# Practical Messaging

A 101 guide to messaging

Ian Cooper

Twitter: ICooper

# Who are you?

- Software Developer for  more than 25 years
  - Stuff I care about: Messaging, EDA, Microservices, TDD, XP, OO, RDD & DDD, Code that Fits in My Head, C#
  - Places I have worked: DTI, Reuters, Sungard, Beazley, Huddle, Just Eat Takeaway
- No smart folks
  - Just the folks in this room

BRIGHTER

HOME INTRO TUTORIALS CONCEPTS DOCS

# Welcome to Brighter

This project is a Command Processor & Dispatcher implementation with support for task queues that can be used as a lightweight library.

It can be used for implementing Ports and Adapters and CQRS (PDF) architectural styles in .NET.

It can also be used in microservices architectures for decoupled communication between the services

GET STARTED

# Day One Agenda

- Distribution
- Integration Styles
- Request Driven Architectures
- Event Driven Architectures
- Messaging Patterns

# When do you get to write code?

When we get to messaging patterns, hopefully by the afternoon of Day One and then into the morning of Day Two

# Prerequisites

We will use Rabbit MQ for examples. Either you need to have RMQ installed on your machine, or you should have Docker installed on your machine, as exercises provide a Docker Compose file to spin up RMQ.

You will need to be able to author C#, Python, or JavaScript with an editor/IDE of your choice.

You will need Python 3, .NET Core, or ES6

# Exercise Code

https://github.com/iancooper/Practical-Messaging-Sharp
https://github.com/iancooper/Practical-Messaging-Python
https://github.com/iancooper/Practical-Messaging-JavaScript

# Day One

What is driving messaging

# 1.DISTRIBUTED SYSTEMS

# Why Distribute?

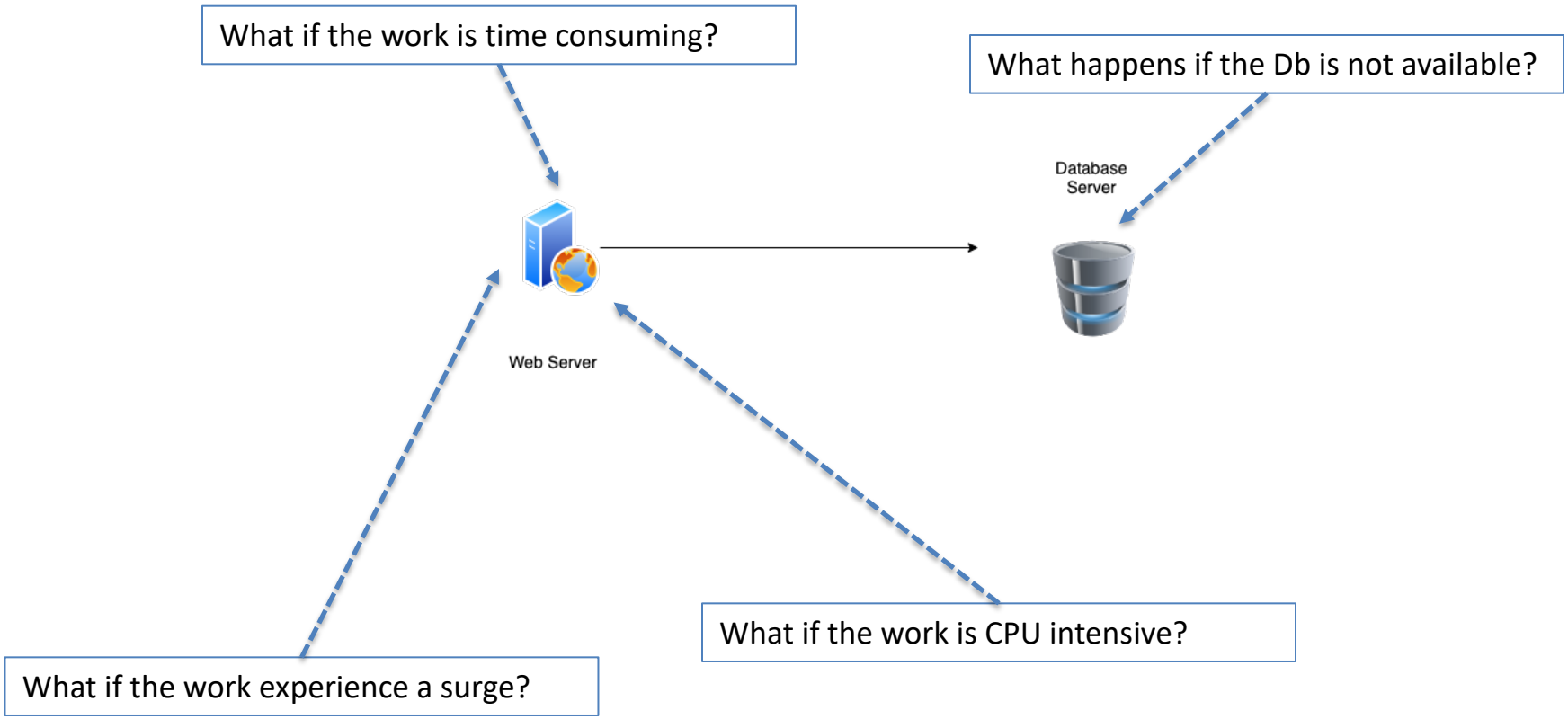
Performance and Scalability

Availability

Maintainability

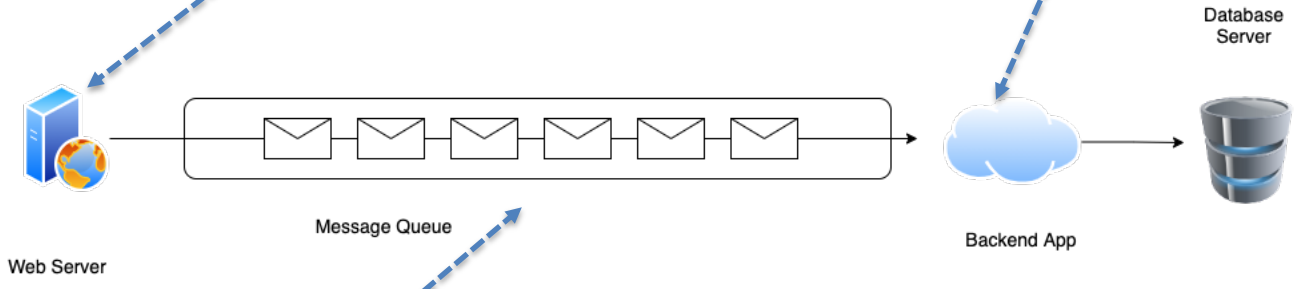
Inherent Distribution

# Example: Task Queues


What if the work is time consuming?
What happens if the Db is not available?
Database Server
Web Server
What if the work is CPU intensive?
What if the work experience a surge?

