RabbitMQ Stream Java Client

Version 0.1.0-SNAPSHOT: (cac5981)

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?

RabbitMQ Stream was developed to cover the following messaging use cases:

- *Large fan-outs:* when several consumer applications need to read the same messages.
- *Replay / Time-traveling:* when consumer applications need to read the whole history of data or from a given point in a stream.
- *Throughput performance:* when higher throughput than with other protocols (AMQP, STOMP, MQTT) is required.
- *Large logs:* when large amount of data need to be stored, with minimal in-memory overhead.

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.

Message de-duplication is also supported on the publisher side.

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.
- *avoid publishing duplicate messages* thanks to message de-duplication.
- *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 minimize inter-node 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

There are different ways to make the broker visible to the client application when running in Docker. The next sections show a couple of options suitable for local development.

NOTE	*Docker on macOS* Docker runs on a virtual machine when using macOS, so do not expect high performance when using RabbitMQ Stream inside Docker on a Mac.

With Docker Bridge Network Driver

This section shows how to start a broker instance for local development (the broker Docker container and the client application are assumed to run on the same host).

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 5551:5551 \
    -e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS="-rabbitmq_stream advertised_host
localhost" \
    pivotalrabbitmq/rabbitmq-stream
```

The previous command exposes only the stream port (5551), you can expose ports for other protocols:

Exposing the AMQP 0.9.1 and management ports:

```
docker run -it --rm --name rabbitmq -p 5551:5551 -p 5672:5672 -p 15672:15672 \
    -e RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS="-rabbitmq_stream advertised_host
localhost" \
    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 Docker Host Network Driver

This is the simplest way to run the broker locally. The container uses the host network, this is perfect for experimenting locally.

Running RabbitMQ Stream with the host network driver

```
docker run -it --rm --network host pivotalrabbitmq/rabbitmq-stream
```

The command above will use the following ports: 5551 (for stream), 5672 (for AMQP), 15672 (for management plugin).

NOTE	*Docker Host Network Driver Support* The host networking driver **only works on Linux hosts**.

With the Generic Unix Package

The generic Unix package requires Erlang to be installed.

- Download the latest generic Unix `alpha-stream` archive.
- 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 5551.

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>0.33.8</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:0.33.8"
}

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 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:5551

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:5551/%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` and `Consumer` instances. 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://host1:5551",
                "rabbitmq-stream://host2:5551",
                "rabbitmq-stream://host3:5551")
        )
        .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 an `Environment`:

Parameter Name	Description	Default
`uri`	The URI of the node to connect to (single node).	`rabbitmq-stream://guest:guest@localhost:5551/%2f`
`uris`	The URI of the nodes to try to connect to (cluster).	`rabbitmq-stream://guest:guest@localhost:5551/%2f` singleton list
`host`	Host to connect to.	`localhost`
`port`	Port to use.	`5551`
`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());`
`maxProducersByConnection`	The maximum number of `Producer` instances a single connection can maintain before a new connection is open. The value must be between 1 and 255.	255
`maxCommittingConsumersByConnection`	The maximum number of `Consumer` instances that commit their offset a single connection can maintain before a new connection is open. The value must be between 1 and 255.	50
`maxConsumersByConnection`	The maximum number of `Consumer` instances a single connection can maintain before a new connection is open. The value must be between 1 and 255.	255

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 and time-based retention policies: once the stream reaches a given size or a given age, 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.

NOTE

When does the broker enforce the retention policy?

The broker enforces the retention policy when the segments of a stream roll over, that is when the current segment has reached its maximum size and is closed in favor of a new one. This means the maximum segment size is a critical setting in the retention mechanism.

