message ID, correlation ID, content type, etc).
- application properties: a set of arbitrary key/value pairs.
- body: typically an array of bytes.
- message annotations: a set of key/value pairs (aimed at the infrastructure).

The RabbitMQ Stream Java client uses the `Message` interface to abstract a message and the recommended way to create `Message` instances is to use the `Producer#messageBuilder()` method. To publish a `Message`, use the `Producer#send(Message,ConfirmationHandler)`:

Creating a message with properties

```
Message message = producer.messageBuilder()  ①
        .properties()  ②
            .messageId(UUID.randomUUID())
            .correlationId(UUID.randomUUID())
            .contentType("text/plain")
        .messageBuilder()  ③
            .addData("hello".getBytes(StandardCharsets.UTF_8))  ④
        .build();  ⑤
producer.send(message, confirmationStatus -> { }); ⑥
```

① Get the message builder from the producer

② Get the properties builder and set some properties

③ Go back to message builder

④ Set byte array payload

⑤ Build the message instance

⑥ Publish the message

NOTE

Is RabbitMQ Stream based on AMQP 1.0?

AMQP 1.0 is a standard that defines *an efficient binary peer-to-peer protocol for transporting messages between two processes over a network*. It also defines *an abstract message format, with concrete standard encoding*. This is only the latter part that RabbitMQ Stream uses. The AMQP 1.0 protocol is not used, only AMQP 1.0 encoded messages are wrapped into the RabbitMQ Stream binary protocol.

The actual AMQP 1.0 message encoding and decoding happen on the client side, the RabbitMQ Stream plugin stores only bytes, it has no idea that AMQP 1.0 message format is used.

AMQP 1.0 message format was chosen because of its flexibility and its advanced type system. It provides good interoperability, which allows streams to be accessed as AMQP 0-9-1 queues, without data loss.

Consumer

`Consumer` is the API to consume messages from a stream.

Creating a Consumer

A `Consumer` instance is created with `Environment#consumerBuilder()`. The main settings are the stream to consume from, the place in the stream to start consuming from (the *offset*), and a callback when a message is received (the `MessageHandler`). The next snippet shows how to create a `Consumer`:

Creating a consumer

```
Consumer consumer = environment.consumerBuilder()  ①
        .stream("my-stream")  ②
        .offset(OffsetSpecification.first())  ③
        .messageHandler((offset, message) -> {
            message.getBodyAsBinary(); ④
        })
        .build();  ⑤
// ...
consumer.close();  ⑥
```

① Use `Environment#consumerBuilder()` to define the consumer

② Specify the stream to consume from

③ Specify where to start consuming from

④ Define behavior on message consumption

⑤ Build the consumer

⑥ Close consumer after usage

The broker start sending messages as soon as the `Consumer` instance is created.

Specifying an Offset

The offset is the place in the stream where the consumer starts consuming from. The possible values for the offset parameter are the following:

- `OffsetSpecification.first()`: starting from the first available offset. If the stream has not been truncated, this means the beginning of the stream (offset 0).
- `OffsetSpecification.last()`: starting from the end of the stream and returning the last chunk of messages immediately (if the stream is not empty).
- `OffsetSpecification.next()`: starting from the next offset to be written. Contrary to `OffsetSpecification.last()`, consuming with `OffsetSpecification.next()` will not return anything if no-one is publishing to the stream. The broker will start sending messages to the consumer when messages are published to the stream.
- `OffsetSpecification.offset(offset)`: starting from the specified offset. 0 means consuming from the beginning of the stream (first messages). The client can also specify any number, for example the offset where it left off in a previous incarnation of the application.
- `OffsetSpecification.timestamp(timestamp)`: starting from the messages stored after the specified timestamp.

Tracking the Offset for a Consumer

A consumer can track the offset it has reached in a stream. This allows a new incarnation of the consumer to restart consuming where it left off. Offset tracking works in 2 steps:

- the consumer must have a **name**. The name is set with `ConsumerBuilder#name(String)`. The name can be any value (under 256 characters) and is expected to be unique (from the application point of view). Note neither the client library, nor the broker enforces uniqueness of the name: if 2 `Consumer` Java instances share the same name, their offset tracking will likely be interleaved, which applications usually do not expect.
- the consumer must periodically **commit the offset** it has reached so far. The way offsets are committed depends on the commit strategy: automatic or manual.

Whatever commit strategy you use, **a consumer must have a name to be able to commit offsets**.

Automatic Offset Commit

The following snippet shows how to enable automatic commit with the defaults:

Using automatic commit strategy with the defaults

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .autoCommitStrategy()   ②
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use automatic commit strategy with defaults

The automatic commit strategy has the following available settings:

- **message count before commit**: the client will commit the offset after the specified number of messages, right after the execution of the message handler. *The default is every 10,000 messages*.
- **flush interval**: the client will make sure to commit the last received offset at the specified interval. This avoids having pending, not committed offsets in case of inactivity. *The default is 5 seconds*.

