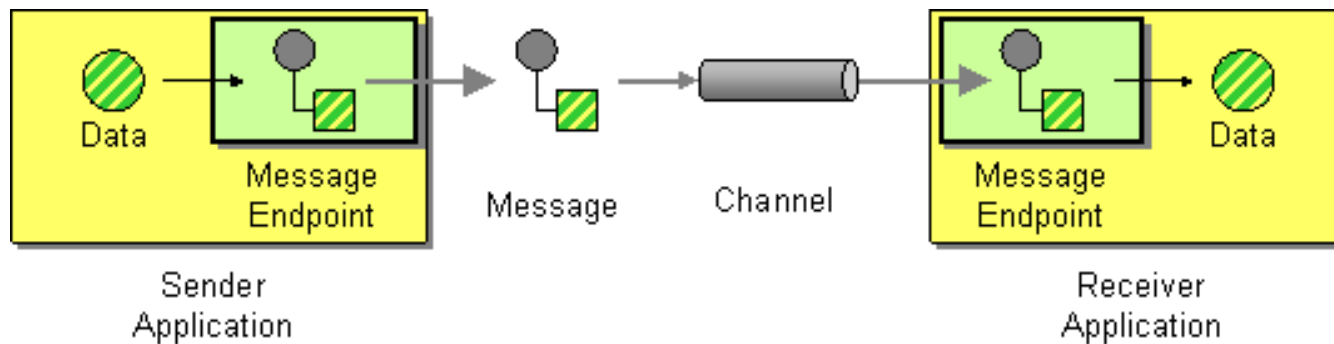


# Message Endpoint

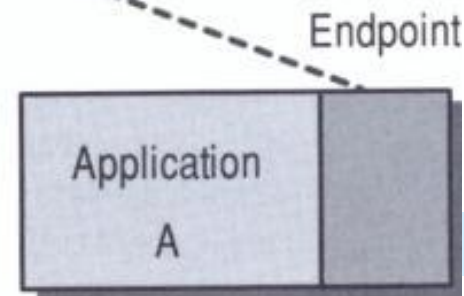
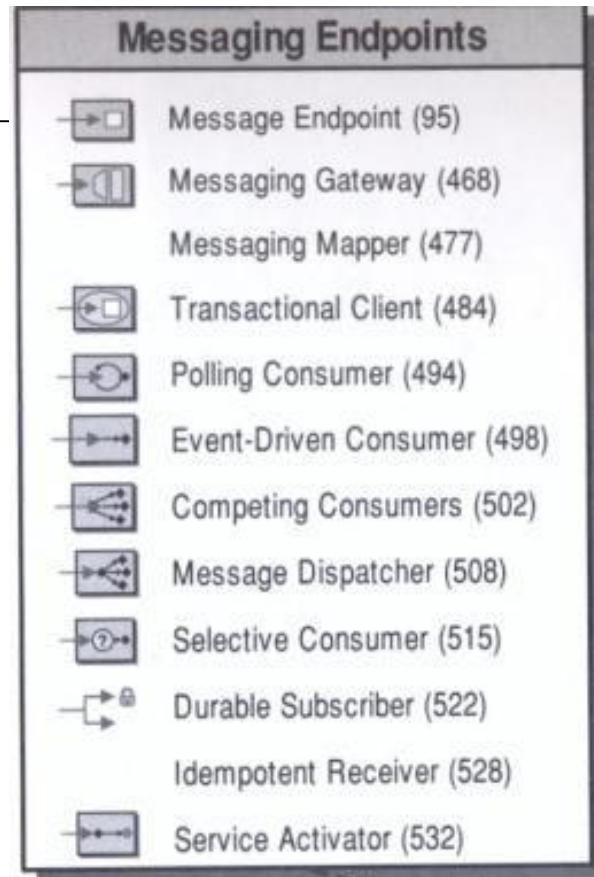
---

- How does an application connect to a messaging channel to send and receive messages?



- Use a *Message Endpoint*, i.e. a **client** of the messaging system that the application can use to send or receive messages.

# Pattern Overview



# Types of messaging clients

---

- Endpoint code you write yourself to a messaging API (e.g. MSMQ)
- Libraries and tools in commercial middleware packages

Two types of messaging endpoint patterns:

- Patterns that relate to both Send and Receive
- Patterns that relate to Message Consumption

# Patterns for Both Send and Receive

---

- Encapsulate the messaging code
  - a thin layer of code performs the application's part of the integration  
*Messaging Gateway* (468)
- Data translation
  - *Messaging Mapper* (477) to convert data between the application format and the message format
- Externally-controlled transactions
  - *Transactional Client* (484) to control transactions externally

# Message Consumer Patterns

---

- Synchronous or asynchronous consumer?
  - *Polling Consumer* (494)
  - *Event-Driven Consumer* (498)
- Message grab versus message assignment ?
  - *Competing Consumers* (502)
  - *Message Dispatcher* (508)
- Accept all messages or filter?
  - *Selective Consumer* (515)
- Subscribe while disconnected
  - *Durable Subscriber* (522)
- Idempotency
  - *Idempotent Receiver* (528)
- Synchronous or asynchronous service
  - *Service Activator* (532)

# Message Consumer Challenge

---

Receiving messages is the tricky part!

*Throttling*

Application ability to control the rate at which it consumes messages

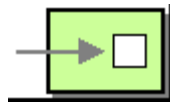
- Consumer can't control rate at which clients **send** requests
- Consumer can control the rate at which it **processes** those requests
  - Queue up
  - Concurrent message consumers

# Endpoint Concepts in EIP book

---

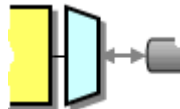
- Often difficult to tell these concepts apart

Endpoint

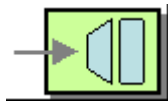


An **message endpoint** is a **specialized channel adapter** custom developed for, and integrated into the application.