A backend application can perform long-running or CPU intensive work, allowing the web server to service new requests.

The web server puts the work on a queue

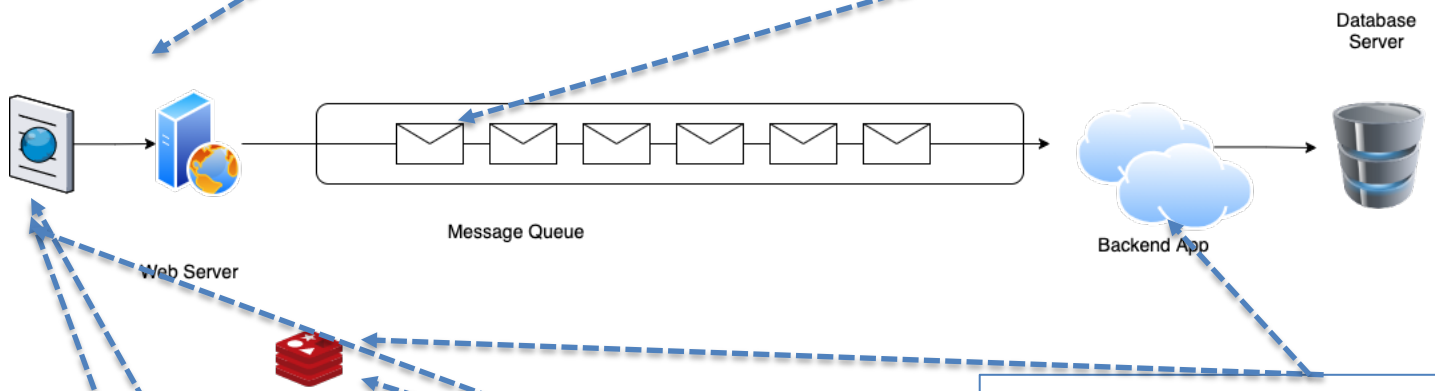



The queue stores work until we are ready to consume it. We can *throttle* to prevent surges.

We can scale out the backend services using a competing consumers approach, to ensure the queue does not backup.

We return 202 Accepted – we have your work request, and won't lost it.

We enqueue a work item for the request.



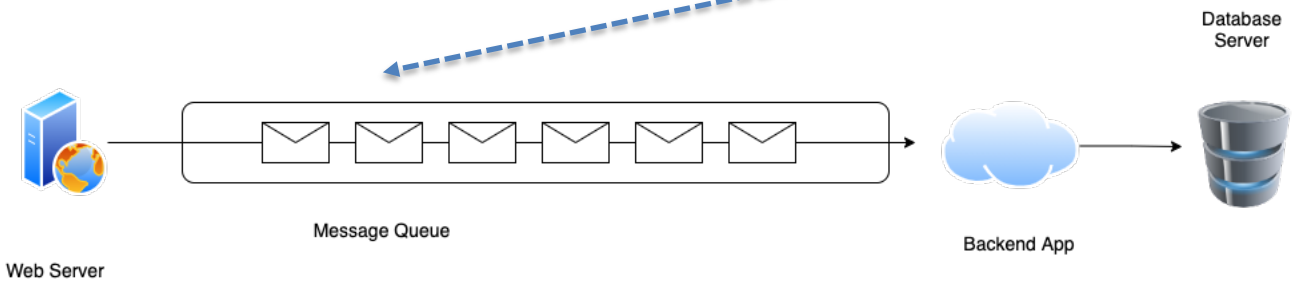

Backend app does work at sustainable pace, and updates KV store if required.

We return a link header, so you can monitor progress.

Link to a progress page, backed by a KV store where we note progress

Link to Resource – 404 until created

This general technique is known as Decoupled Invocation – we separate building the command from executing it.



# Decoupled Invocation Pattern

Use Decoupled Invocation. A producer puts a message onto a queue at the service endpoint. A consumer reads messages from the queue.

The queue stores messages for eventual processing. If the queue is durable we gain guaranteed delivery, and at-least once guarantees.

If the rate of arrival at the endpoint is unpredictable, the queue acts as a buffer that makes it possible to predict the rate of consumption.

This makes it simpler to do capacity planning because peaks of requests are smoothed out by the queue.

The consumer must be able to control the rate of processing, otherwise a spike is simply passed down the wire.