RabbitMQ Stream also supports a time-based retention policy: segments get truncated when they reach a certain age. The following snippet illustrates how to set the time-based retention policy:

Setting a time-based retention policy when creating a stream

```
environment.streamCreator()
        .stream("my-stream")
        .maxAge(Duration.ofHours(6))  ①
        .maxSegmentSizeBytes(ByteCapacity.MB(500))  ②
        .create();
```

① Set the maximum age to 6 hours

② Set the segment size to 500 MB

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
`name`	The logical name of the producer. Specify a name to enable message de-deduplication.	`null` (no de-duplication)
`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 and potential duplicated messages even when de-duplication is enabled.	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
`confirmTimeout`	Time before the client calls the confirm callback to signal outstanding unconfirmed messages timed out.	30 seconds
`enqueueTimeout`	Time before enqueueing of a message fail when the maximum number of unconfirmed is reached. The callback of the message will be called with a negative status. Set the value to `Duration.ZERO` if there should be no timeout.	10 seconds.

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.

NOTE

Use a `MessageBuilder` instance only once

A `MessageBuilder` instance is meant to create only one message. You need to create a new instance of `MessageBuilder` for every message you want to create.

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.

WARNING

Keep the confirmation callback as short as possible

The confirmation callback should be kept as short as possible to avoid blocking the connection thread. Not doing so can make the `Environment`, `Producer`, `Consumer` instances sluggish or even block them. Any long processing should be done in a separate thread (e.g. with an asynchronous `ExecutorService`).

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.

Message De-deduplication

RabbitMQ Stream provides publisher confirms to avoid losing messages: once the broker has persisted a message it sends a confirmation for this message. But this can lead to duplicate messages: imagine the connection closes because of a network glitch after the message has been persisted but *before* the confirmation reaches the producer. Once reconnected, the producer will retry to send the same message, as it never received the confirmation. So the message will be

persisted twice.

Luckily RabbitMQ Stream can detect and filter out duplicated messages, based on 2 client-side elements: the *producer name* and the *message publishing ID*.

WARNING

De-duplication is not guaranteed when using sub-entries batching

It is not possible to guarantee de-duplication when sub-entry batching is in use. Sub-entry batching is disabled by default and it does not prevent from batching messages in a single publish frame, which can already provide very high throughput.

Setting the Name of a Producer

The producer name is set when creating the producer instance, which automatically enables de-duplication:

Naming a producer to enable message de-duplication

```
Producer producer = environment.producerBuilder()
    .name("my-app-producer")  ①
    .confirmTimeout(Duration.ZERO)  ②
    .stream("my-stream")
    .build();
```

① Set a name for the producer

② Disable confirm timeout check

Thanks to the name, the broker will be able to track the messages it has persisted on a given stream for this producer. If the producer connection unexpectedly closes, it will automatically recover and retry outstanding messages. The broker will then filter out messages it has already received and persisted. No more duplicates!

IMPORTANT

Why setting `confirmTimeout` to 0 when using de-duplication?

The point of de-duplication is to avoid duplicates when retrying unconfirmed messages. But why retrying in the first place? To avoid *losing* messages, that is enforcing *at-least-once* semantics. If the client does not stubbornly retry messages and gives up at some point, messages can be lost, which maps to *at-most-once* semantics. This is why the de-duplication examples set the `confirmTimeout` setting to `Duration.ZERO`: to disable the background task that calls the confirmation callback for outstanding messages that time out. This way the client will do its best to retry messages until they are confirmed.

Consider the producer name a logical name. It should not be a random sequence that changes when the producer application is restarted. Names like `online-shop-order` or `online-shop-invoice` are better names than `3d235e79-047a-46a6-8c80-9d159d3e1b05`. There should be only one living instance of a producer with a given name on a given stream at the same time.

Understanding Publishing ID

The producer name is only one part of the de-duplication mechanism, the other part is the *message publishing ID*. If the producer has a name, the client automatically assigns a publishing ID to each outbound message for the producer. The publishing ID is a monotonic sequence, starting at 0 and incremented for each message. The default publishing sequence is good enough for de-duplication, but it is possible to assign a publishing ID to each message:

Using an explicit publishing ID

```
Message message = producer.messageBuilder()
    .publishingId(1)  ①
    .addData("hello".getBytes(StandardCharsets.UTF_8))
    .build();
producer.send(message, confirmationStatus -> { });
```

① Set a publishing ID on a message

There are a few rules to follow when using a custom publishing ID sequence:

- the sequence should start at 0
- the sequence must be monotonic, that is always increments
- there can be gaps in the sequence (e.g. 0, 1, 2, 3, 6, 7, 9, 10, etc)

A custom publishing ID sequence has usually a meaning: it can be the line number of a file or the primary key in a database.