Adapter



Gateway

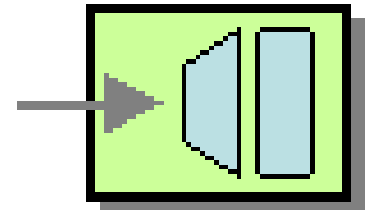


A **message endpoint** should be **designed** as a **message gateway** to encapsulate message code

# Send & Receive patterns

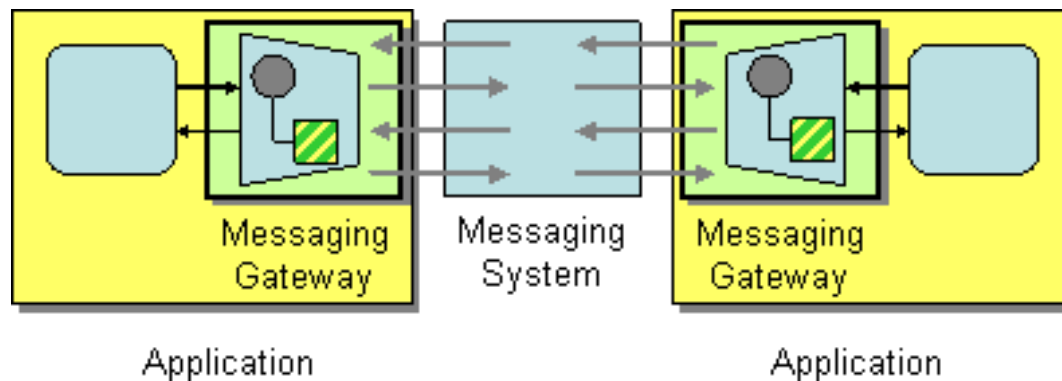


# Messaging Gateway



**Encapsulates** messaging-specific code

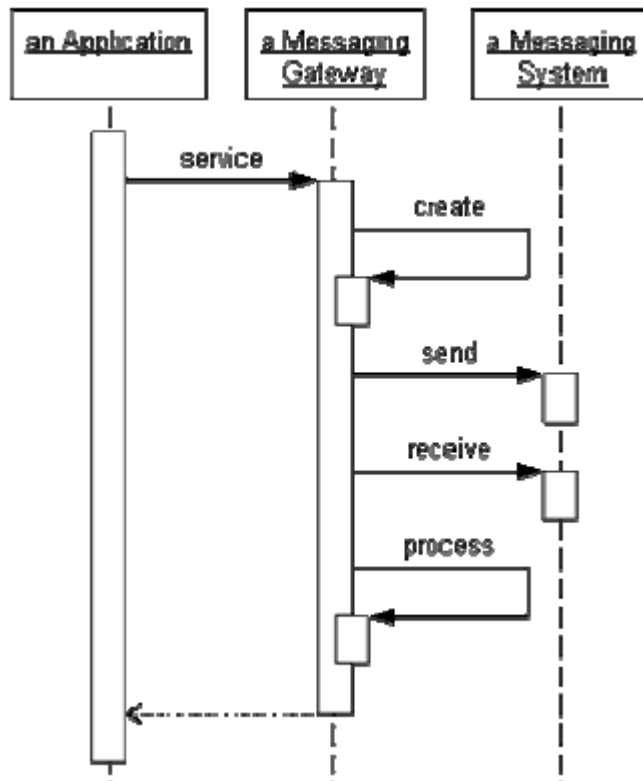
**Separates** it from the rest of the application code



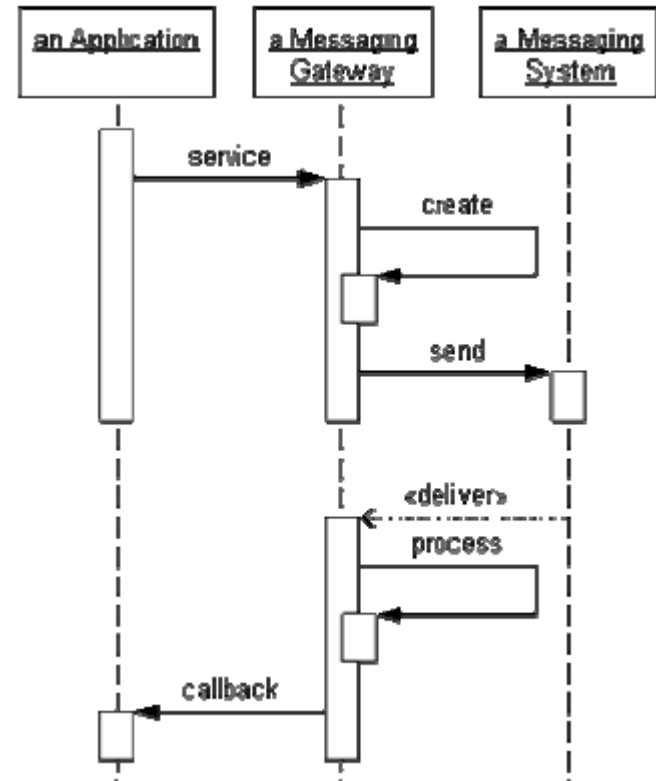
Only Messaging Gateway code knows about messaging →  
Makes it possible to swap out the gateway with a different implementation ☺