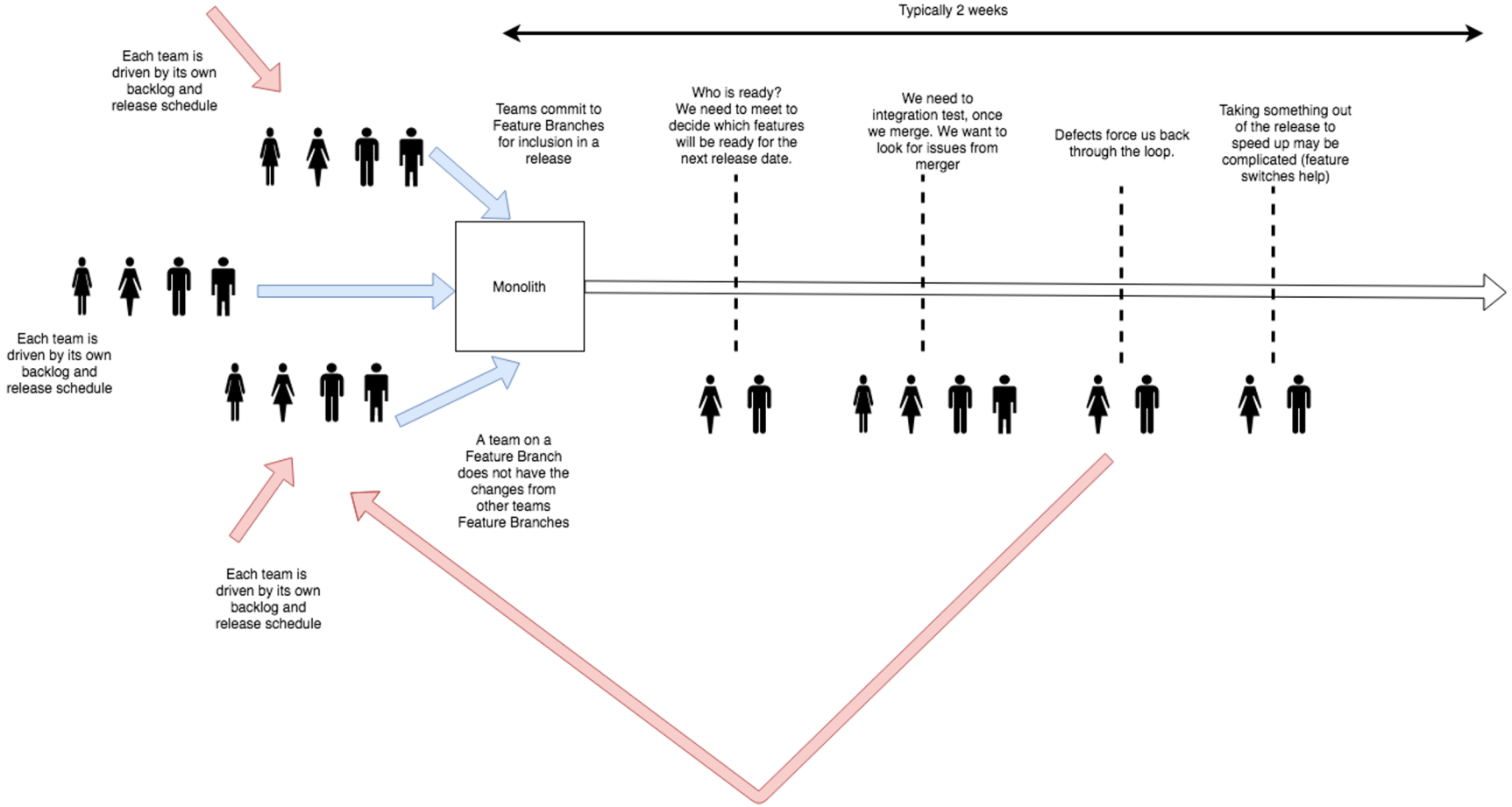
# Example: Microservices

# It’s all about velocity!!!

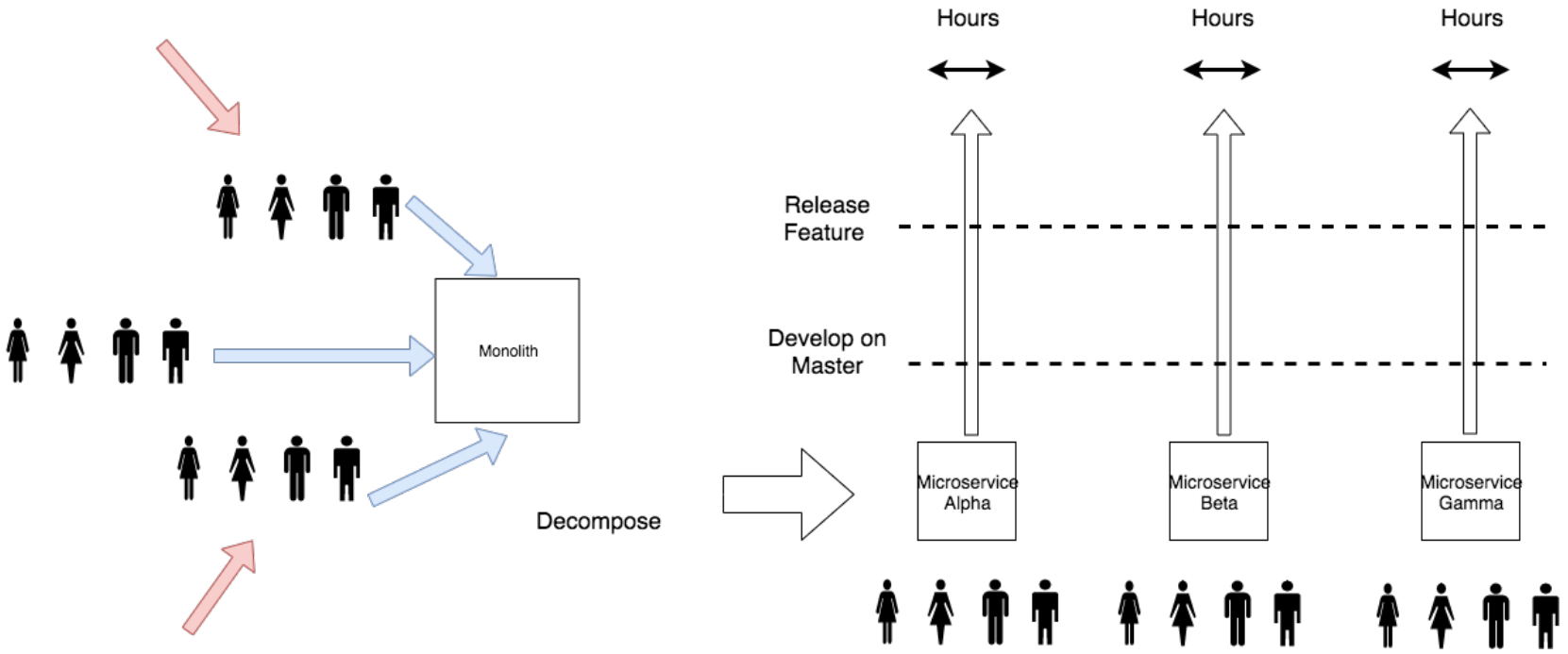
“Speed wins in the marketplace”

