Messaging Patterns With RabbitMQ

Álvaro Videla - Liip AG

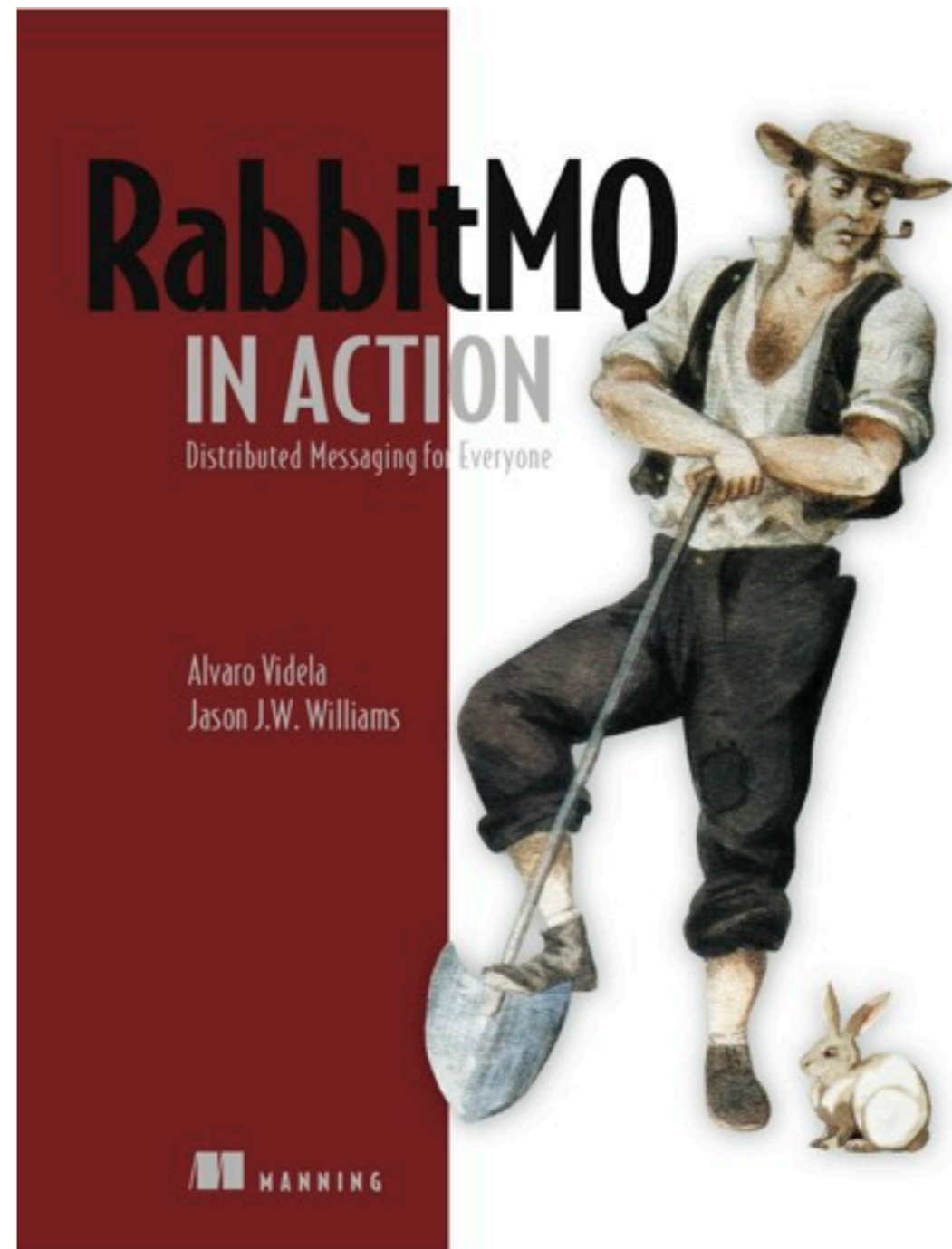# About Me

- Developer at Liip AG

- Blog: http://videlalvaro.github.com/

- Twitter: @old_sound

About Me

Co-authoring

RabbitMQ in Action

http://bit.ly/rabbitmq

Why Do I need Messaging?

An Example

Implement a Photo Gallery

Two Parts:

Upload Picture

Select image from HD [Browse]

[Upload]

Image Gallery
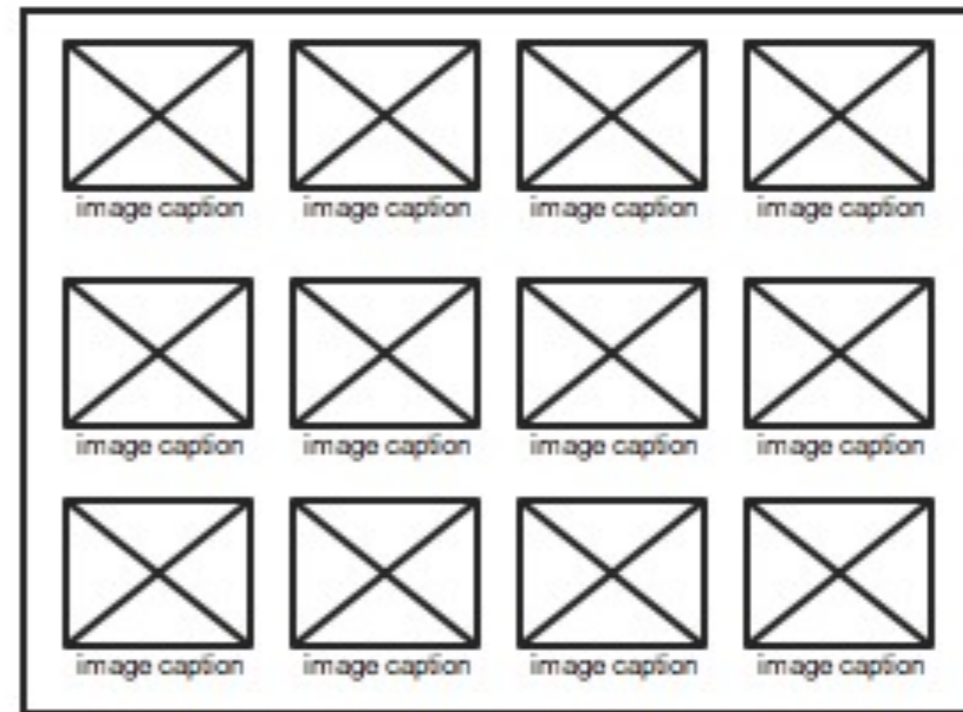
Pretty Simple

'Till new requirements arrive

The Product Owner

Can we also notify the user friends when she uploads a new image?

Can we also notify the user friends when she uploads a new image?

I forgot to mention we need it for tomorrow…

The Social Media Guru

We need to give badges to users for each picture upload

We need to give badges to users for each picture upload

and post uploads to Twitter

The Sysadmin

Dumb! You're delivering full size images! The bandwidth bill has tripled!

Dumb! You're delivering full size images! The bandwidth bill has tripled!

We need this fixed for yesterday!

The Developer in the other team

I need to call your PHP stuff but from Python

I need to call your PHP stuff but from Python

And also Java starting next week

The User

I don't want to wait till your app resizes my image!

You

FML!