# Gateway request-reply example

## 1. Blocking (Synchronous)



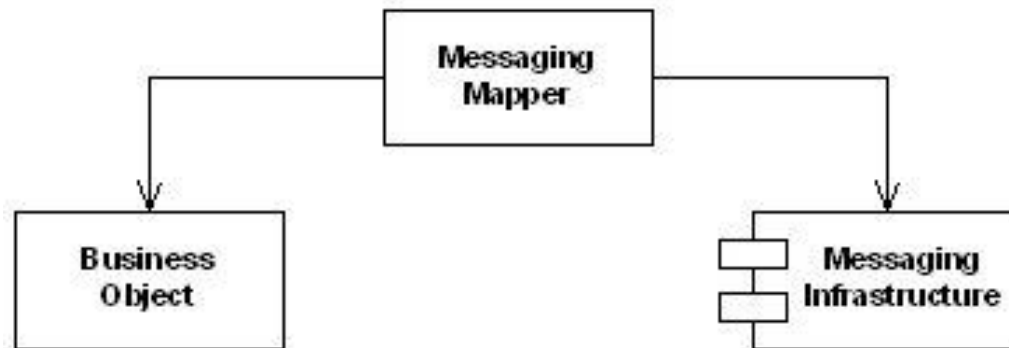
## 2. Event-Driven (Asynchronous)



# Messaging Mapper (477)

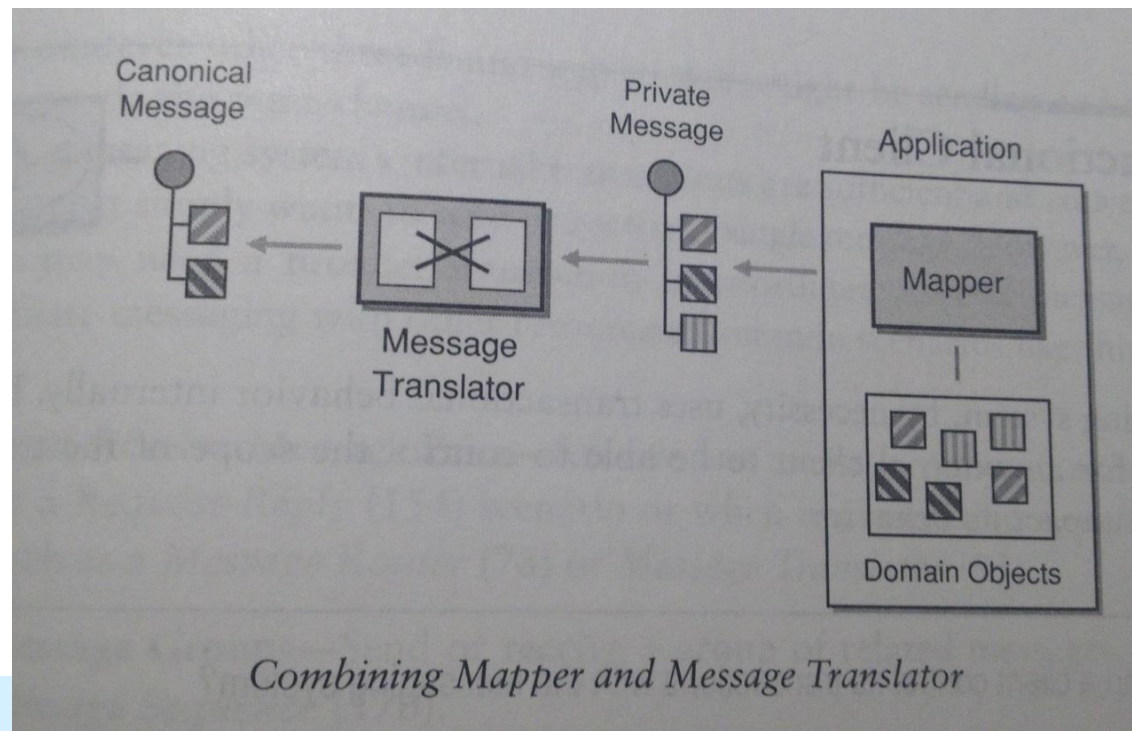
---

- How to move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
- Create a separate *Messaging Mapper* that contains the mapping logic between the messaging infrastructure and the domain objects. Neither the objects nor the infrastructure have knowledge of the Messaging Mapper's existence

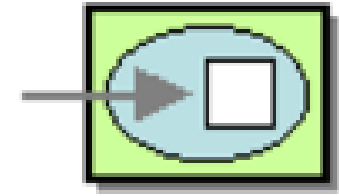


# Combining Mapper and Translator?

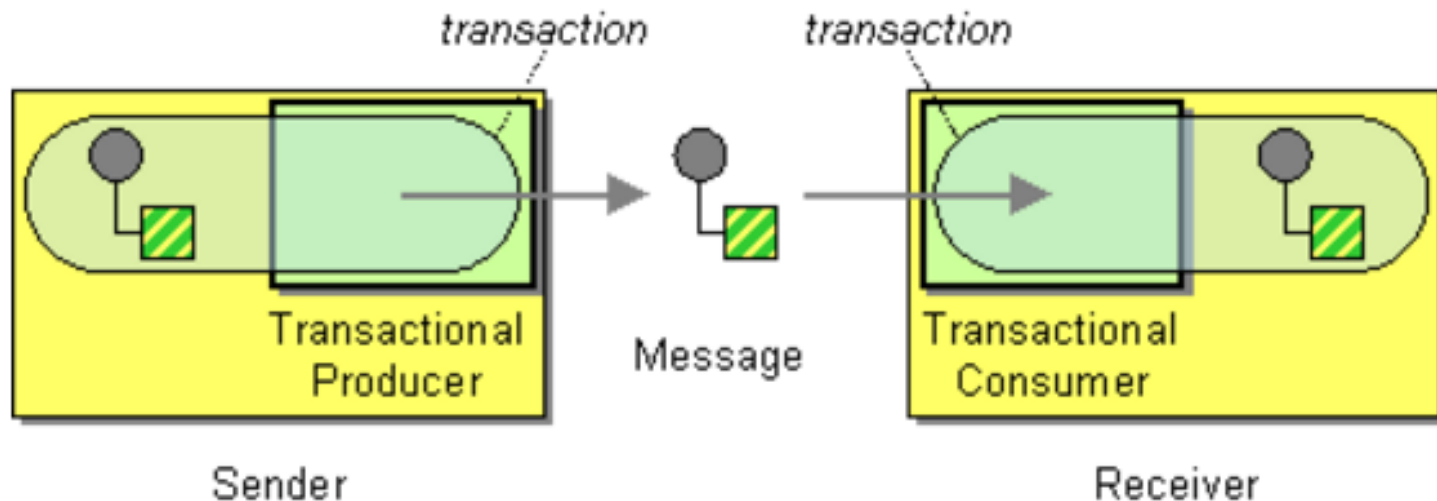
- **Message Translator:** Structural mappings inside messaging layer
- **Messaging Mapper:** Object references, data type conversion etc.
- Illustration of combination of these patterns (EIP p. 483)