Those settings are configurable, as shown in the following snippet:

Configuring the automatic commit strategy

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .autoCommitStrategy()   ②
            .messageCountBeforeCommit(50_000)   ③
            .flushInterval(Duration.ofSeconds(10))   ④
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use automatic commit strategy

③ Commit every 50,000 messages

④ Make sure to commit offset at least every 10 seconds

Note the automatic commit is the default commit strategy, so if you are fine with its defaults, it is enabled as soon as you specify a name for the consumer:

Setting only the consumer name to enable automatic commit

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .messageHandler((context, message) -> {
          // message handling code...
        })
        .build();
```

① Set only the consumer name to enable automatic commit with defaults

Automatic commit is simple and provides good guarantees. It is nevertheless possible to have more fine-grained control over offset commit by using manual commit.

Manual Offset Commit

The manual commit strategy lets the developer in charge of committing offsets whenever they want, not only after a given number of messages has been received and supposedly processed, like automatic commit does.

The following snippet shows how to enable manual commit and how to commit the offset at some point:

Using manual commit with defaults

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")   ①
        .manualCommitStrategy()   ②
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...

          if (conditionToCommit()) {
            context.commit();   ③
          }
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use manual commit with defaults

③ Commit at the current offset on some condition

Manual commit has only one setting: the **check interval**. The client checks that the last requested committed offset has been actually committed at the specified interval. *The default check interval is 5 seconds*.

The following snippet shows the configuration of manual commit:

Configuring manual commit strategy

```
Consumer consumer =
    environment.consumerBuilder()
        .stream("my-stream")
        .name("application-1")    ①
        .manualCommitStrategy()    ②
            .checkInterval(Duration.ofSeconds(10))    ③
        .builder()
        .messageHandler((context, message) -> {
          // message handling code...

          if (conditionToCommit()) {
            context.commit();    ④
          }
        })
        .build();
```

① Set the consumer name (mandatory for offset tracking)

② Use manual commit with defaults

③ Check last requested offset every 10 seconds

④ Commit at the current offset on some condition

The snippet above uses `MessageHandler.Context#commit()` to commit at the offset of the current message, but it is possible to commit anywhere in the stream with `MessageHandler.Context#consumer()#commit(long)` or simply `Consumer#commit(long)`.

Considerations On Offset Tracking

When to commit offsets? Avoid committing offsets too often or, worse, for each message. Even though offset tracking is a small and fast operation, it will make the stream grow unnecessarily, as the broker persists offset tracking entries in the stream itself.

A good rule of thumb is to commit every few thousands of messages. Of course, when the consumer will restart consuming in a new incarnation, the last tracked offset may be a little behind the very last message the previous incarnation actually processed, so the consumer may see some messages that have been already processed.

A solution to this problem is to make sure processing is idempotent or filter out the last duplicated messages.

Is the offset a reliable absolute value? Message offsets may not be contiguous. This implies that the message at offset 500 in a stream may not be the 501 message in the stream (offsets start at 0). There can be different types of entries in a stream storage, a message is just one of them. For example, committing an offset creates an offset tracking entry, which has its own offset.

This means one must be careful when basing some decision on offset values, like a modulo to perform an operation every X messages. As the message offsets have no guarantee to be contiguous, the operation may not happen exactly every X messages.

Building the Client

You need JDK 1.8 or more installed.

To build the JAR file:

```
./mvnw clean package -DskipITs -DskipTests
```

To launch the test suite (requires a local RabbitMQ node with stream plugin enabled):

```
./mvnw verify -Drabbitmqctl.bin=/path/to/rabbitmqctl
```

The Performance Tool

The library contains also a performance tool to test the RabbitMQ Stream plugin. It is downloadable from Bintray as an uber JAR and can be built separately as well.

Using the Performance Tool

To launch a run:

```
$ java -jar stream-perf-test-{version}.jar
10:11:54.324 [main] INFO  c.r.stream.perf.StreamPerfTest - Created stream stream1
10:11:54.385 [main] INFO  c.r.stream.perf.StreamPerfTest - Producer will stream
stream1
10:11:54.387 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting consuming on
stream1
10:11:54.390 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting producer
1, published 155230 msg/s, confirmed 147824 msg/s, consumed 124487 msg/s, latency
min/median/75th/95th/99th 1121/8225/17647/62468/73991 µs, chunk size 109
2, published 359193 msg/s, confirmed 336535 msg/s, consumed 306748 msg/s, latency
min/median/75th/95th/99th 1398/56590/80607/127818/135925 µs, chunk size 345
3, published 523429 msg/s, confirmed 509044 msg/s, consumed 478710 msg/s, latency
min/median/75th/95th/99th 1478/29996/69536/111946/135079 µs, chunk size 529
4, published 599735 msg/s, confirmed 594707 msg/s, consumed 568315 msg/s, latency
min/median/75th/95th/99th 964/21032/52977/98643/133399 µs, chunk size 548
5, published 632114 msg/s, confirmed 609804 msg/s, consumed 591426 msg/s, latency
min/median/75th/95th/99th 964/34303/74318/110684/127440 µs, chunk size 588
6, published 619328 msg/s, confirmed 618229 msg/s, consumed 598410 msg/s, latency
min/median/75th/95th/99th 964/45918/86391/114714/138207 µs, chunk size 657
^C
Summary: published 641792 msg/s, confirmed 635240 msg/s, consumed 636256 msg/s,
latency 95th 112730 µs, chunk size 711
```

The previous command will start publishing to and consuming from a stream created only for the test. The tool outputs live metrics on the console and write more detailed metrics in a `stream-perf-test-current.txt` file that get renamed to `stream-perf-test-yyyy-MM-dd-HHmmss.txt` when the run ends.

To see the options:

```
java -jar stream-perf-test-{version}.jar --help
```

The performance tool comes also with a completion script. You can download it and enable it in your `~/.zshrc` file:

```
alias stream-perf-test='java -jar target/stream-perf-test.jar'
source ~/.zsh/stream-perf-test_completion
```

Note the activation requires an alias which must be `stream-perf-test`. The command can be anything though.

Building the Performance Tool

To build the uber JAR:

```
./mvnw clean package -Dmaven.test.skip -P performance-tool
```

Then run the tool:

```
java -jar target/stream-perf-test.jar
```