Let's see the code evolution

First Implementation:

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
    image_handler:do_upload(ReqData:get_file()),
    ok.
```

Second Implementation:

```erlang
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  resize_image(Image),
  ok.
```

Third Implementation:

```erlang
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  resize_image(Image),
  notify_friends(ReqData:get_user()),
  ok.
```

Fourth Implementation:

```erlang
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  resize_image(Image),
  notify_friends(ReqData:get_user()),
  add_points_to_user(ReqData:get_user()),
  ok.
```

Final Implementation:

```erlang
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  resize_image(Image),
  notify_friends(ReqData:get_user()),
  add_points_to_user(ReqData:get_user()),
  tweet_new_image(User, Image),
  ok.
```

Can our code scale to new requirements?

What if

What if

- We need to speed up image conversion

What if

- We need to speed up image conversion

- User notification has to be sent by email

What if

- We need to speed up image conversion

- User notification has to be sent by email

- Stop tweeting about new images

What if

- We need to speed up image conversion

- User notification has to be sent by email

- Stop tweeting about new images
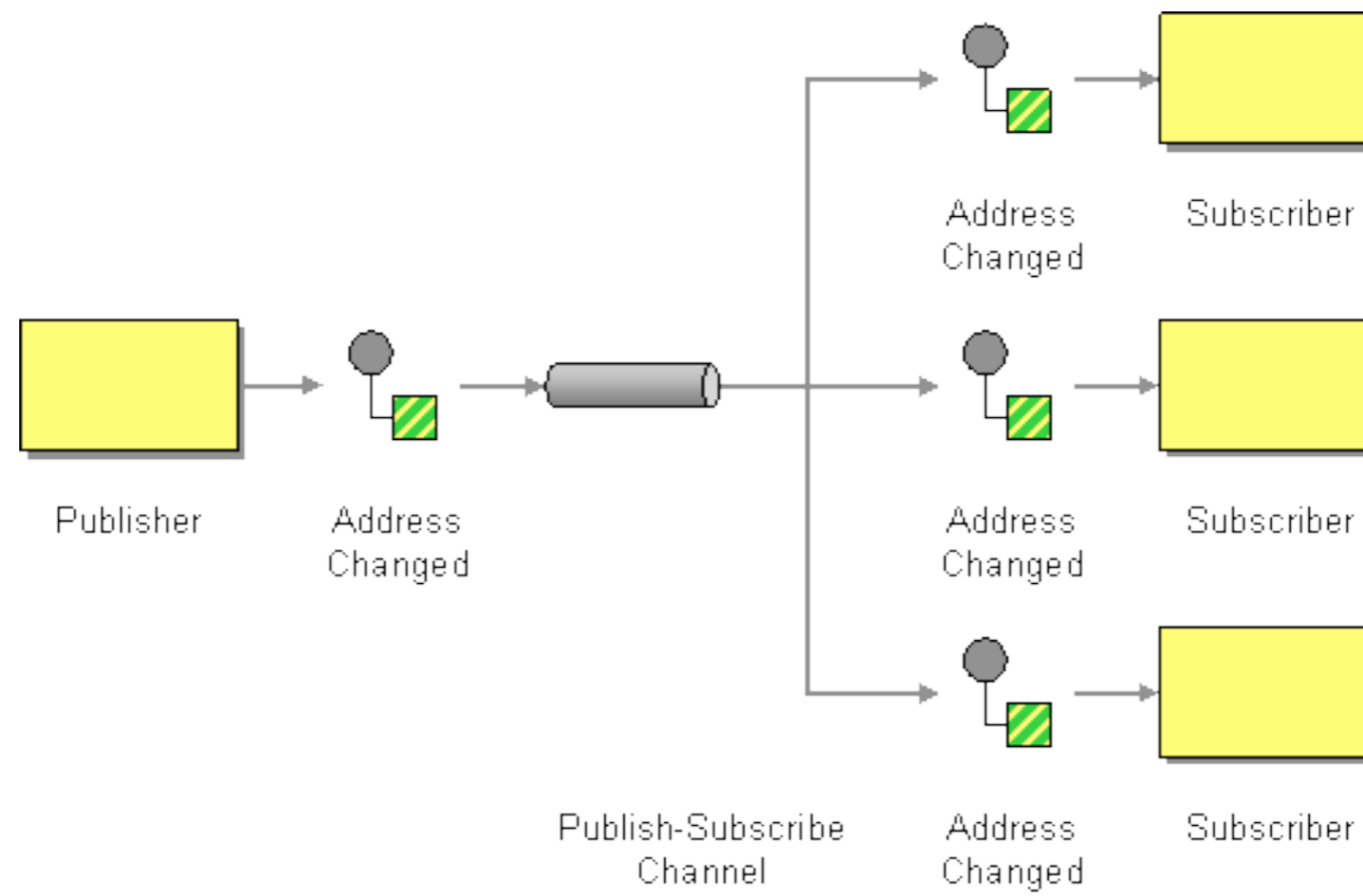
- Resize in different formats

Can we do better?

Sure.
Using messaging

Design

Publish / Subscribe Pattern

First Implementation:

```erlang
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  Msg = #msg{user = ReqData:get_user(), image = Image},
  publish_message('new_image', Msg).
```

First Implementation:

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  Msg = #msg{user = ReqData:get_user(), image = Image},
  publish_message('new_image', Msg).

%% friends notifier
on('new_image', Msg) ->
  notify_friends(Msg.user, Msg.image).
```

First Implementation:

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  Msg = #msg{user = ReqData:get_user(), image = Image},
  publish_message('new_image', Msg).

%% friends notifier
on('new_image', Msg) ->
  notify_friends(Msg.user, Msg.image).

%% points manager
on('new_image', Msg) ->
  add_points(Msg.user, 'new_image').
```

First Implementation:

```
%% image_controller
handle('PUT', "/user/image", ReqData) ->
  {ok, Image} = image_handler:do_upload(ReqData:get_file()),
  Msg = #msg{user = ReqData:get_user(), image = Image},
  publish_message('new_image', Msg).

%% friends notifier
on('new_image', Msg) ->
  notify_friends(Msg.user, Msg.image).

%% points manager
on('new_image', Msg) ->
  add_points(Msg.user, 'new_image').