# Transactional Client (484)



- Can a client control its transactions with the messaging system?



- **Sender:** the message isn't "really" added to the channel until the sender commits the transaction.
- **Receiver:** the message isn't "really" removed from the channel until the receiver commits the transaction.

# RabbitMQ Example – receiver commit

---

- RabbitMQ supports message *acknowledgements*
- Message acknowledgment is turned on by default
- We must explicitly turned them off with

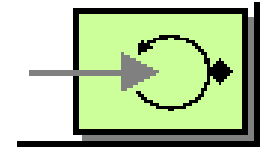
autoAck=true flag

```
QueueingConsumer consumer = new QueueingConsumer(channel);
boolean autoAck = false;
channel.basicConsume("hello", autoAck, consumer);

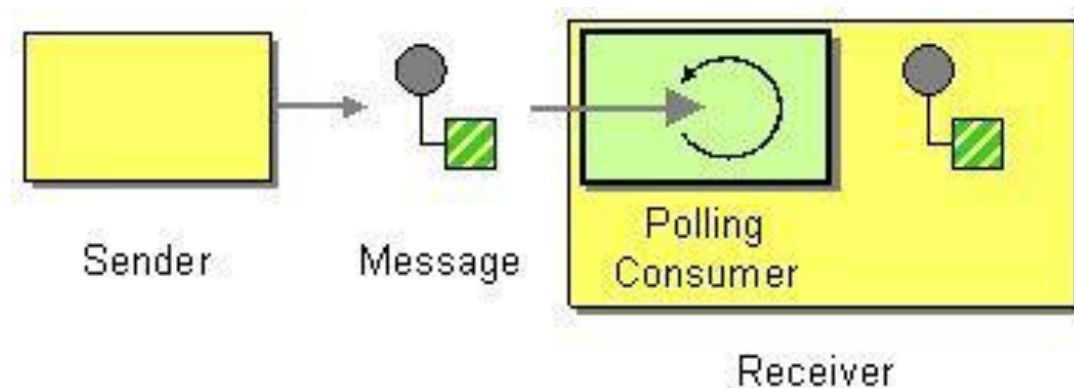
while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    //...
    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
}
```

# Consumer patterns

# Polling Consumer (494)



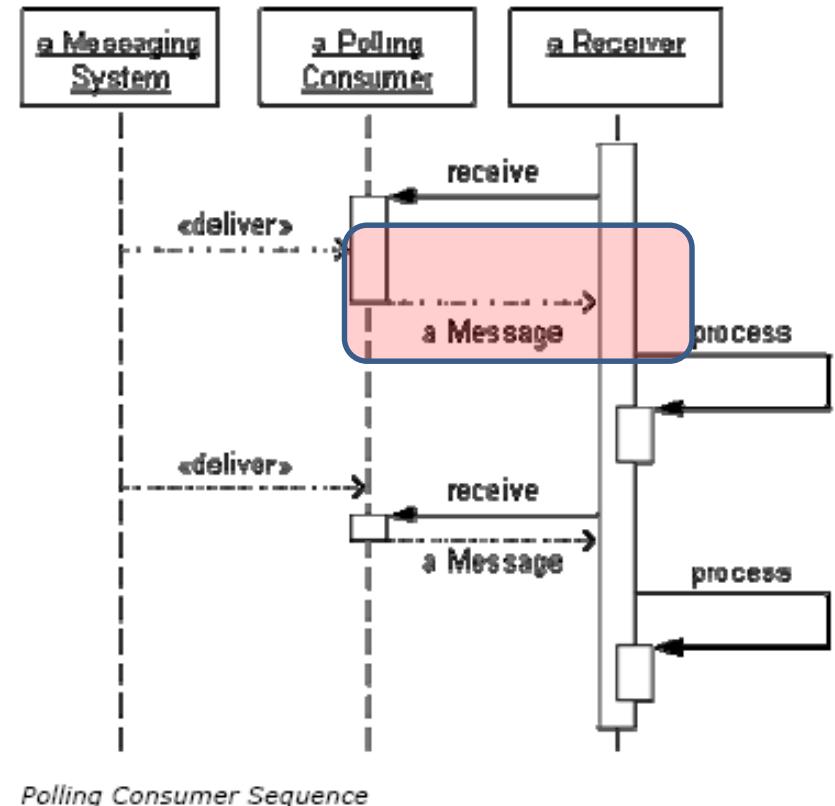
- A polling consumer **actively** checks for new messages
- A polling consumer won't poll for more messages until it has finished processing the current message, i.e. when it is ready!