Note the publishing ID is not part of the message: it is not stored with the message and so is not available when consuming the message. It is still possible to store the value in the AMQP 1.0 message application properties or in an appropriate properties (e.g. `messageId`).

IMPORTANT	*Do not mix client-assigned and custom publishing ID* As soon as a producer name is set, message de-duplication is enabled. It is then possible to let the producer assign a publishing ID to each message or assign custom publishing IDs. **Do one or the other, not both!**

Restarting a Producer Where It Left Off

Using a custom publishing sequence is even more useful to restart a producer where it left off. Imagine a scenario whereby the producer is sending a message for each line in a file and the application uses the line number as the publishing ID. If the application restarts because of some necessary maintenance or even a crash, the producer can restart from the beginning of the file: there would no duplicate messages because the producer has a name and the application sets publishing IDs appropriately. Nevertheless, this is far from ideal, it would be much better to restart just after the last line the broker successfully confirmed. Fortunately this is possible thanks to the `Producer#getLastPublishing()` method, which returns the last publishing ID for a given producer. As the publishing ID in this case is the line number, the application can easily scroll to the next line and restart publishing from there.

The next snippet illustrates the use of `Producer#getLastPublishing()`:

Setting a producer where it left off

```
Producer producer = environment.producerBuilder()
    .name("my-app-producer")  ①
    .confirmTimeout(Duration.ZERO)  ②
    .stream("my-stream")
    .build();
long nextPublishingId = producer.getLastPublishingId() + 1;  ③
while (moreContent(nextPublishingId)) {
    byte[] content = getContent(nextPublishingId); ④
    Message message = producer.messageBuilder()
        .publishingId(nextPublishingId) ⑤
        .addData(content)
        .build();
    producer.send(message, confirmationStatus -> {});
    nextPublishingId++;
}
```

① Set a name for the producer

② Disable confirm timeout check

③ Query last publishing ID for this producer and increment it

④ Scroll to the content for the next publishing ID

⑤ Set the message publishing

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.

WARNING	*Keep the message processing callback as short as possible* The message processing callback should be kept as short as possible to avoid blocking the connection thread. Not doing so can make the `Environment`, `Producer`, `Consumer` instances sluggish or even block them. Any long processing should be done in a separate thread (e.g. with an asynchronous `ExecutorService`).

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.

NOTE	*What is a chunk of messages?* A chunk is simply a batch of messages. This is the storage and transportation unit used in RabbitMQ Stream, that is messages are stored contiguously in a chunk and they are delivered as part of a chunk. A chunk can be made of one to several thousands of messages, depending on the ingress.

The following figure shows the different offset specifications in a stream made of 2 chunks:

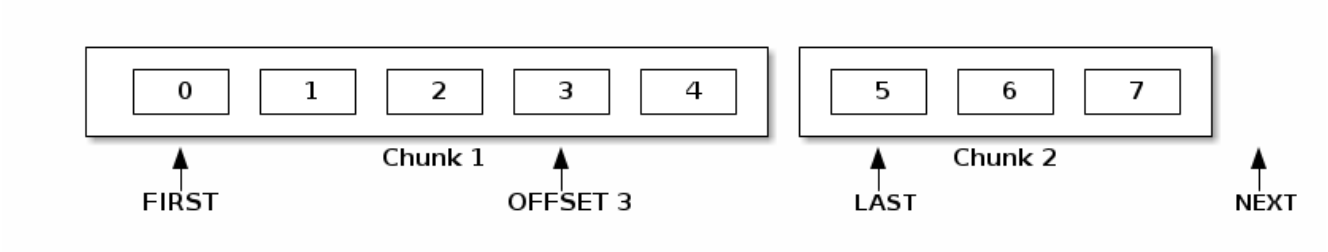

Figure 1. Offset specifications in a stream made of 2 chunks

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 usable as an

uber JAR downloadable from GitHub Release or as a Docker image. It can be built separately as well.