%% resizer
on('new_image', Msg) ->
  resize_image(Msg.image).
```

Second Implementation:

Second Implementation:

`%% there's none.`

Messaging

Messaging

- Share data across processes

Messaging

- Share data across processes

- Processes can be part of different apps

Messaging

- Share data across processes

- Processes can be part of different apps

- Apps can live in different machines

Messaging

- Share data across processes

- Processes can be part of different apps

- Apps can live in different machines

- Communication is Asynchronous

Main Concepts

Main Concepts

- Messages are sent by **Producers**

Main Concepts

- Messages are sent by **Producers**

- Messages are delivered to **Consumers**

Main Concepts

- Messages are sent by **Producers**

- Messages are delivered to **Consumers**

- Messages goes through a **Channel**

Messaging and RabbitMQ

What is RabbitMQ?

RabbitMQ

- Enterprise Messaging System

- Open Source MPL

- Written in Erlang/OTP

- Commercial Support

- Messaging via AMQP

Features

- Reliable and High Scalable

- Easy To install

- Easy To Cluster

- Runs on: Windows, Solaris, Linux, OSX

- AMQP 0.8 - 0.9.1

Client Libraries

- Java

- .NET/C#

- Erlang

- Ruby, Python, PHP, Perl, AS3, Lisp, Scala, Clojure, Haskell

AMQP

- Advanced Message Queuing Protocol

- Suits Interoperability

- Completely Open Protocol

- Binary Protocol

Message Flow

AMQP Model

- Exchanges
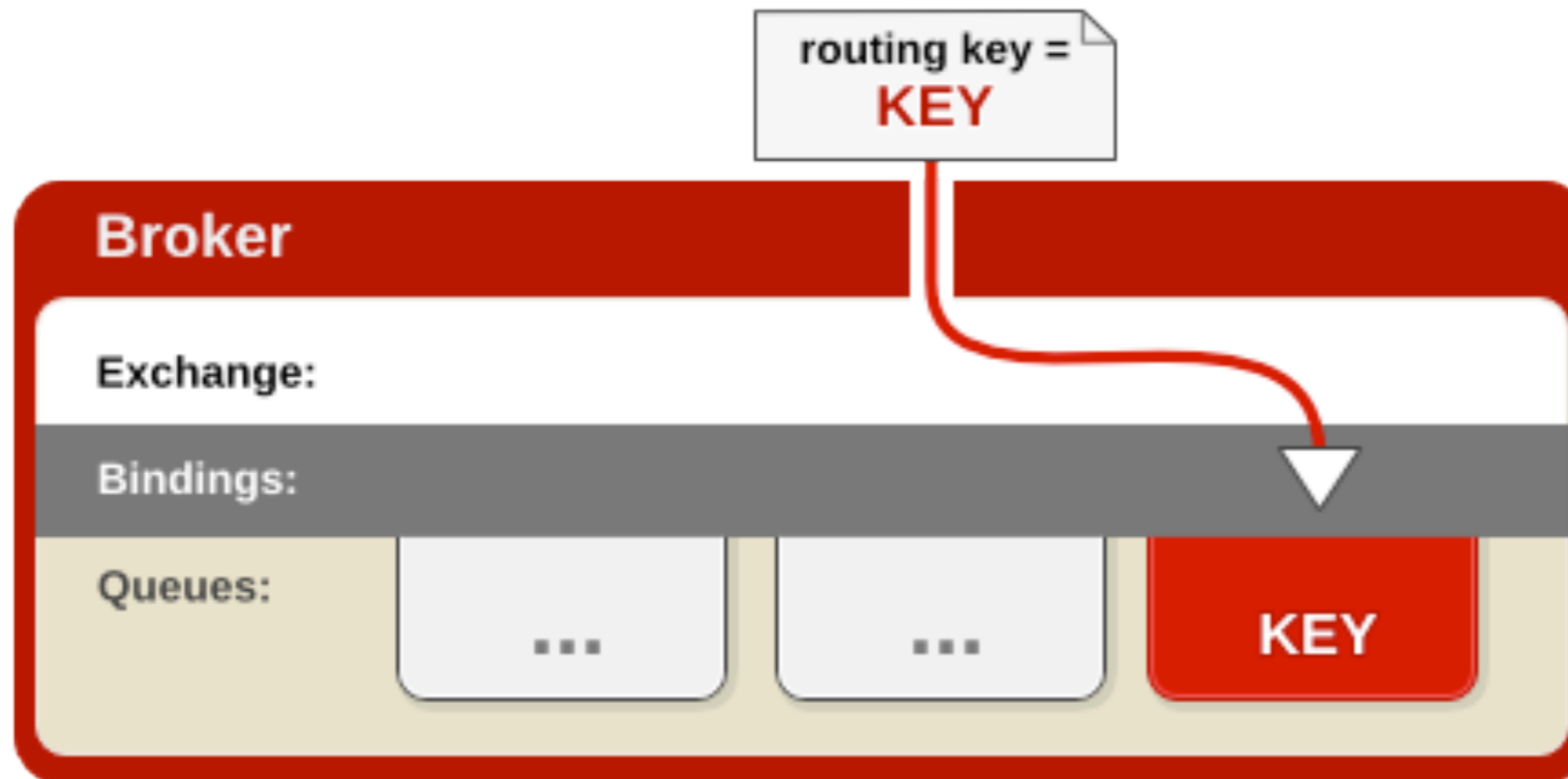
- Message Queues

- Bindings

- Rules for binding them

Exchange Types
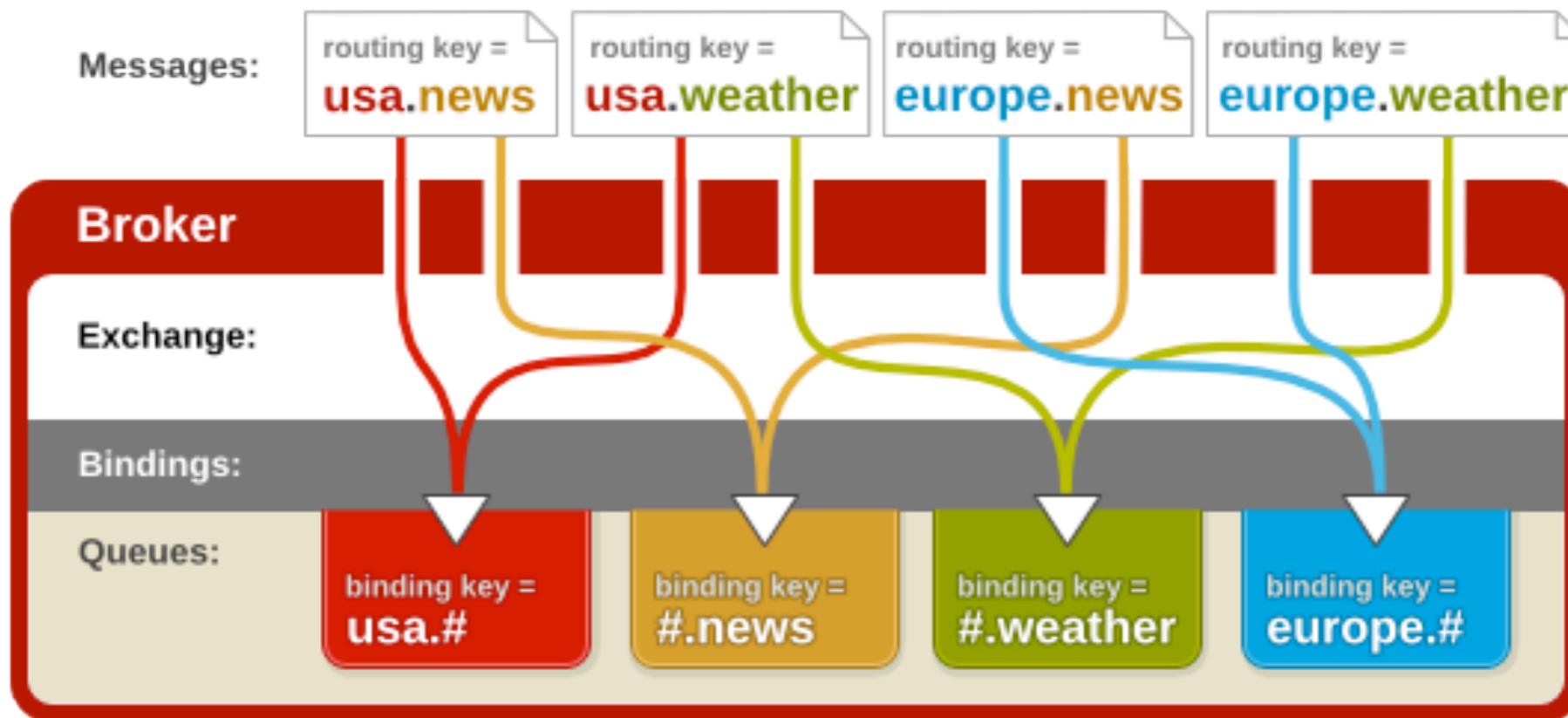
- Fanout
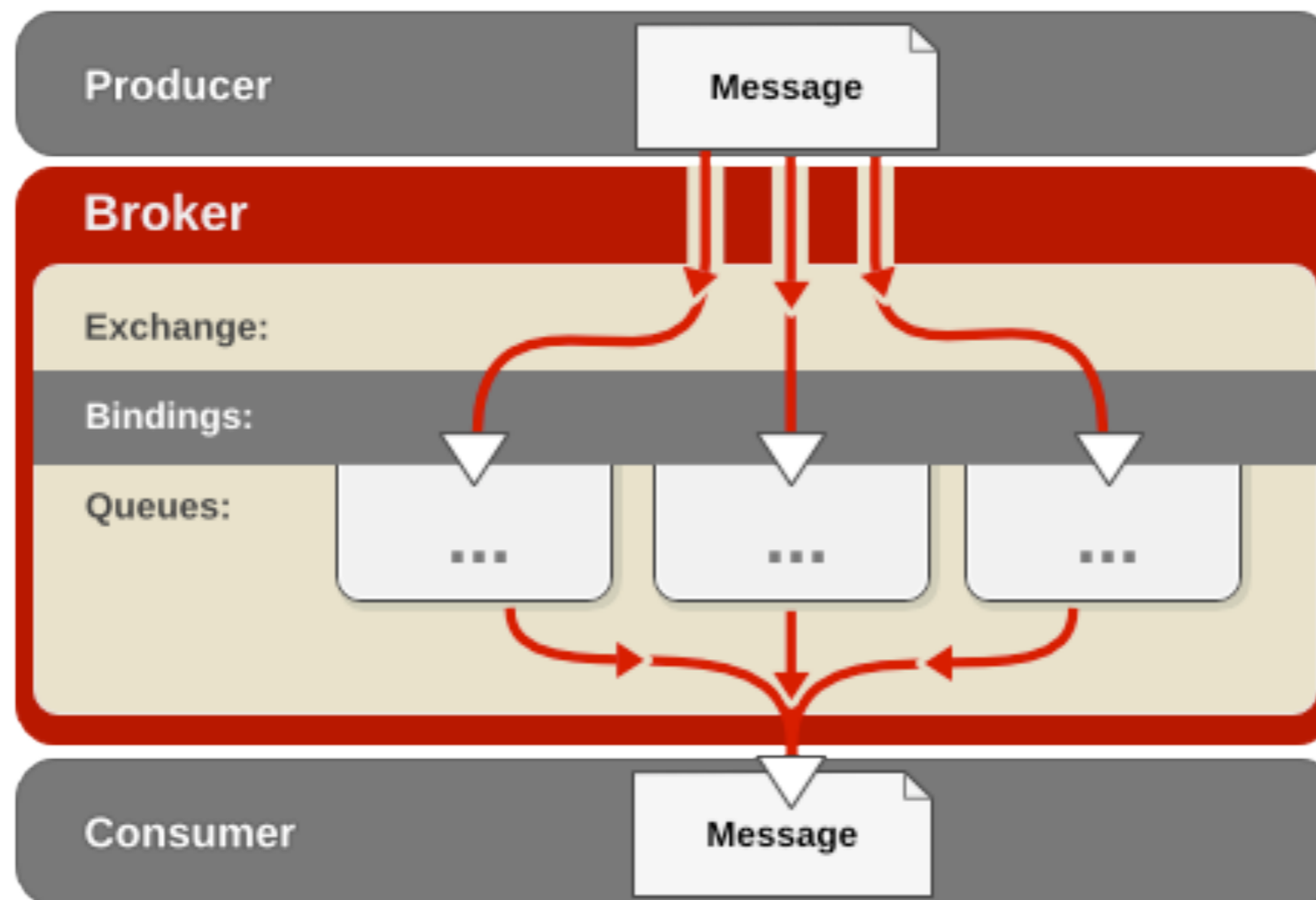
- Direct

- Topic
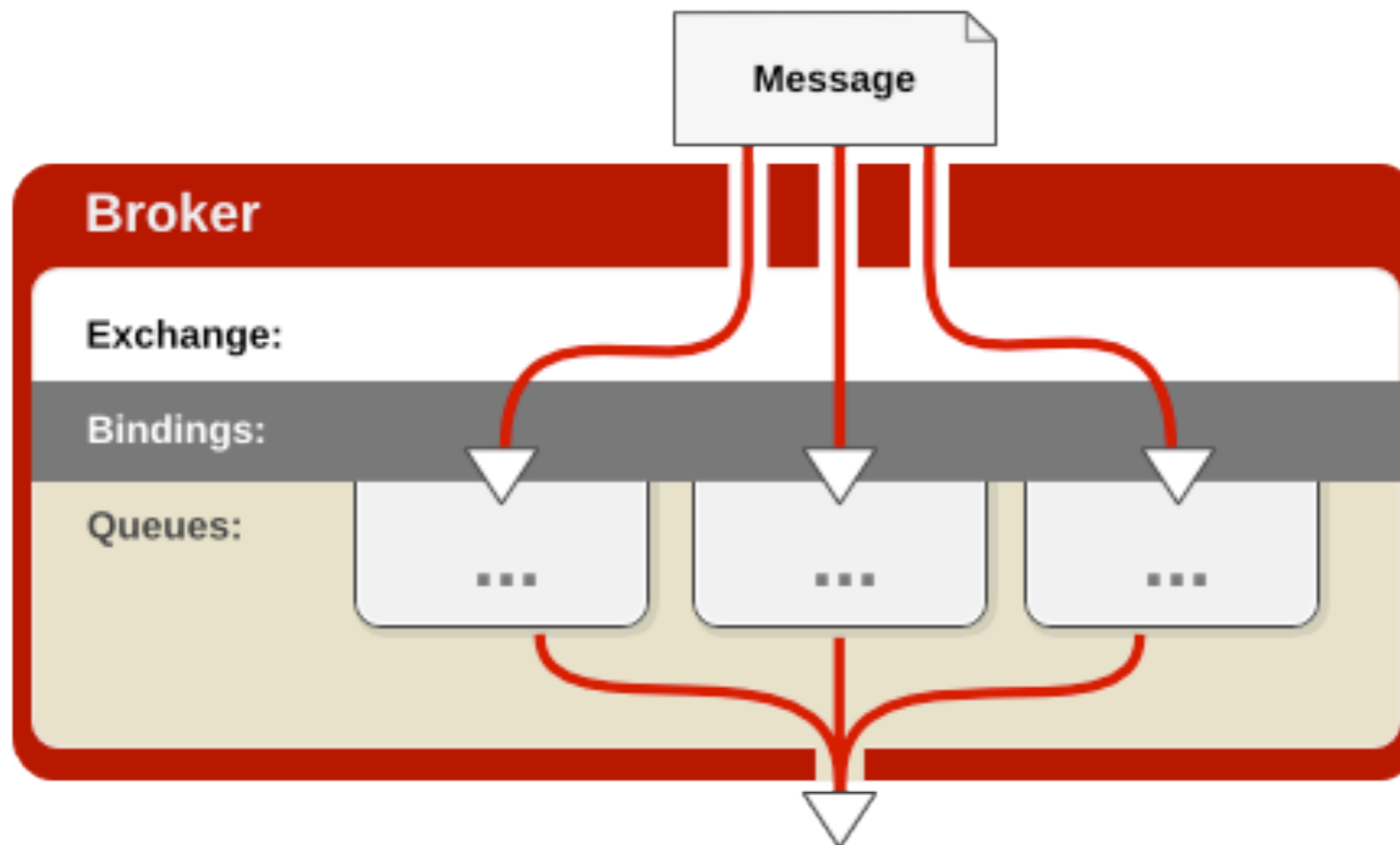
http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_MRG/1.0/html/Messaging_Tutorial/sect-Messaging_Tutorial-Initial_Concepts-Fanout_Exchange.html
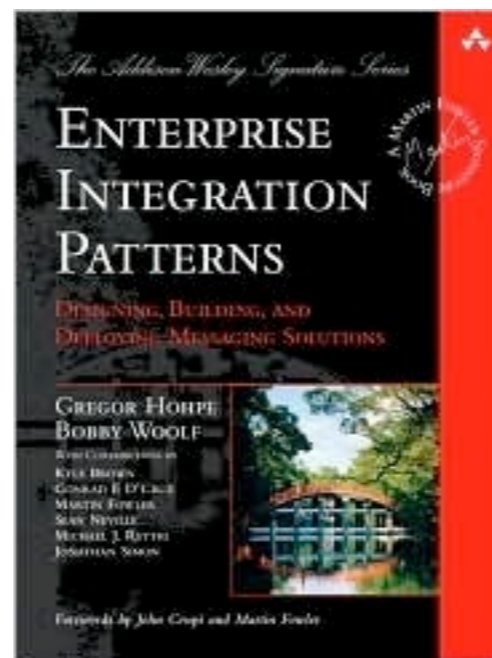
Direct Exchange

Topic Exchange

Messaging Patterns

There are many messaging patterns

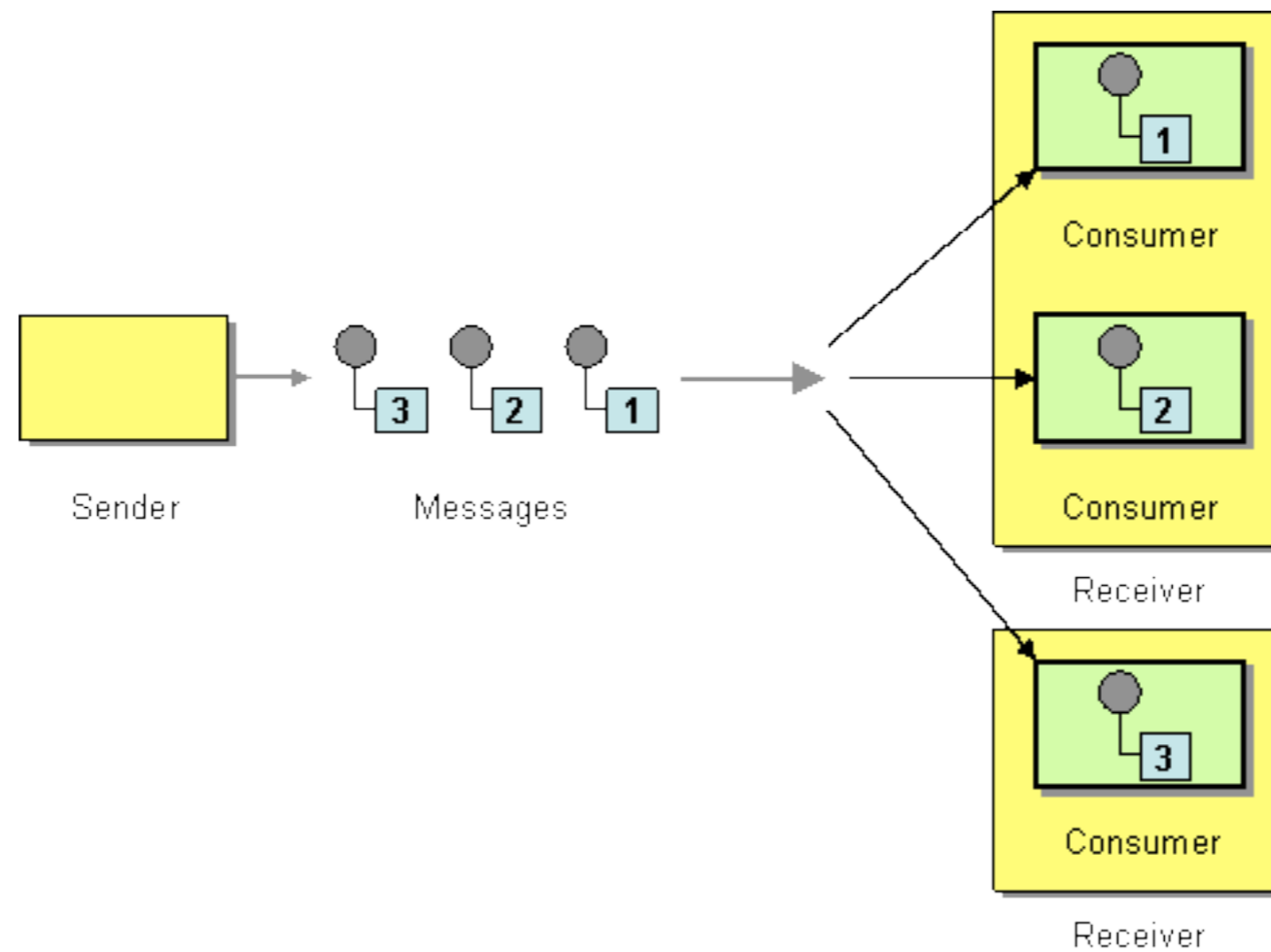http://www.eaipatterns.com/

Basic Patterns

Competing Consumers

How can a messaging client process multiple messages concurrently?

Competing Consumers

Create multiple *Competing Consumers* on a single channel so that the consumers can process multiple messages concurrently.

Competing Consumers

Publisher Code

