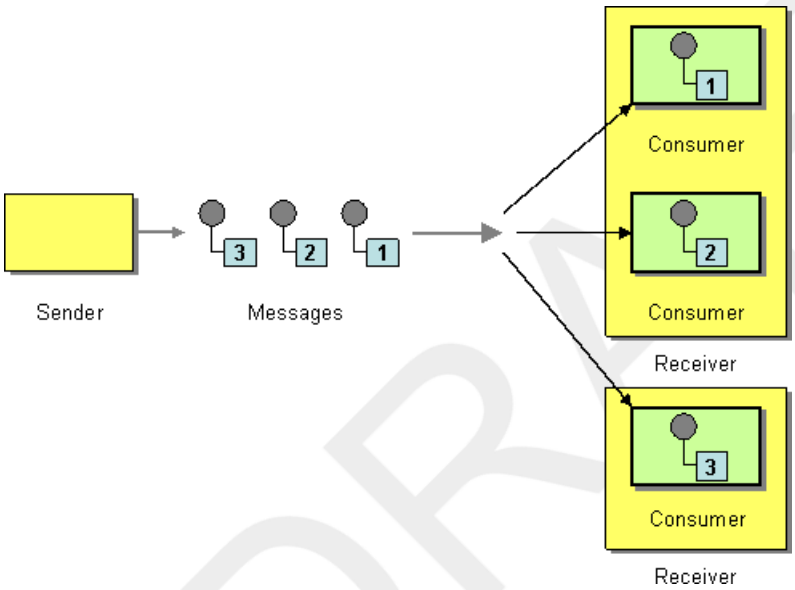
Competing Consumers

Overview

The *competing consumers* pattern enables multiple consumers to pull messages off the same queue, with the guarantee that *each message is consumed once only*. This pattern can therefore be used to replace serial message processing with concurrent message processing (bringing a corresponding reduction in response latency).

Figure 28. Competing Consumers Pattern



The following components demonstrate the competing consumers pattern:

- JMS based competing consumers on page 91
- SEDA based competing consumers on page 92

JMS based competing consumers

A regular JMS queue implicitly guarantees that each message can be consumed at most once. Hence, a JMS queue automatically supports the competing consumers pattern. For example, you could define three competing consumers that pull messages from the JMS queue, `HighVolumeQ`, as follows:

```
from("jms:HighVolumeQ").to("cxf:bean:replica01");
from("jms:HighVolumeQ").to("cxf:bean:replica02");
from("jms:HighVolumeQ").to("cxf:bean:replica03");
```

Where the CXF (Web services) endpoints, `replica01`, `replica02`, and `replica03`, process messages from the `HighVolumeQ` queue in parallel.

Note

JMS topics *cannot* support the competing consumers pattern. By definition, a JMS topic is intended to send multiple copies of the same message to different consumers. It is, therefore, incompatible with the competing consumers pattern.

SEDA based competing consumers

The purpose of the SEDA component to simplify concurrent processing by breaking the computation up into stages. A SEDA endpoint essentially encapsulates an in-memory blocking queue (implemented by `java.util.concurrent.BlockingQueue`). You can, therefore, use a SEDA endpoint to break a route up into stages, where each stage might use multiple threads. For example, you can define a SEDA route consisting of two stages, as follows:

```
// Stage 1: Read messages from file system.
from("file://var/messages").to("seda:fanout");

// Stage 2: Perform concurrent processing (3 threads).
from("seda:fanout").to("cxf:bean:replica01");
from("seda:fanout").to("cxf:bean:replica02");
from("seda:fanout").to("cxf:bean:replica03");
```

Where the first stage contains a single thread that consumes message from a file endpoint, `file://var/messages`, and routes them to a SEDA endpoint, `seda:fanout`. The second stage contains three threads: a thread that routes exchanges to `cxf:bean:replica01`, a thread that routes exchanges to `cxf:bean:replica02`, and a thread that routes exchanges to `cxf:bean:replica03`. These three threads compete to take exchange instances from the SEDA endpoint, which is implemented using a blocking queue. Because the blocking queue uses locking to prevent more than one thread accessing the queue at a time, you are guaranteed that each exchange instance is consumed at most once.