Adrian Cockcroft, former lead architect at Netflix

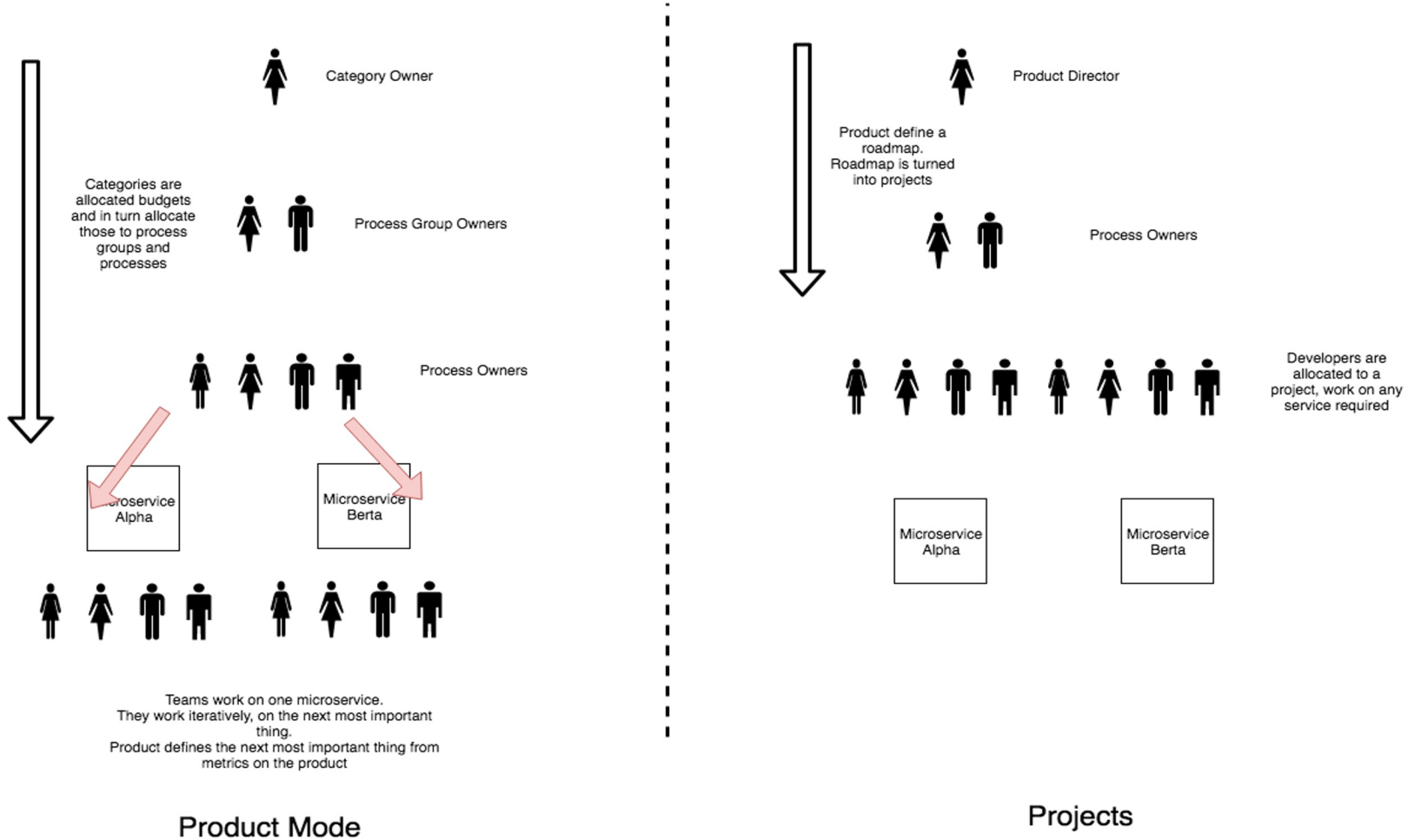
# Monoliths Do Not Scale To Many Teams!



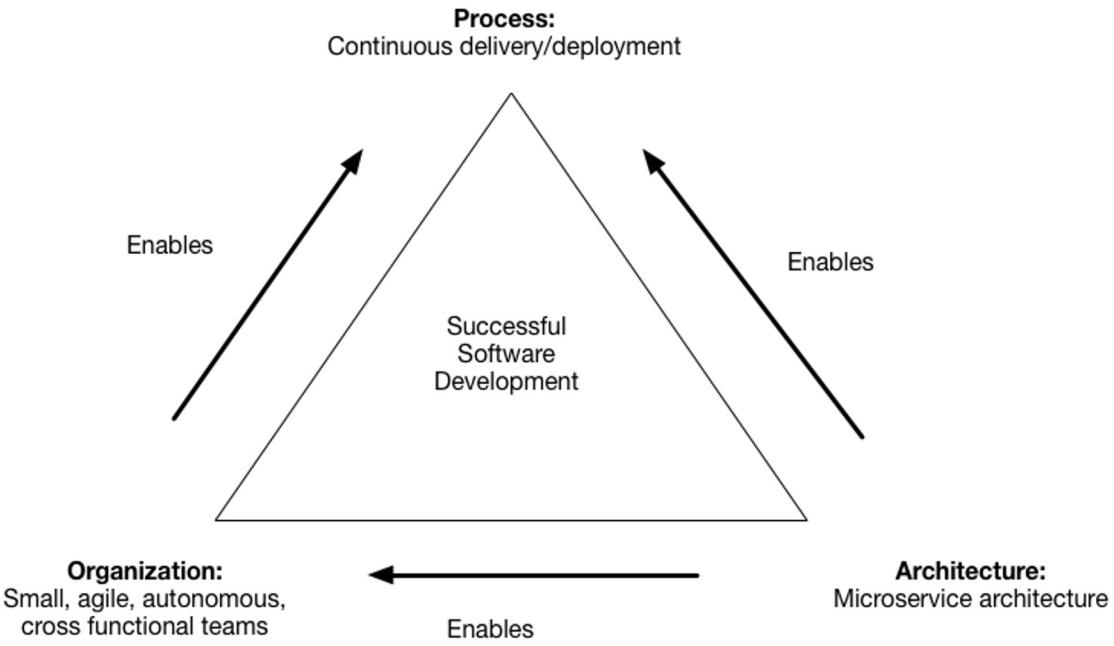
# Microservices let us scale an organisation



# Product Mode

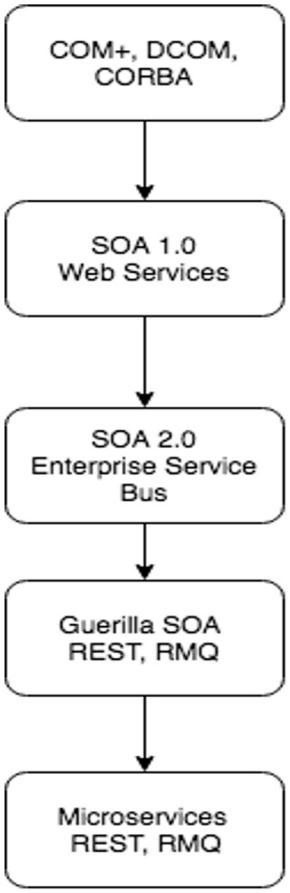


# Microservices enable agility




https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

# A Brief History of Microservices




Distributed Object Models dominate, a merger of RPC with OO

Don Box at Microsoft creates SOAP and the 4 tenets of SOA, launching the popularity of Web Service based SOA