```erlang
init(Exchange, Queue) ->
    #'exchange.declare'{exchange = Exchange,
                        type = <<"direct">>,
                        durable = true},
    #'queue.declare'{queue = Queue, durable = false},
    #'queue.bind'{queue = Queue, exchange = Exchange}.


publish_msg(Exchange, Payload) ->
    Props = #'P_basic'{content_type = <<"application/json">>,
                       delivery_mode = 2}, %% persistent
    publish(Exchange, #amqp_msg{props = Props, payload = Payload}).
```

Consumer Code

```erlang
init_consumer(Exchange, Queue) ->
    init(Exchange, Queue),
    #'basic.consume'{queue = Queue}.


on(#'basic.deliver'{delivery_tag = DeliveryTag},
   #amqp_msg{} = Msg) ->
       do_something_with_msg(Msg),
       #'basic.ack'{delivery_tag = DeliveryTag}.
```
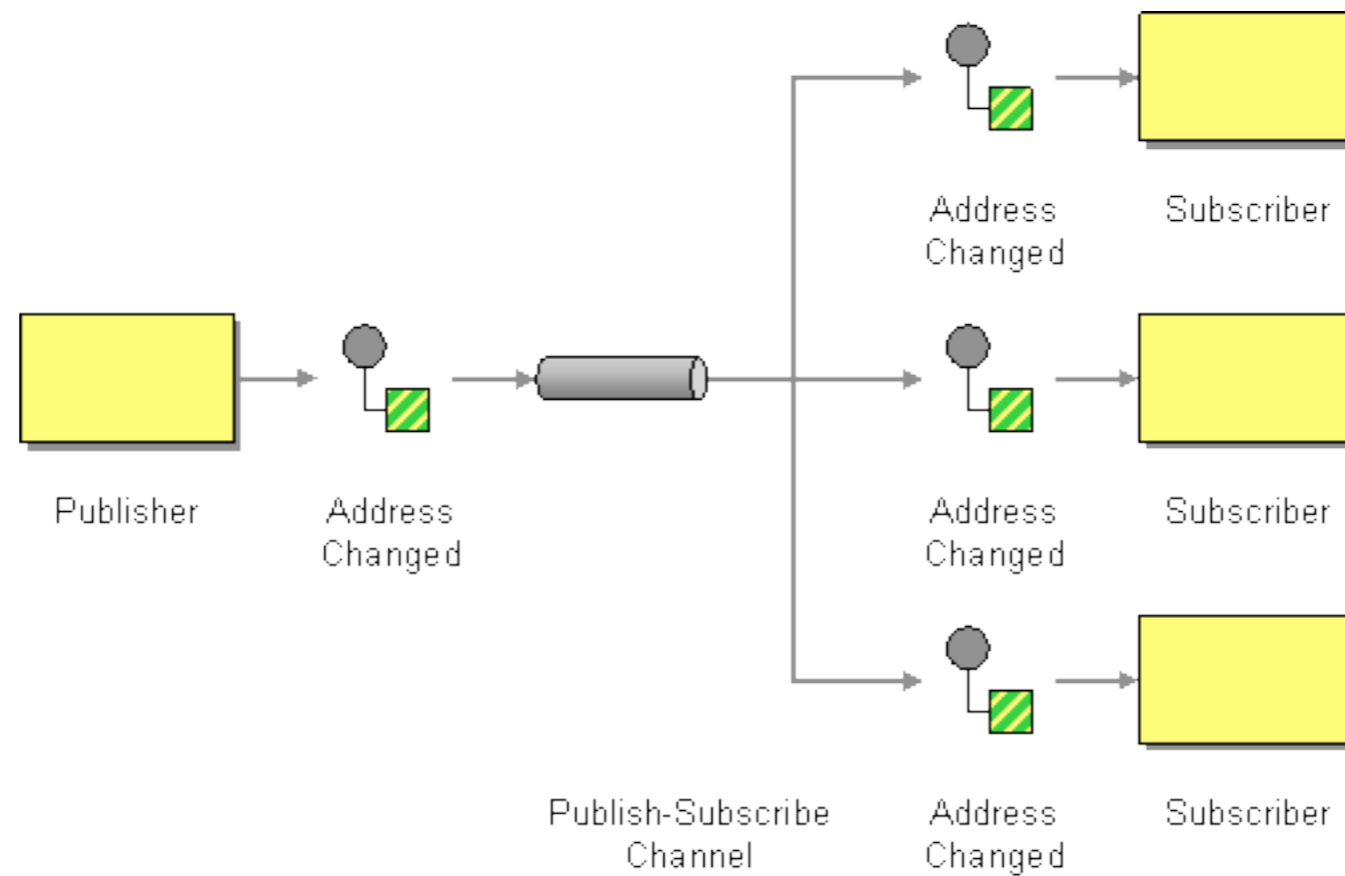
Publish/Subscribe

How can the sender broadcast an event to all interested receivers?

Publish/Subscribe

Send the event on a *Publish-Subscribe Channel*, which delivers a copy of a particular event to each receiver.

Publish/Subscribe

Publisher Code

```erlang
init(Exchange, Queue) ->
    #'exchange.declare'{exchange = Exchange,
                        type = <<"fanout">>, %% different type
                        durable = true}
    %% same as before ...


publish_msg(Exchange, Payload) ->
    Props = #'P_basic'{content_type = <<"application/json">>,
                       delivery_mode = 2}, %% persistent
    publish(Exchange, #amqp_msg{props = Props, payload = Payload}).
```

Consumer Code A

```
init_consumer(Exchange, ResizeImageQueue) ->
    init(Exchange, ResizeImageQueue),
    #'basic.consume'{queue = ResizeImageQueue}.


on(#'basic.deliver'{delivery_tag = DeliveryTag},
    #amqp_msg{} = Msg) ->
        resize_message(Msg),
        #'basic.ack'{delivery_tag = DeliveryTag}.
```

Consumer Code B

```
init_consumer(Exchange, NotifyFriendsQueue) ->
    init(Exchange, NotifyFriendsQueue),
    #'basic.consume'{queue = NotifyFriendsQueue}.


on(#'basic.deliver'{delivery_tag = DeliveryTag},
    #amqp_msg{} = Msg) ->
        notify_friends(Msg),
        #'basic.ack'{delivery_tag = DeliveryTag}.
```

Consumer Code C

```
init_consumer(Exchange, LogImageUpload) ->
    init(Exchange, LogImageUpload),
    #'basic.consume'{queue = LogImageUpload}.