Using the Performance Tool

With Docker

The performance tool is available as a Docker image. You can use the Docker image to list the available options:

Listing the available options of the performance tool

```
docker run -it --rm pivotalrabbitmq/stream-perf-test --help
```

There are all sorts of options, if none is provided, the tool will start publishing to and consuming from a stream created only for the test.

When using Docker, the container running the performance tool must be able to connect to the broker, so you have to figure out the appropriate Docker configuration to make this possible. You can have a look at the Docker network documentation to find out more.

NOTE

Docker on macOS

Docker runs on a virtual machine when using macOS, so do not expect high performance when using RabbitMQ Stream and the performance tool inside Docker on a Mac.

We show next a couple of options to easily use the Docker image.

With Docker Host Network Driver

This is the simplest way to run the image locally, with a local broker running in Docker as well. The containers use the host network, this is perfect for experimenting locally.

Running the broker and performance tool with the host network driver

```
# run the broker
docker run -it --rm --network host pivotalrabbitmq/rabbitmq-stream
# open another terminal and run the performance tool
docker run -it --rm --network host pivotalrabbitmq/stream-perf-test
```

NOTE

Docker Host Network Driver Support

According to Docker's documentation, the host networking driver **only works on Linux hosts**. Nevertheless, the commands above work on some Mac hosts.

With Docker Bridge Network Driver

Containers need to be able to communicate with each other with the bridge network driver, this

can be done by defining a network and running the containers in this network.

Running the broker and performance tool with the bridge network driver

```
# create a network
docker network create stream-perf-test
# run the broker
docker run -it --rm --network stream-perf-test --name rabbitmq
pivotalrabbitmq/rabbitmq-stream
# open another terminal and run the performance tool
docker run -it --rm --network stream-perf-test pivotalrabbitmq/stream-perf-test \
    --uris rabbitmq-stream://rabbitmq:5551
```

With the Java Binary

The Java binary is on GitHub Release:

```
wget https://github.com/rabbitmq/rabbitmq-java-tools-binaries-dev/releases/download/v-
stream-perf-test-latest/stream-perf-test-latest.jar
```

To launch a run:

```
$ java -jar stream-perf-test-latest.jar
17:51:26.207 [main] INFO  c.r.stream.perf.StreamPerfTest - Starting producer
1, published 560277 msg/s, confirmed 554088 msg/s, consumed 556983 msg/s, latency
min/median/75th/95th/99th 2663/9799/13940/52304/57995 µs, chunk size 1125
2, published 770722 msg/s, confirmed 768209 msg/s, consumed 768585 msg/s, latency
min/median/75th/95th/99th 2454/9599/12206/23940/55519 µs, chunk size 1755
3, published 915895 msg/s, confirmed 914079 msg/s, consumed 916103 msg/s, latency
min/median/75th/95th/99th 2338/8820/11311/16750/52985 µs, chunk size 2121
4, published 1004257 msg/s, confirmed 1003307 msg/s, consumed 1004981 msg/s, latency
min/median/75th/95th/99th 2131/8322/10639/14368/45094 µs, chunk size 2228
5, published 1061380 msg/s, confirmed 1060131 msg/s, consumed 1061610 msg/s, latency
min/median/75th/95th/99th 2131/8247/10420/13905/37202 µs, chunk size 2379
6, published 1096345 msg/s, confirmed 1095947 msg/s, consumed 1097447 msg/s, latency
min/median/75th/95th/99th 2131/8225/10334/13722/33109 µs, chunk size 2454
7, published 1127791 msg/s, confirmed 1127032 msg/s, consumed 1128039 msg/s, latency
min/median/75th/95th/99th 1966/8150/10172/13500/23940 µs, chunk size 2513
8, published 1148846 msg/s, confirmed 1148086 msg/s, consumed 1149121 msg/s, latency
min/median/75th/95th/99th 1966/8079/10135/13248/16771 µs, chunk size 2558
9, published 1167067 msg/s, confirmed 1166369 msg/s, consumed 1167311 msg/s, latency
min/median/75th/95th/99th 1966/8063/9986/12977/16757 µs, chunk size 2631
10, published 1182554 msg/s, confirmed 1181938 msg/s, consumed 1182804 msg/s, latency
min/median/75th/95th/99th 1966/7963/9949/12632/16619 µs, chunk size 2664
11, published 1197069 msg/s, confirmed 1196495 msg/s, consumed 1197291 msg/s, latency
min/median/75th/95th/99th 1966/7917/9955/12503/15386 µs, chunk size 2761
12, published 1206687 msg/s, confirmed 1206176 msg/s, consumed 1206917 msg/s, latency
min/median/75th/95th/99th 1966/7893/9975/12503/15280 µs, chunk size 2771
...
^C
Summary: published 1279444 msg/s, confirmed 1279019 msg/s, consumed 1279019 msg/s,
latency 95th 12161 µs, chunk size 2910
```