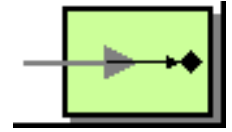


# Polling Consumer Flow - Example

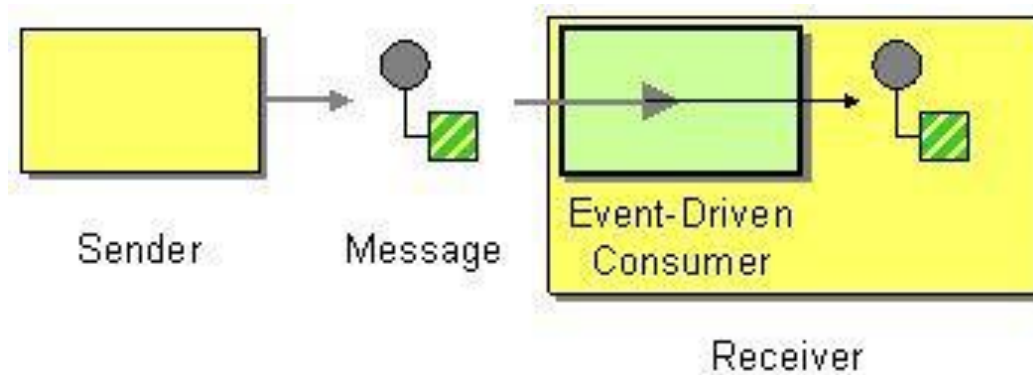
- Synchronous receiver **explicitly** requests messages
- **Blocks** until a message is delivered
- Can **control** how many messages are consumed concurrently (by limiting no. of threads that are polling)
  - Extra messages queue up until receiver can process
- Poll how often?



# Event-Driven Consumer (498)

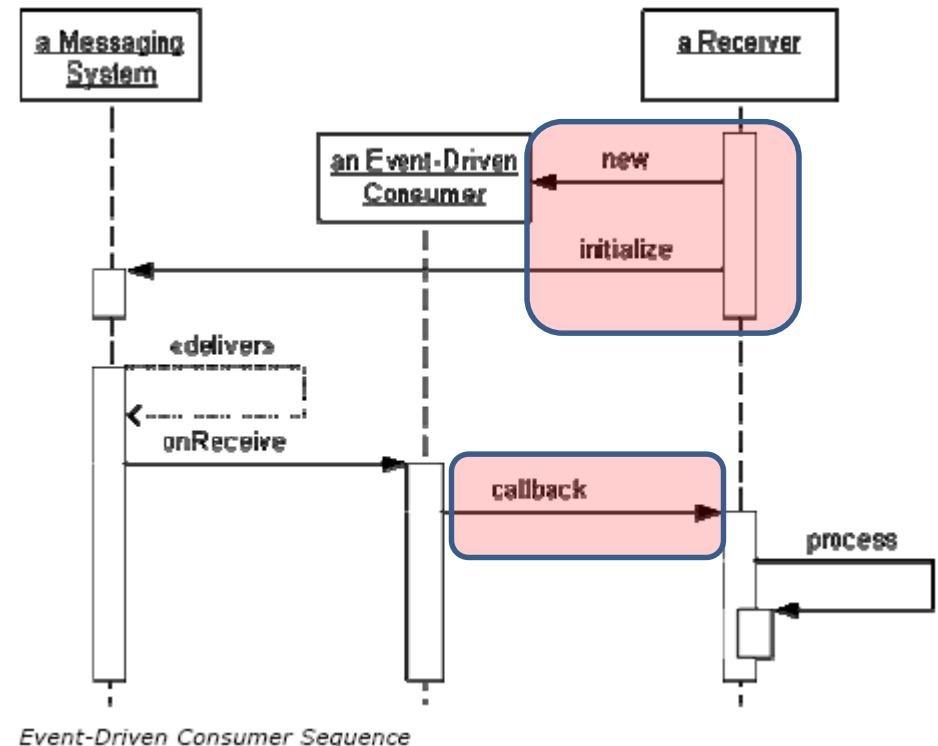


- An Event-Driven Consumer **listens** on a channel and **waits** for a client to send messages to it.
- When a message arrives, the consumer 'wakes up' and takes the message for processing

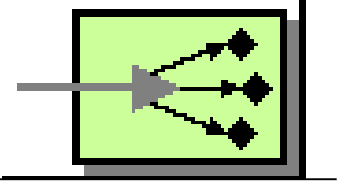


# Event-Driven Consumer Flow - Example

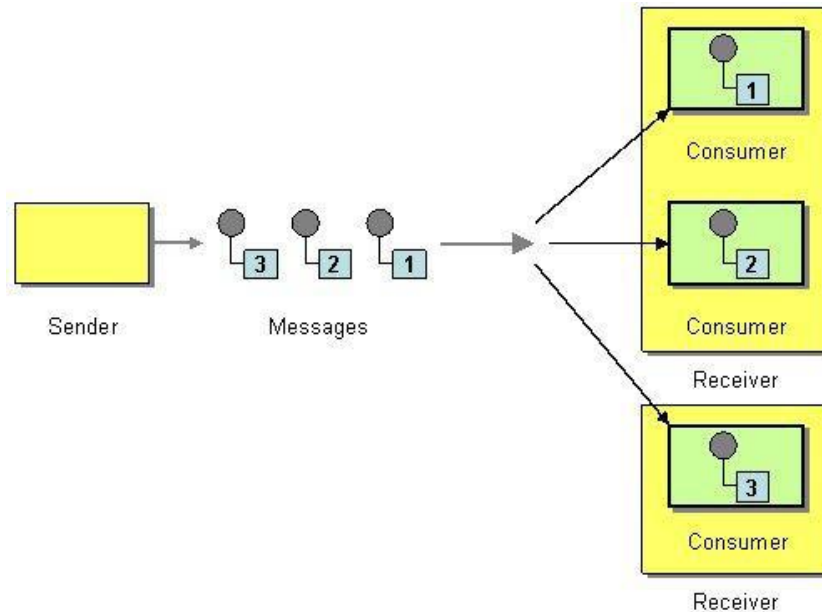
- Asynchronous receiver is invoked by messaging system when a message arrives
- Consumer passes message to application through callback
- Consumer is idle between messages waiting to be invoked