on(#'basic.deliver'{delivery_tag = DeliveryTag},
    #amqp_msg{} = Msg) ->
        log_image_upload(Msg),
        #'basic.ack'{delivery_tag = DeliveryTag}.
```
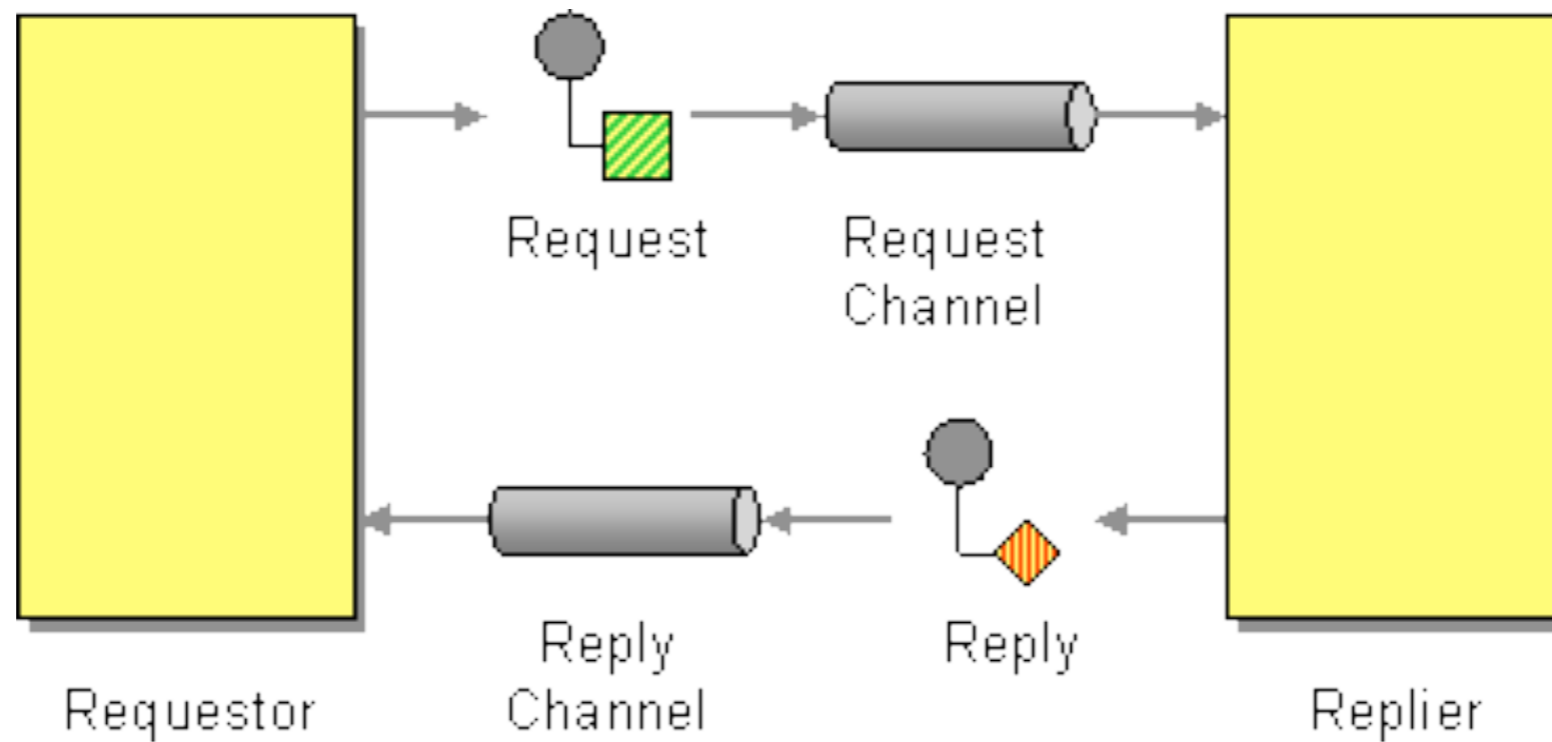
Request/Reply

When an application sends a message, how can it get a response from the receiver?

Request/Reply

Send a pair of *Request-Reply* messages, each on its own channel.
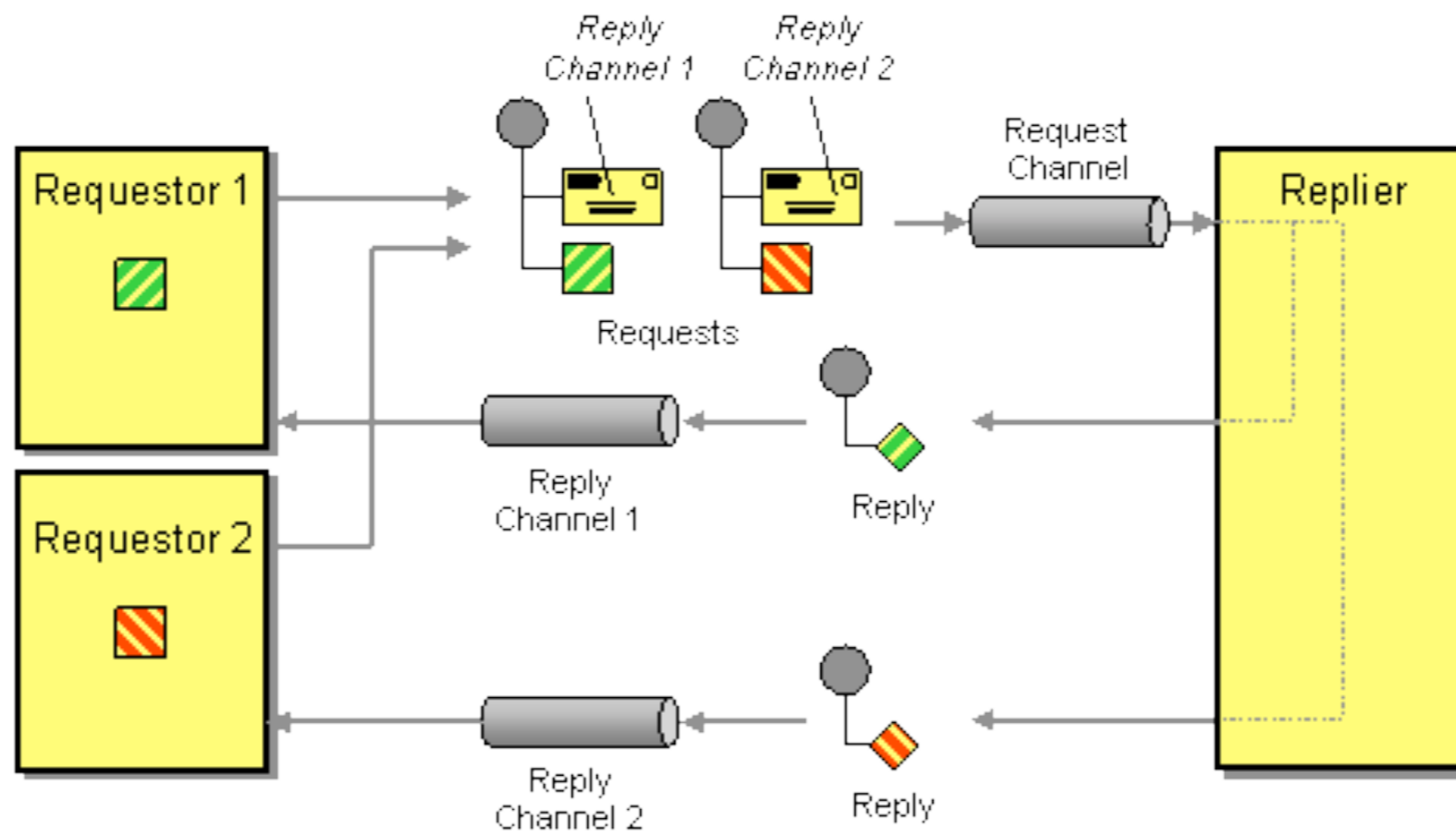
Request/Reply

Return Address

How does a replier know where to send the reply?

Return Address

The request message should contain a *Return Address* that indicates where to send the reply message.
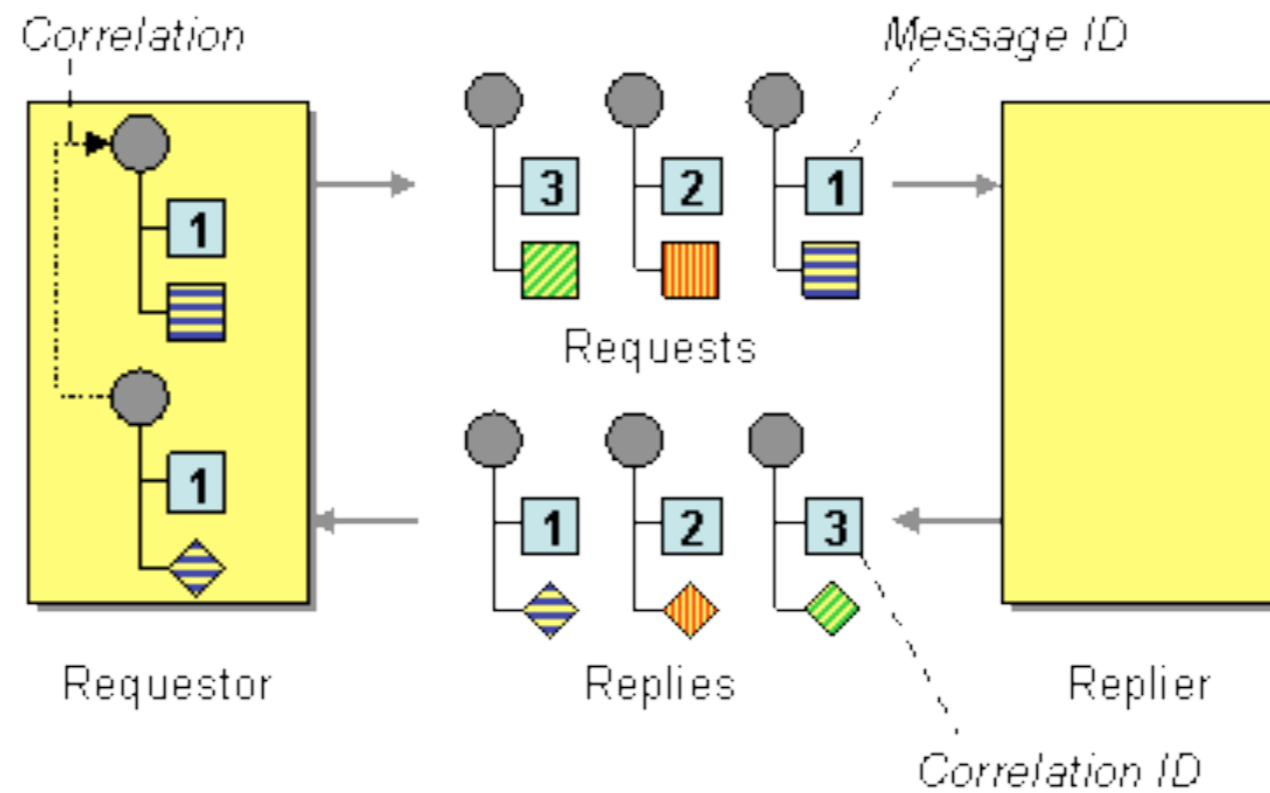
Return Address

Correlation Identifier

How does a requestor that has received a reply know which request this is the reply for?

Correlation Identifier

Each reply message should contain a *Correlation Identifier*, a unique identifier that indicates which request message this reply is for.

Correlation Identifier

Putting it all together

RPC Client