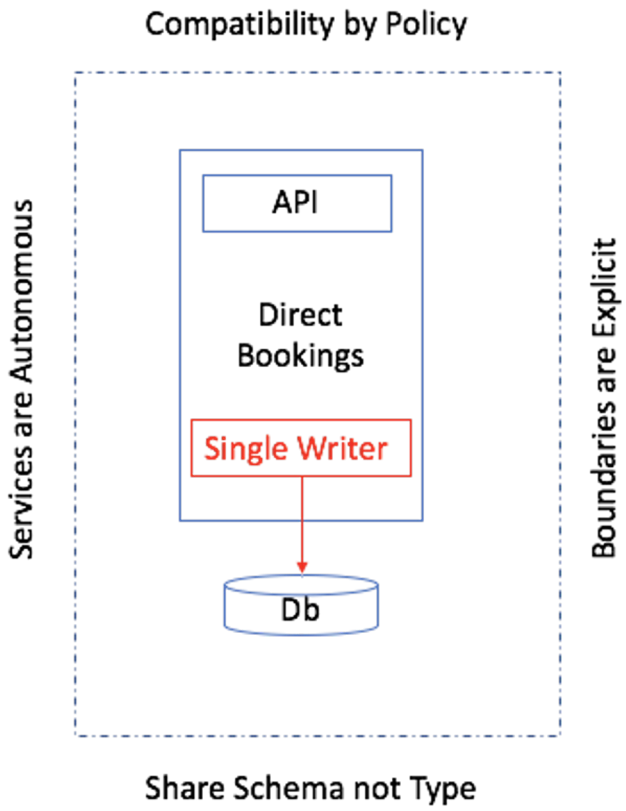
Events emerge as the key to loosely coupled services and vendors quickly own the middleware space

Jim Webber tries to break SOA away from centralized governance and the ESB favouring a bottom up approach using lightweight middleware or REST

Netflix under Adrian Cockcroft pioneers the Fine-Grained services model; Martin Fowler and James Lewis document as microservices

The definition of Microservices, we use comes from the paper by Martin Fowler and James Lewis

# Microservices are partitions of software



# The Price of Distribution

# Fallacies of Distributed Computing

The network is reliable.

Latency is zero.

Bandwidth is infinite.

The network is secure.

Topology doesn't change.

There is one administrator.

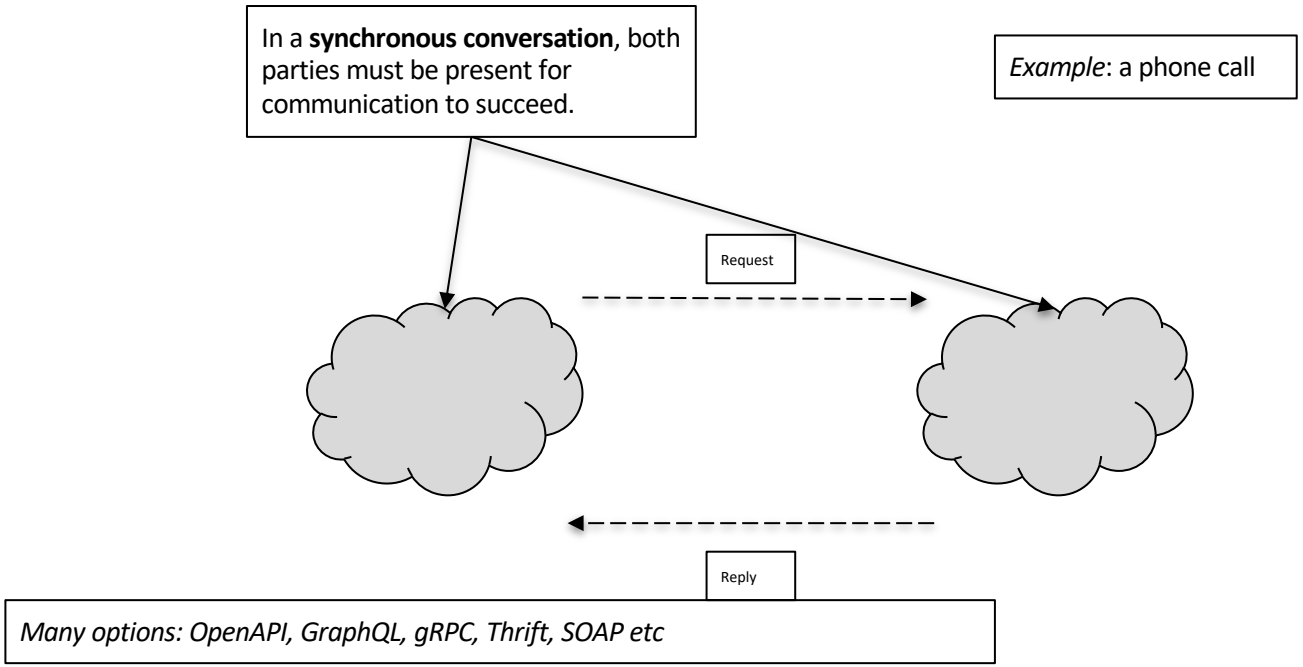
Transport cost is zero.

The network is homogeneous.

How do we communicate between microservices?