The previous command will start publishing to and consuming from a `stream1` 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-latest.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.

Common Usage

Connection

The performance tool connects by default to localhost, on port 5551, with default credentials (`guest`/`guest`), on the default `/` virtual host. This can be changed with the `--uris` option:

```
java -jar stream-perf-test.jar --uris rabbitmq-stream://rabbitmq-1:5551
```

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. The next command shows how to set up the different elements of the URI:

```
java -jar stream-perf-test.jar \
  --uris rabbitmq-stream://guest:guest@localhost:5551/%2f
```

The option accepts several values, separated by commas. By doing so, the tool will be able to pick another URI for its "locator" connection, in case a node crashes:

```
java -jar stream-perf-test.jar \
  --uris rabbitmq-stream://rabbitmq-1:5551,rabbitmq-stream://rabbitmq-2:5551
```

Note the tool uses those URIs only for management purposes, it does not use them to distribute publishers and consumers across a cluster.

Publishing Rate

It is possible to limit the publishing rate with the `--rate` option:

```
java -jar stream-perf-test.jar --rate 10000
```

RabbitMQ Stream can easily saturate the resources of the hardware, it can especially max out the storage IO. Reasoning when a system is under severe constraints can be difficult, so setting a low publishing rate can be a good idea to get familiar with the performance tool and the semantics of streams.

Number of Producers and Consumers

You can set the number of producers and consumers with the `--producers` and `--consumers` options, respectively:

```
java -jar stream-perf-test.jar --producers 5 --consumers 5
```

With the previous command, you should see a higher consuming rate than publishing rate. It is because the 5 producers publish as fast as they can and each consumer consume the messages from the 5 publishers. In theory the consumer rate should be 5 times the publishing rate, but as stated previously, the performance tool may put the broker under severe constraints, so the numbers may not add up.

You can set a low publishing rate to verify this theory:

```
java -jar stream-perf-test.jar --producers 5 --consumers 5 --rate 10000
```

With the previous command, each publisher should publish 10,000 messages per second, that is 50,000 messages per second overall. As each consumer consumes each published messages, the consuming rate should be 5 times the publishing rate, that is 250,000 messages per second. Using a small publishing rate should let plenty of resources to the system, so the rates should tend towards those values.

Streams

The performance tool uses a `stream1` stream by default, the `--streams` option allows specifying streams that will be created and used only for the test run. Note producer and consumer counts must be set accordingly, as they are not spread across the stream automatically. The following command will run a test with 3 streams, with a producer and a consumer on each of them:

```
java -jar stream-perf-test.jar --streams stream1,stream2,stream3 \
                               --producers 3 --consumers 3
```

If you do not want the tool to create and delete streams for a run, because they are already created, use the `--pre-declared` option:

```
java -jar stream-perf-test.jar --streams stream1,stream2,stream3 \
                               --producers 3 --consumers 3 \
                               --pre-declared
```

Specifying streams one by one can become tedious as their number grows, so the `--stream-count` option can be combined with the `--streams` option to specify a number or a range and a stream name pattern, respectively. The following table shows the usage of these 2 options and the resulting exercised streams. Do not forget to also specify the appropriate number of producers and consumers if you want all the declared streams to be used.

Options	Computed Streams	Details
`--stream-count 5 --streams stream`	`stream-1,stream-2,stream-3,stream-4,stream-5`	Stream count starts at 1.
`--stream-count 5 --streams stream-%d`	`stream-1,stream-2,stream-3,stream-4,stream-5`	Possible to specify a Java printf-style format string.

