Direct Publisher -> Processor -> Subscriber

The next tutorial in our series of sample applications will address some basic patterns for event-driven applications: publishing (sending messages), subscribing (receiving messages), and a specific combination of the two. The term “processor” is pulled from the Spring Framework, where it denotes a microservice that receives a message, does some processing / annotation / decoration / validation upon it, and publishes a new message as a result.

Most real-world event-driven applications and microservices employ some combination of publishing and subscribing; it is rare for an app to exclusively publish or exclusively subscribe. For example:

* Mainly publishing applications (financial market data feeds, or sensor gateways) might subscribe to health check or presence topics, or command-and-control topics
* Mainly subscribing applications (database updaters, stats/metrics dashboard GUIs) might publish occasional logging information or metrics of how they are currently operating

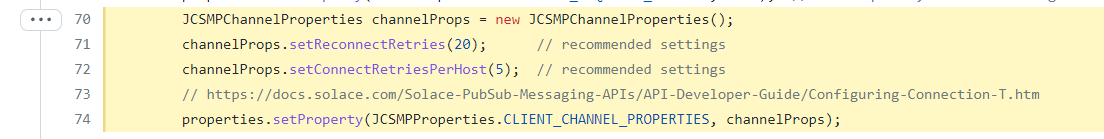
Once we begin to look at request-reply patterns and samples, those applications by design must be able to both publish and subscribe.

**Please ensure you have reviewed the HelloWorld sample and tutorial to start**, as this tutorial continues to build on first principles introduced there, and will introduce a few more features and concepts. The Direct Pub and Sub samples are intended to be more performant examples: they do not print every message to the screen or perform heavy calculations in the message callback, and should be able to run at very high message rates if the thread sleep is removed.

## Reconnection Settings

It is inevitable that your application will at some point lose network connectivity to the Solace PubSub+ broker. This occurs due to networking issues or broker failures. The Solace APIs are designed to reconnect back to the broker automatically, and the application does not have to code for this. However, there are a number of settings that can be adjusted to help ensure your application has the best chance to reconnect.

Typically, during a broker failover from its primary node to standby node in an HA Pair, it should usually take around 15-20 seconds. In extreme cases, it is possible to take up to a minute. However, due to misconfigurations, bugs, networking or container management issues, it might be possible to take longer. Solace recommends configuring the automatic reconnect timers to take up to 5 minutes to err on the side of caution.

[](https://github.com/SolaceSamples/solace-samples-java-jcsmp/blob/master/src/main/java/com/solace/samples/jcsmp/patterns/DirectPublisher.java#L70-L74)

Note that a setting of -1 means “retry forever”, which can sometimes be appropriate. However, other times it is a better practice to have the automatic reconnects eventually timeout, and have control pass back to the application where it can restart the connection process again if it wants to.

## Message Headers and Metadata

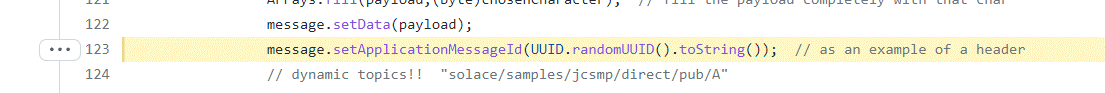
Solace messages can contain a lot of additional information besides their topic and payload; messages are not simply key/value pairs of data like in some other products. For example, here is a *partial* list of additional fields and values you can set as part of a message:

* Sender Timestamp: a Long, milliseconds since epoch
* Application Message ID: a String, useful as a unique identifier
* Sequence Number: a Long
* Sender ID: a String, usually to indicate the publisher application
* TTL (Time to Live): a Long, in milliseconds, for Guaranteed messages expiry
* DMQ-Eligible (Dead Letter Queue Eligible): a Boolean flag, for whether an expired or retry-exhausted message should be discarded or moved to a special error-handling queue

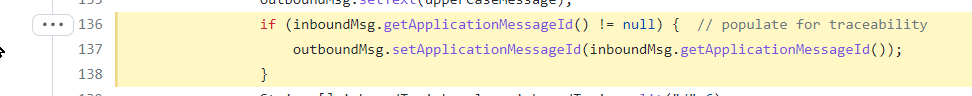
These various flags and values can be used by the Solace broker or consuming client applications to alter behaviour about how to process the message. For a full listing of values, consult the documentation.

These samples will set the Application Message ID as one example. This is typically used to give each message a unique ID to help consuming applications know/verify if they have received the same message twice due to failures/retransmission. In the case of the Processor application, it maintains the original Application Message ID to help with traceability: it can be used to group a “flow” or “trace” of related messages together.