```
init() ->
    #'queue.declare_ok'{queue = SelfQueue} =
        #'queue.declare'{exclusive = true, auto_delete = true},
    #'basic.consume'{queue = SelfQueue, no_ack = true},
    SelfQueue.
```

RPC Client

```
init() ->
    #'queue.declare_ok'{queue = SelfQueue} =
        #'queue.declare'{exclusive = true, auto_delete = true},
    #'basic.consume'{queue = SelfQueue, no_ack = true},
    SelfQueue.


request(Payload, RequestId) ->
    Props = #'P_basic'{correlation_id = RequestId,
                       reply_to = SelfQueue},
    publish(ServerExchange, #amqp_msg{props = Props,
                                      payload = Payload}).
```

RPC Client

```
init() ->
    #'queue.declare_ok'{queue = SelfQueue} =
        #'queue.declare'{exclusive = true, auto_delete = true},
    #'basic.consume'{queue = SelfQueue, no_ack = true},
    SelfQueue.


request(Payload, RequestId) ->
    Props = #'P_basic'{correlation_id = RequestId,
                       reply_to = SelfQueue},
    publish(ServerExchange, #amqp_msg{props = Props,
                                      payload = Payload}).


on(#'basic.deliver'{},
   #amqp_msg{props = Props, payload = Payload}) ->
    CorrelationId = Props.correlation_id,
    do_something_with_reply(Payload).
```

RPC Server

```erlang
on(#'basic.deliver'{},
    #amqp_msg{props = Props, payload = Payload}) ->