# 2.INTEGRATION STYLES

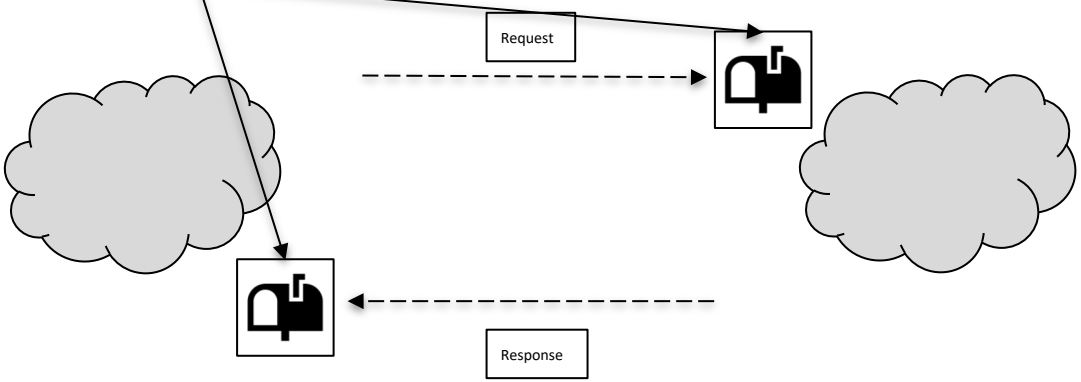
Synchronous Conversation



Asynchronous Conversation

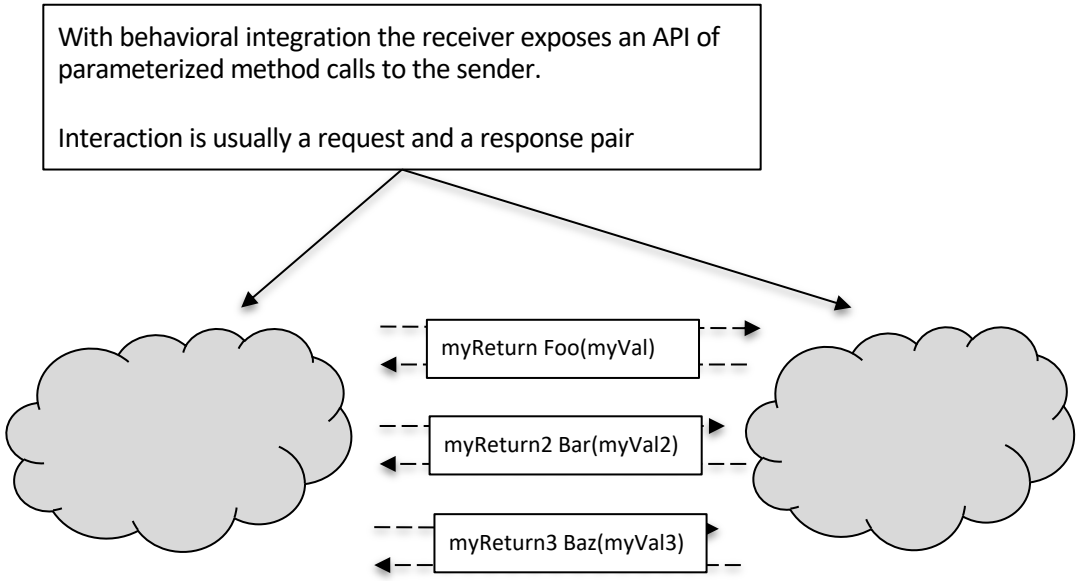
In an a**synchronous conversation**, the *receiver* does not need to be present at the time the *sender* communicates with them, using **store and forward** to pick up the message later.