As an example, in the Publisher:

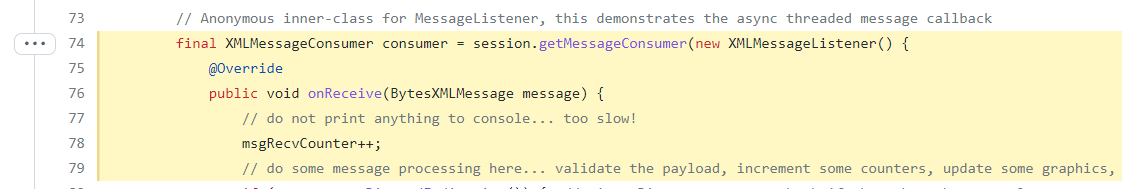
[](https://github.com/SolaceSamples/solace-samples-java-jcsmp/blob/master/src/main/java/com/solace/samples/jcsmp/patterns/DirectPublisher.java#L123)

And the Processor reuses the same Application Message ID:

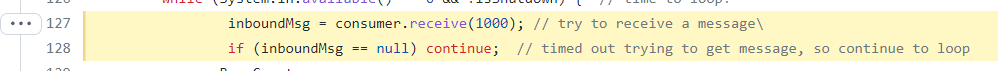
[](https://github.com/SolaceSamples/solace-samples-java-jcsmp/blob/master/src/main/java/com/solace/samples/jcsmp/patterns/DirectProcessor.java#L136-L138)

## Blocking vs. Asynchronous Receive

When receiving messages from Solace PubSub+ event brokers, all Solace APIs can receive messages *asynchronously* via a message callback; the callback is a method that an application implements, which the API will run whenever a message is received and is passed to the application for handling (further processing). The thread on which the callback method is called belongs to the API, and it is generally considered best practice to return quickly from the method to avoid blocking other API functionality.



Some APIs (e.g. JCSMP, ..??) also allow for a *blocking* or synchronous receive call from an application thread. This simplified way of asking for a message can either block indefinitely, or timeout after a certain number of milliseconds. Not every Solace API has the ability to a blocking receive: for example, the C API (CCSMP), C#, JavaScript, and NodeJS APIs currently only allow for non-blocking callback receive. This is due to the threading models of the API and/or language that don’t prefer or allow for blocking behavior.

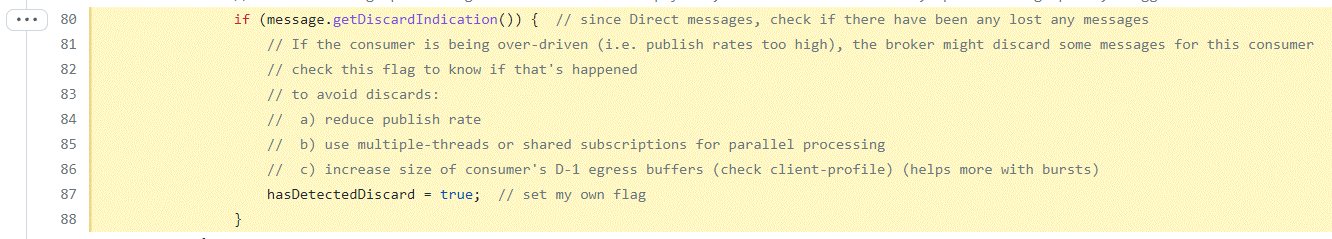


See <https://docs.solace.com/Solace-PubSub-Messaging-APIs/API-Developer-Guide/Receiving-Direct-Message.htm> for more details.

## Message Discard Indication

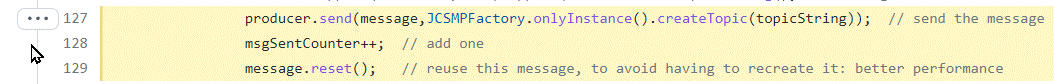
This message flag is typically used by very high-performance Direct messaging applications that want to ensure that the broker is not dropping or discarding any messages prior to delivering to this application. This can occur if the subscribing application is too slow or is overloaded, and the broker runs out of buffer resources for this client (called “TCP Egress Worker Queues”).

It is not strictly necessary to check this flag, but it can be useful to indicate to the subscribing application that it has lost data and might need to refresh its state from a known golden-source.



## Message Reuse On Publish

In the “classic” Solace APIs where Message objects are mutable, it is often desired for performance reasons to reuse message objects after sending, rather than recreating them.



See <https://docs.solace.com/Solace-PubSub-Messaging-APIs/API-Developer-Guide/Creating-Messages.htm#publishing_direct_messages_2102779381_604482> for more information.

Note that in the NextGen Solace APIs (e.g. Java, Python, Go) the Message objects are designed to be immutable and constructed using the Builder pattern, so this is not applicable.