# Competing Consumers (502)

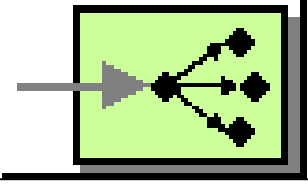


- **Concurrency** is handled by creating multiple Competing Consumers on a single channel
  - consumers process multiple messages concurrently
  - any of the consumers could potentially receive it (they compete)

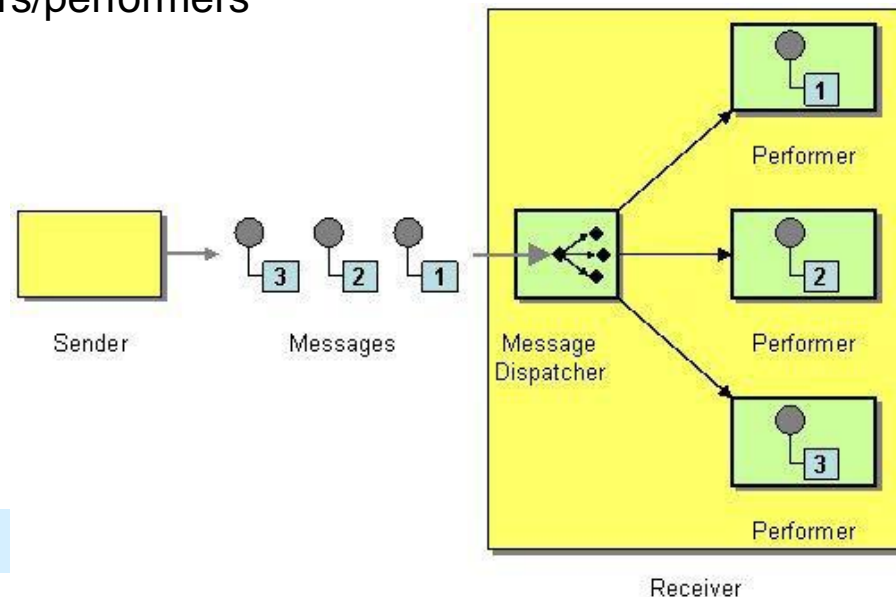


The messaging system's implementation determines which consumer actually receives the message

# Message Dispatcher (508)



- Multiple consumers on a single channel can **coordinate** their own message processing with a message dispatching.
- A Message Dispatcher consumes messages from a channel and distribute them to performers
  - **The client implements the coordination itself**
  - An application can throttle message load by limiting the number of consumers/performers

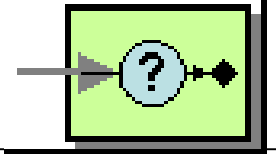


# Message Dispatcher 2

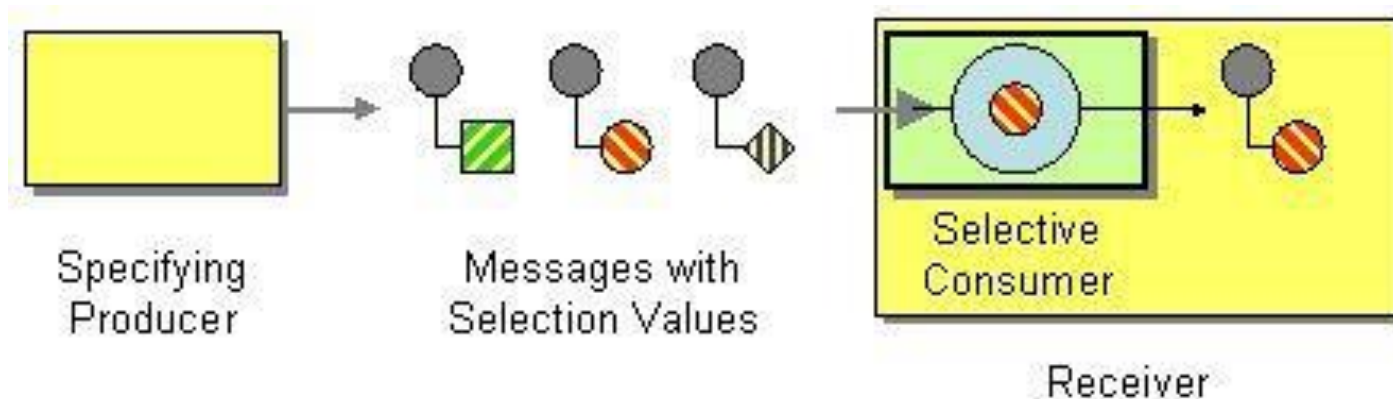
---

- Consists of two parts:
  - **Dispatcher**: consumes messages from a channel and distributes each message to a performer
  - **Performer**: is given the message by the dispatcher and processes it.
- If the **performer** processes in its own **thread**, the dispatcher can receive and delegate other messages, so they can be consumed as fast as the dispatcher can receive and delegate them

# Selective Consumer (515)



- How can a consumer **select** which messages to receive?
- A Selective Consumer **filters** the messages delivered by its channel so that it only receives the ones that match its criteria