    CorrelationId = Props.correlation_id,

    ReplyTo = Props.reply_to,

    Reply = process_request(Payload),

    NewProps = #'P_basic'{correlation_id = CorrelationId},

    publish("", %% anonymous exchange
            #amqp_msg{props = NewProps,
                      payload = Reply},
            ReplyTo). %% routing key
```

Advanced Patterns

Control Bus

How can we effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area?
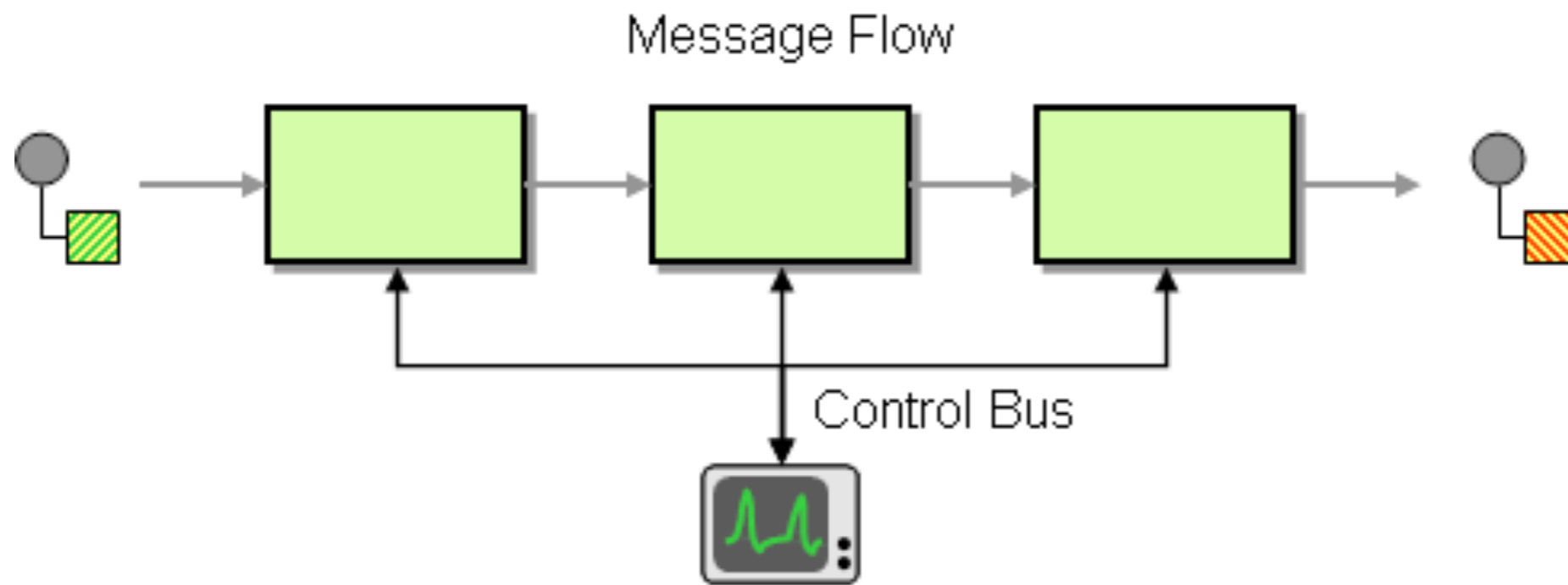
Control Bus

Use a *Control Bus* to manage an enterprise integration system.

Control Bus

- Send Configuration Messages

- Start/Stop Services

- Inject Test Messages

- Collect Statistics

Control Bus

Control Bus

Make Services
## "Control Bus" Enabled

Detour

How can you route a message through intermediate steps to perform validation, testing or debugging functions?

Detour

Construct a *Detour* with a context-based router controlled via the *Control Bus*.

In one state the router routes incoming messages through additional steps while in the other it routes messages directly to the destination channel.

Detour
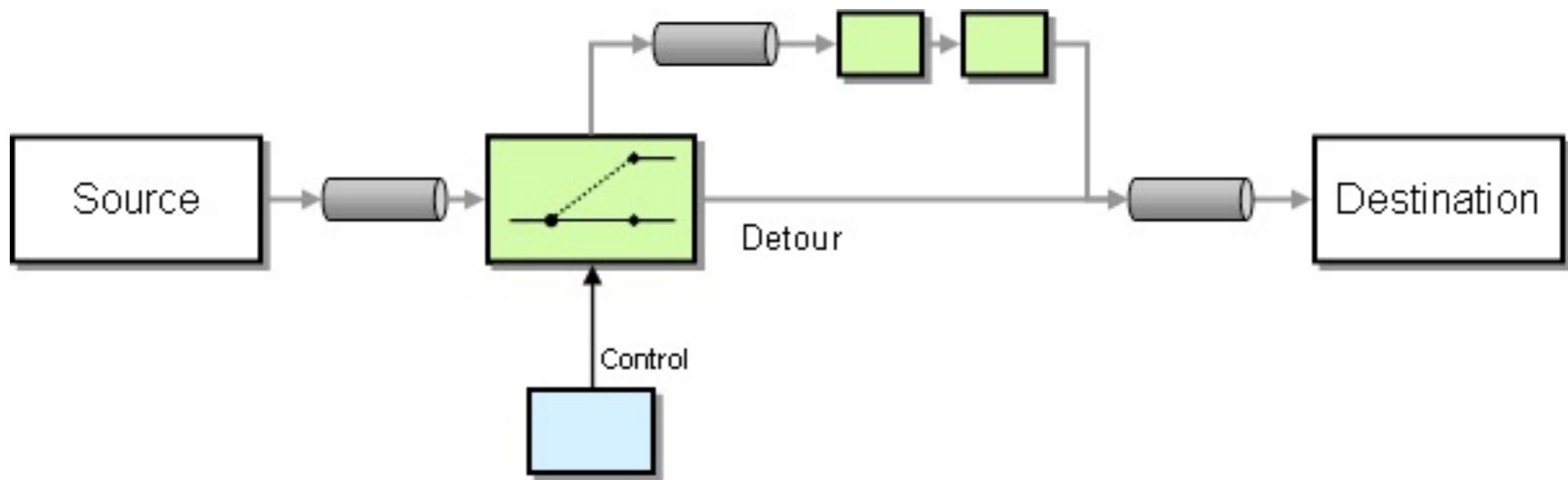
Wire Tap

How do you inspect messages that travel on a point-to-point channel?

Wire Tap

Insert a simple Recipient List into the channel that publishes each incoming message to the main channel and a secondary channel.

Wire Tap

How do you inspect messages that travel on a point-to-point channel?

Wire Tap

Insert a simple Recipient List into the channel that publishes each incoming message to the main channel and a secondary channel.
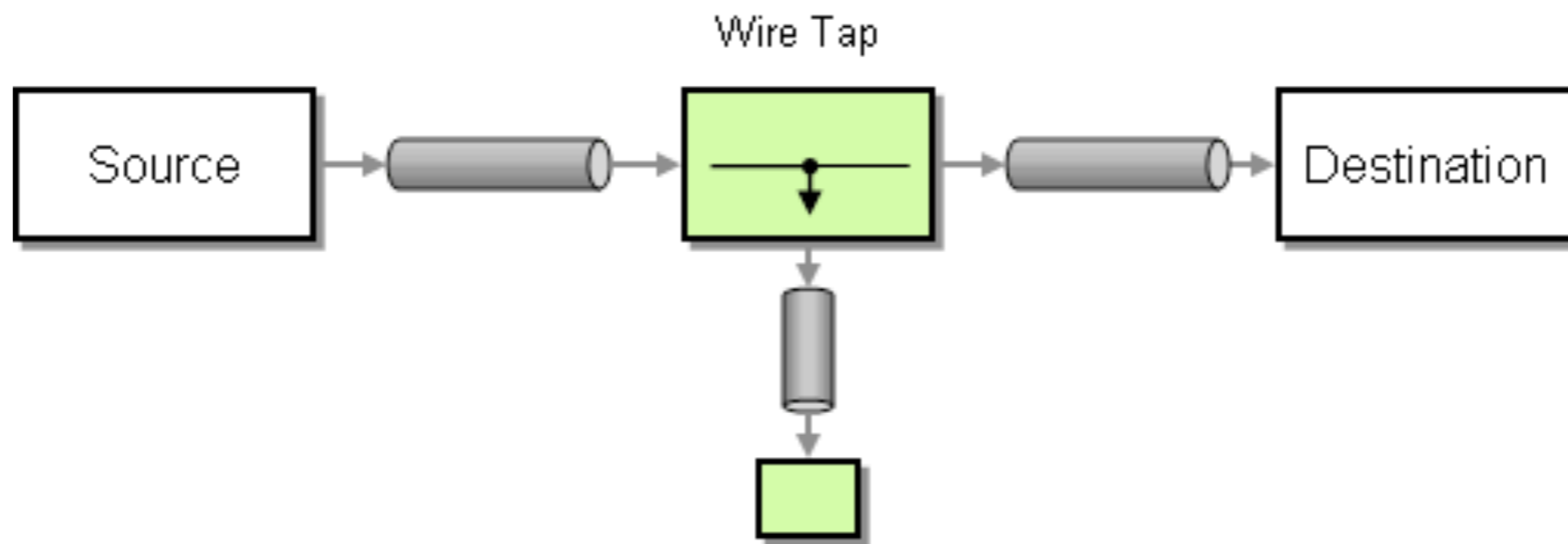
Wire Tap

Wire Tap

Source → Wire Tap → Destination

Smart Proxy

How can you track messages on a service that publishes reply messages to the Return Address specified by the requestor?

Smart Proxy

Use a *Smart Proxy* to store the Return Address supplied by the original requestor and replace it with the address of the *Smart Proxy*.

When the service sends the reply message route it to the original Return Address.

Smart Proxy
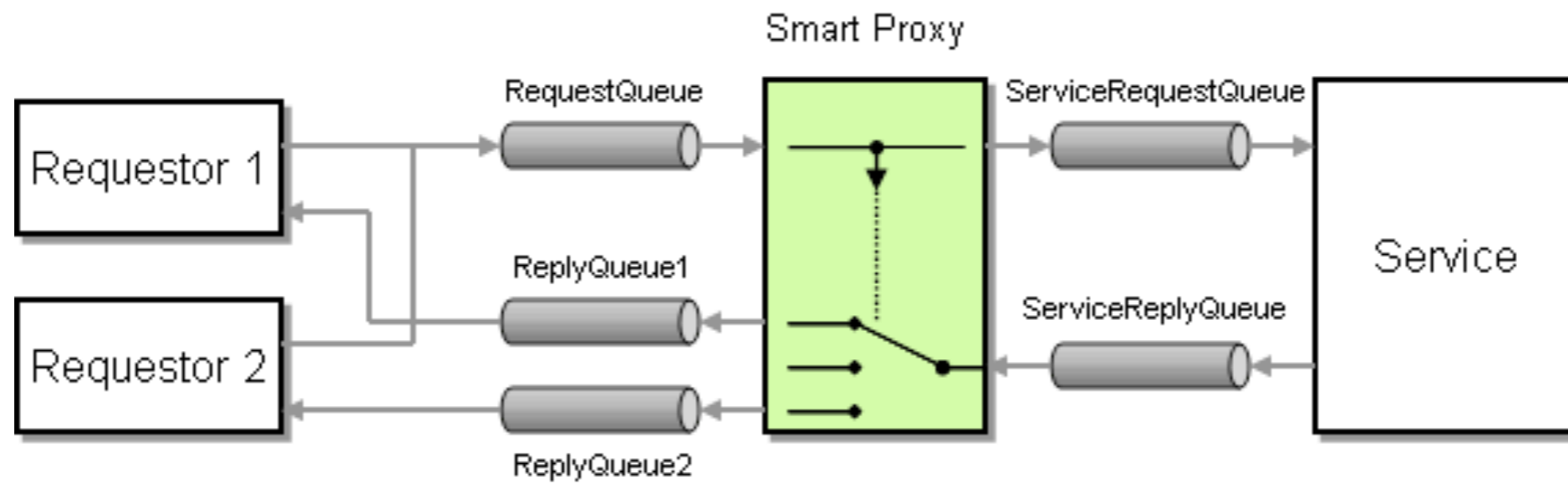
Credits

Pattern graphics and description taken from:
http://www.eaipatterns.com/

Thanks!

@old_sound
http://vimeo.com/user1169087
http://www.slideshare.net/old_sound