*Example*: snail mail

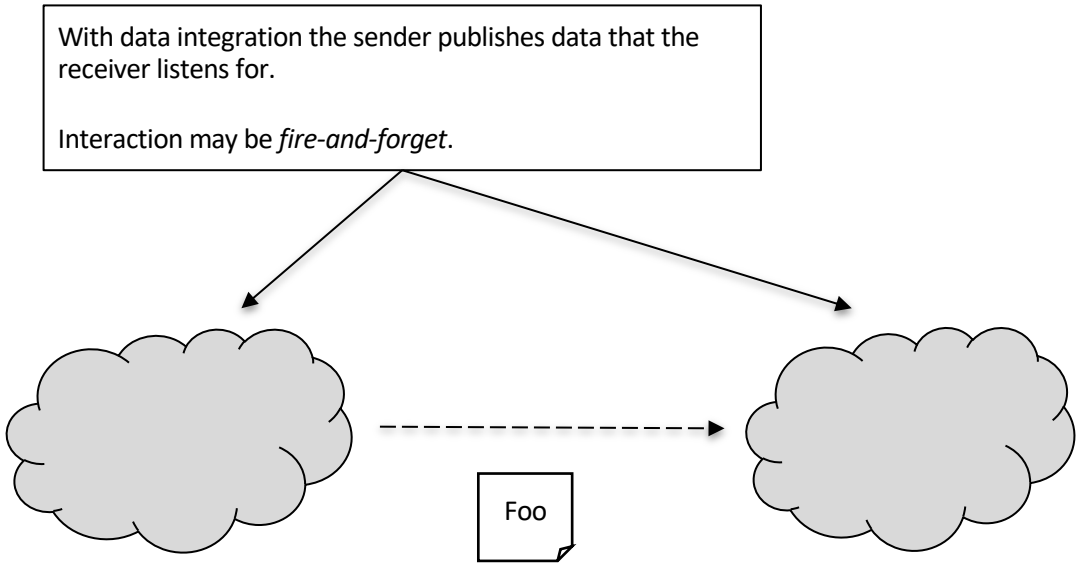



*Many options: SQS, Kafka, AMQP 0-9-1 (RMQ), AMQP 1-0, MQTT, S3*

Behavioral Integration

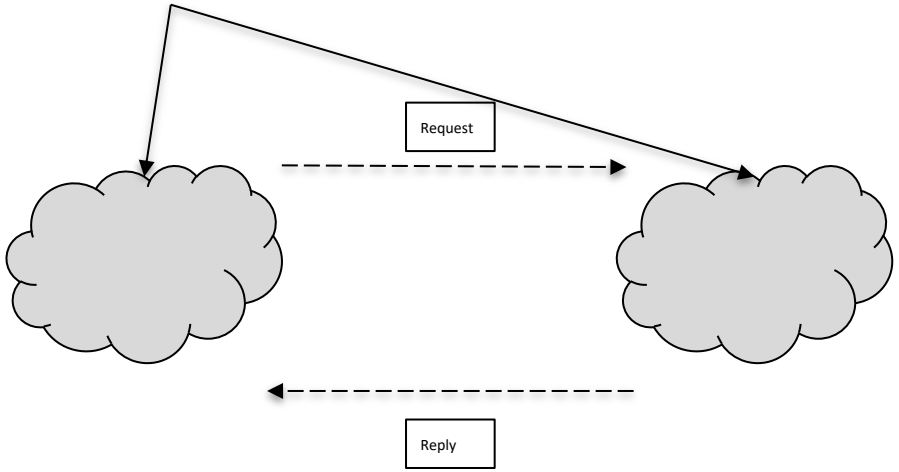


Data Integration



## Temporal Coupling

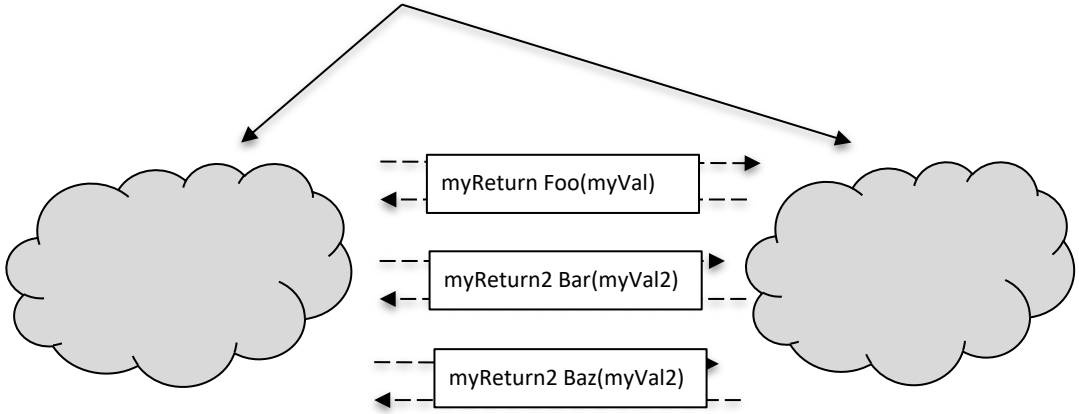
If both parties must be present to succeed we say they are *temporally coupled.* The availability of one has an impact on the availability of another.



## Behavioral Coupling

Where two systems use behavioral integration, they are coupled by the set of calls in the interaction. We find it hard to change who we talk to; we find it hard to change how we talk about something.

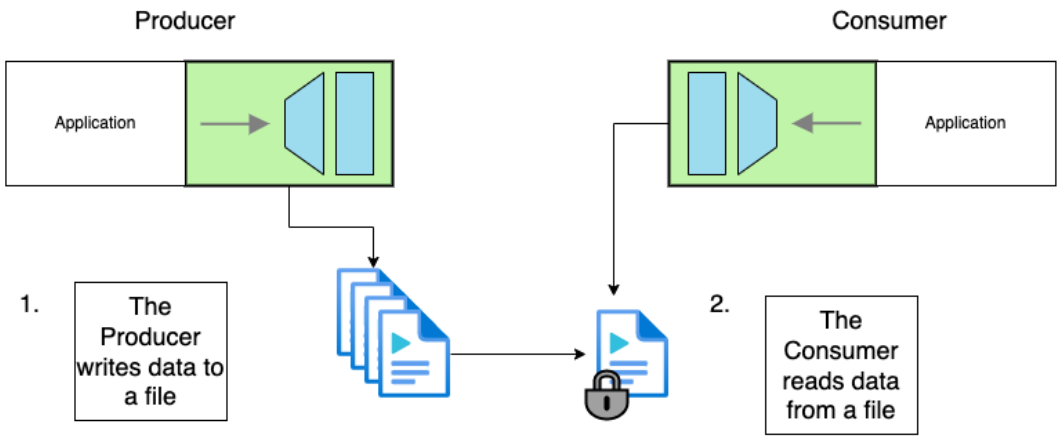
We say such systems are **behaviorally coupled**



## File Transfer

Asynchronous Conversation

Data Integration



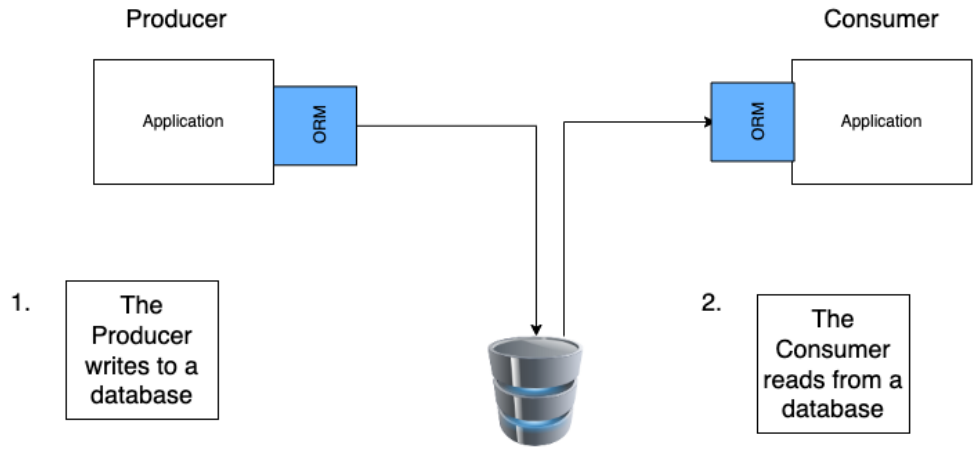

The consumer has to decide how often to poll, depending on acceptable latency of a producer write, or we have to have a notification to tell it to read.

## Shared Database

Asynchronous Conversation

Data Integration

Behavioral Coupling




The producer and consumer depend upon the database schema - they are coupled such that schema changes rippled from one to the other

Busy tables may become bottlenecks as databases hit scale up limits. (Requires read-only replicas or partitioning).

Creating a unified schema that meets the needs of all readers & writers can become a challenge - resulting in over-abstraction or too many fields.

# Remote Procedure Call

Synchronous Conversation

Behavioral Integration

Behavioral Coupling

Temporal Coupling


1. The Client calls a remote procedure on the Server

2. The Server listens for calls, actions them, and returns results

The client uses a 'stub' to abstract the details of the call to the server into a remote call

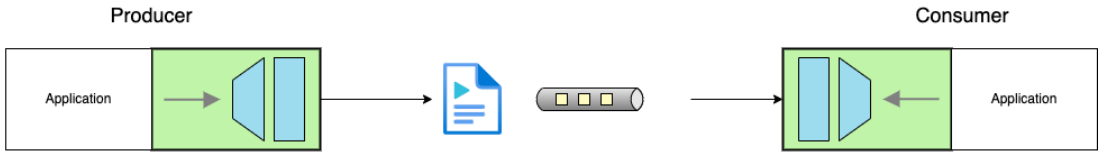
The stub needs to know how to connect to the proxy - typically a distributed KV store contains a discoverable list of servers

The server exposes functionality that the client can call. It may be stateless, in that no state is maintained between calls, or stateful, where there is a session. The proxy turns a network call into a local procedure call

Asynchronous Conversation

Data Integration

## Messaging




1. The Producer writes a message to a channel

2. The Consumer reads a message from a channel

The producer and consumer must agree the content type and format of the message.