# Selective Consumer Filtering Process

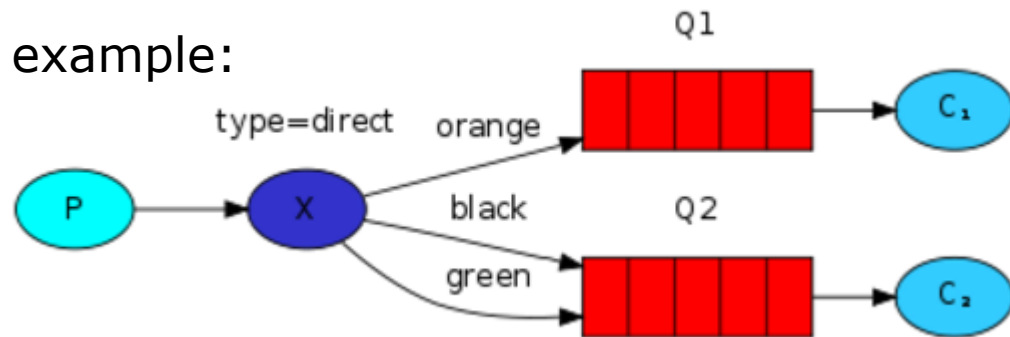
---

- Three parts to the filtering process:
  1. **Producer** specifies the message's selection value before sending
  2. **Selection Value** is one or more values specified in the message
  3. **Selective Consumer** receives messages that meet its selection criteria

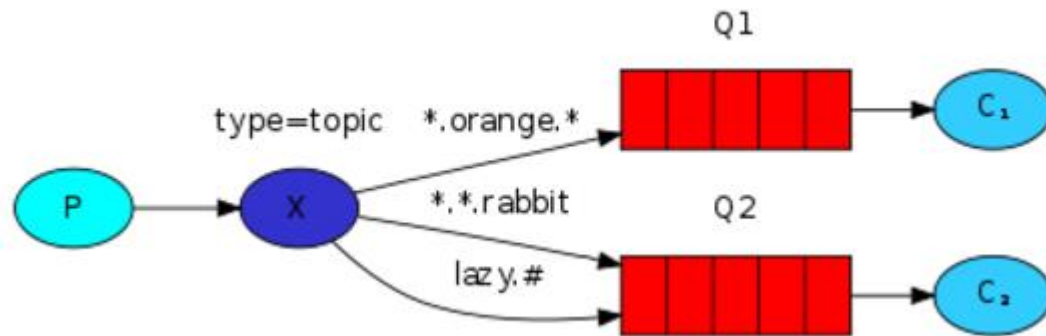


# RabbitMQ Filtering Examples

- **Direct exchange** example:

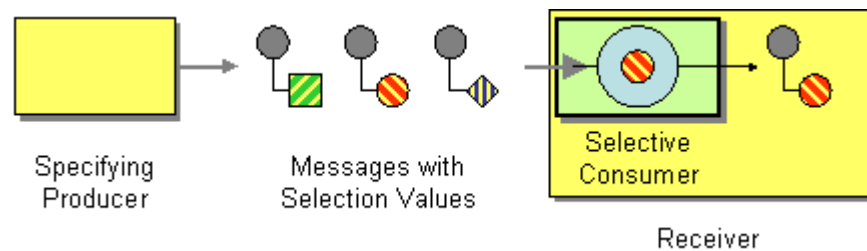


- **Topic exchange** example (more fine-grained routing key)

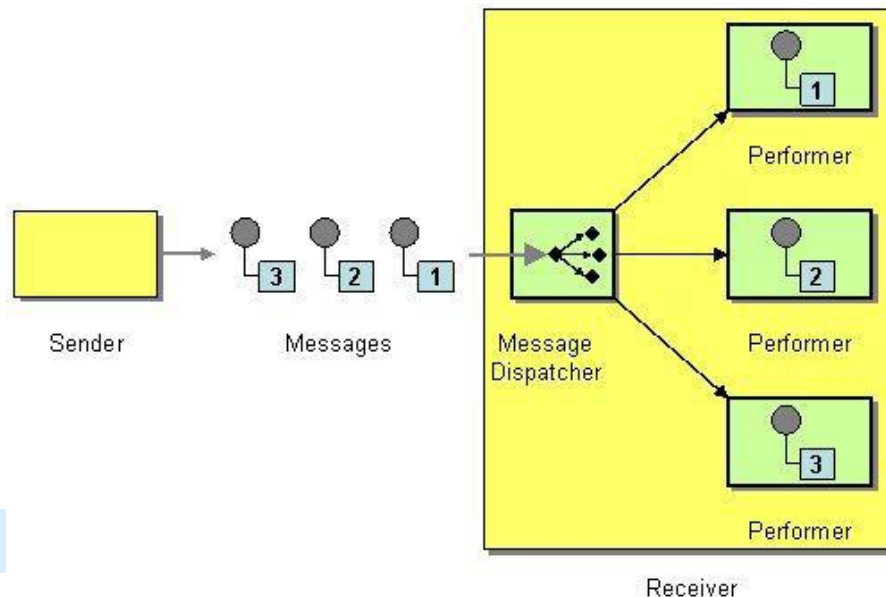


# Selective Consumer vs. Message Dispatcher

You want msg. system to do dispatching (Selective Consumer)



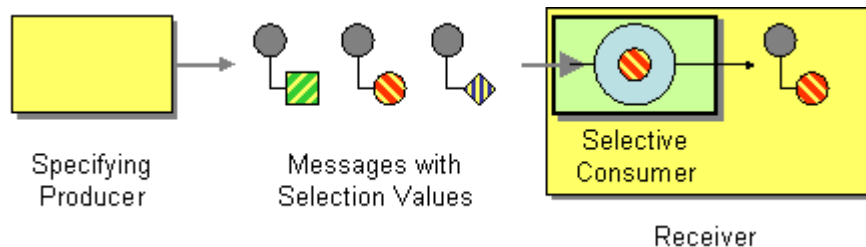
You want the app to do it itself (Message Dispatcher)