Options	Computed Streams	Details
`--stream-count 10 --streams stream-%d`	`stream-1,stream-2,stream-3,⋯, stream-10`	Not bad, but not correctly sorted alphabetically.
`--stream-count 10 --streams stream-%02d`	`stream-01,stream-02,stream-03,⋯, stream-10`	Better for sorting.
`--stream-count 10 --streams stream`	`stream-01,stream-02,stream-03,⋯, stream-10`	The default format string handles the sorting issue.
`--stream-count 50-500 --streams stream-%03d`	`stream-050,stream-051,stream-052,⋯, stream-500`	Ranges are accepted.
`--stream-count 50-500`	`stream-050,stream-051,stream-052,⋯, stream-500`	Default format string.

Publishing Batch Size

The default publishing batch size is 100, that is a publishing frame is sent every 100 messages. The following command sets the batch size to 50 with the `--batch-size` option:

```
java -jar stream-perf-test.jar --batch-size 50
```

There is no ideal batch size, it is a tradeoff between throughput and latency. High batch size values should increase throughput (usually good) and latency (usually not so good), whereas low batch size should decrease throughput (usually not good) and latency (usually good).

Unconfirmed Messages

A publisher can have at most 10,000 unconfirmed messages at some point. If it reaches this value, it has to wait until the broker confirms some messages. This avoids fast publishers overwhelming the broker. The `--confirms` option allows changing the default value:

```
java -jar stream-perf-test.jar --confirms 20000
```

High values should increase throughput at the cost of consuming more memory, whereas low values should decrease throughput and memory consumption.

Message Size

The default size of a message is 10 bytes, which is rather small. The `--size` option lets you specify a different size, usually higher, to have a value close to your use case. The next command sets a size of 1 KB:

```
java -jar stream-perf-test.jar --size 1024
```

Note the message body size cannot be smaller that 8 bytes, as the performance tool stores a long in each message to calculate the latency. Note also the actual size of a message will be slightly higher, as the body is wrapped in an AMQP 1.0 message.

Advanced Usage

Retention

If you run performance tests for a long time, you might be interested in setting a retention strategy for the streams the performance tool creates for a run. This would typically avoid saturating the storage devices of your servers. The default values are 20 GB for the maximum size of a stream and 500 MB for each segment files that composes a stream. You can change these values with the `--max-length-bytes` and `--max-segment-size` options:

```
java -jar stream-perf-test.jar --max-length-bytes 10gb \
                               --max-segment-size 250mb
```

Both options accept units (`kb`, `mb`, `gb`, `tb`), as well as no unit to specify a number of bytes.

It is also possible to use the time-based retention strategy with the `--max-age` option. This can be less predictable than `--max-length-bytes` in the context of performance tests though. The following command shows how to set the maximum age of segments to 5 minutes with a maximum segment size of 250 MB:

```
java -jar stream-perf-test.jar --max-age PT5M \
                               --max-segment-size 250mb
```

The `--max-age` option uses the ISO 8601 duration format.

Offset (Consumer)

Consumers start by default at the first available offset of stream, that is the beginning of the stream if it has not been truncated (offset 0). It is possible to specify an offset to start from with the `--offset` option, if you have existing streams and you want to consume from them at a specific offset. The following command sets the consumer to start consuming at the very end of a stream, as soon as new messages are published:

```
java -jar stream-perf-test.jar --offset next
```

The accepted values for `--offset` are `first` (the default), `last`, `next`, an unsigned long for a given offset, and an ISO 8601 formatted timestamp (eg. `2020-06-03T07:45:54Z`).

Offset Tracking (Consumer)

A consumer can track the point it has reached in a stream to be able to restart where it left off in a new incarnation. The performance tool has the `--commit-every` option to tell consumers to commit the offset every `x` messages to be able to measure the impact of offset tracking in terms of throughput and storage. This feature is disabled by default. The following command shows how to commit the offset every 100,000 messages:

```
java -jar stream-perf-test.jar --commit-every 100000
```

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
```