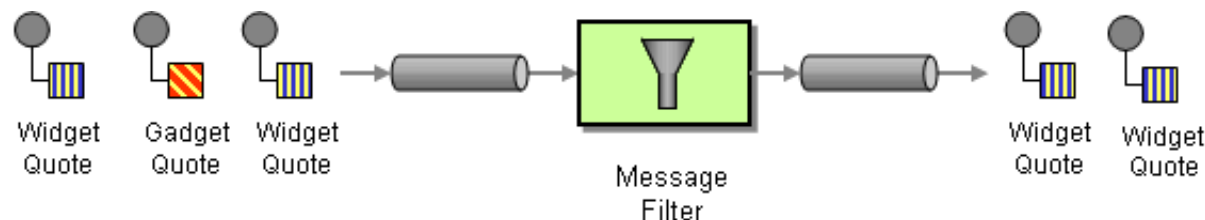
# Selective Consumer vs. Message Filter

Same goal, but different ways

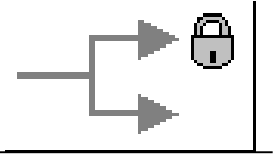
**Selective consumer:** filters the messages delivered by its channel so that it only receives the ones that match its criteria.



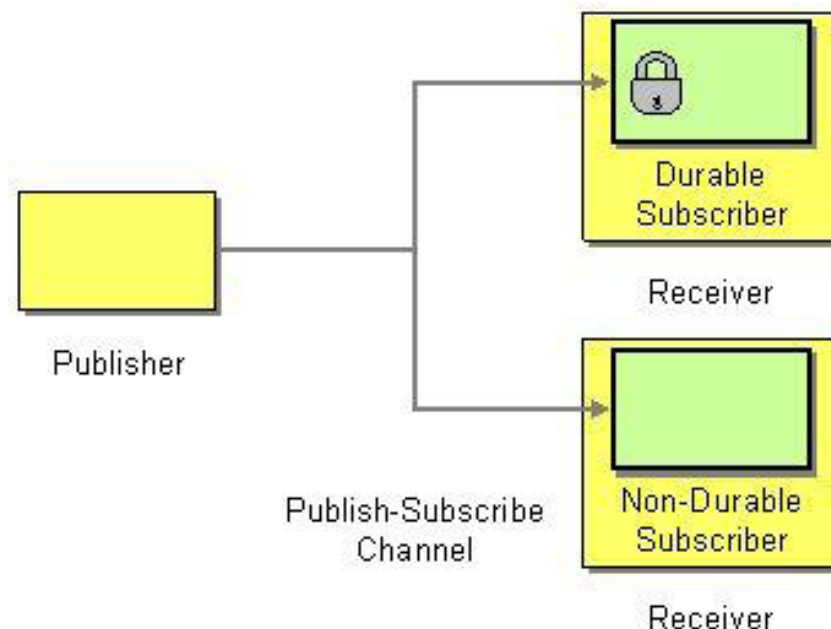
**Message Filter:** eliminates undesired messages from a channel based on a set of criteria.



# Durable Subscriber (522)



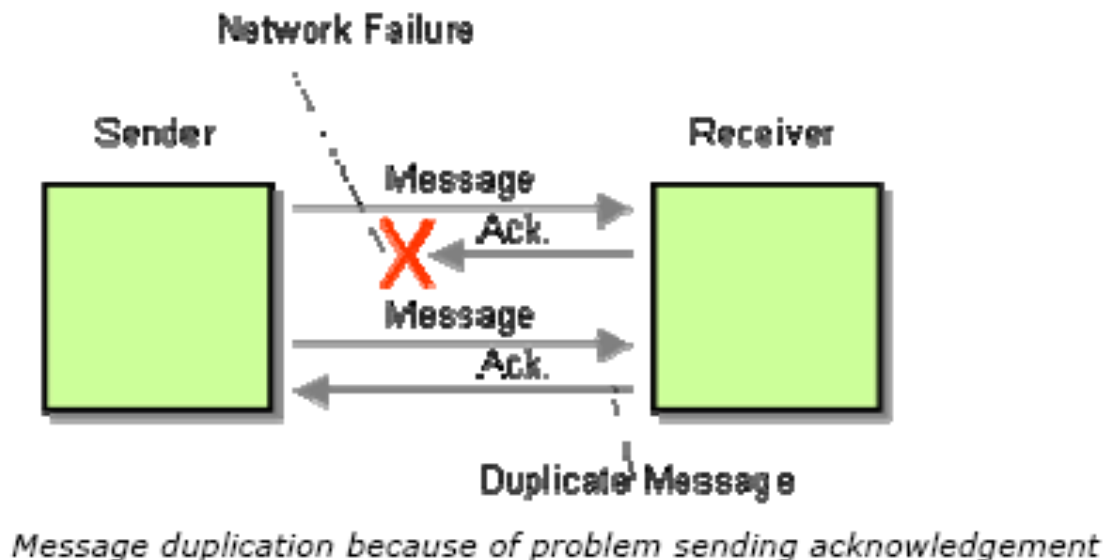
- **How to avoid missing messages while not listening for them?**
- Use a Durable Subscriber to make the messaging system save messages published while the subscriber is disconnected



# Idempotent Receiver (528)

---

- **How to deal with duplicate messages?**
- Design a receiver to be an Idempotent Receiver, one that can safely receive the same message multiple times



# Idempotent Receiver 2

---

Can be achieved through two primary means

- Explicit removal of duplicate messages
  - We must keep track of messages already received
  - How long to keep this history?
  - Must we persist it?
- Define message semantics to support idempotency
  - E.g. send the message so it does not impact the system,
  - Instead of
    - Add \$10 to account 1234
  - Use
    - Set balance of account 1234 to \$110