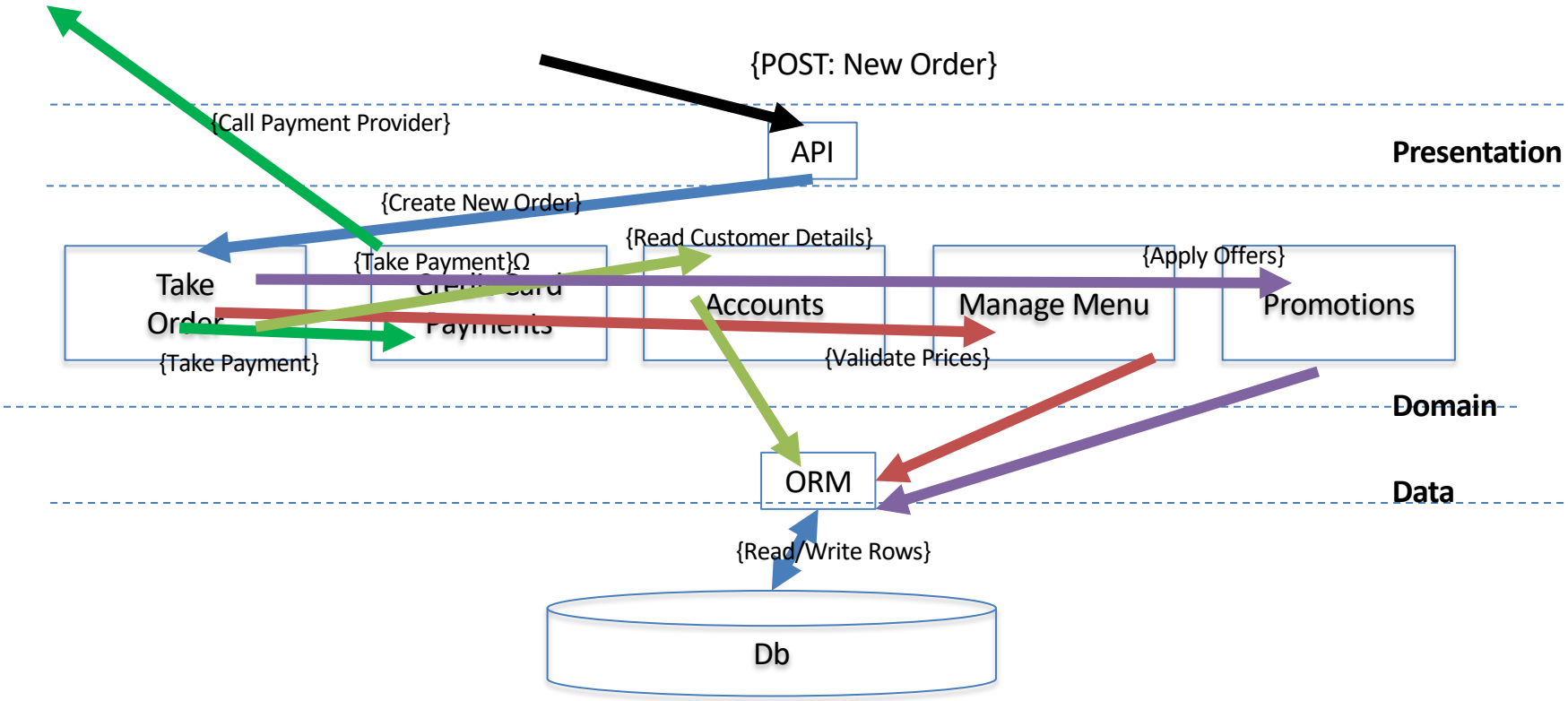
The consumer has to decide either to poll, with greater control of back-pressure traded-off against higher latency, or accept a push.

If there are multiple consumers, then the consumer needs to lock the current message, to ensure other readers don't try to process it.

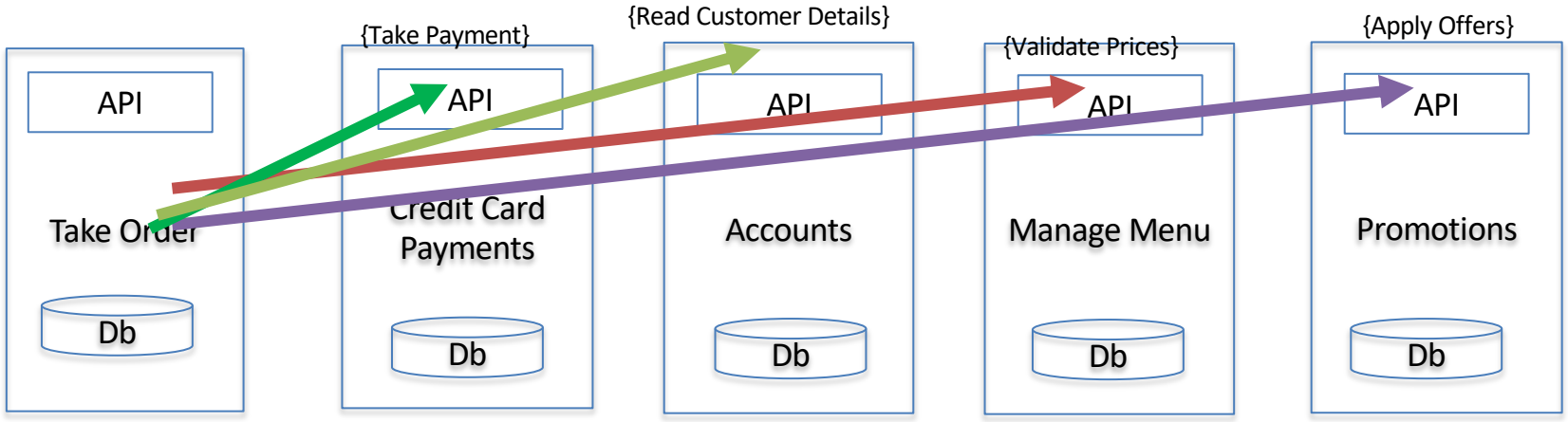
Integrating using RPC or REST

# 3.REQUEST DRIVEN ARCHITECTURES

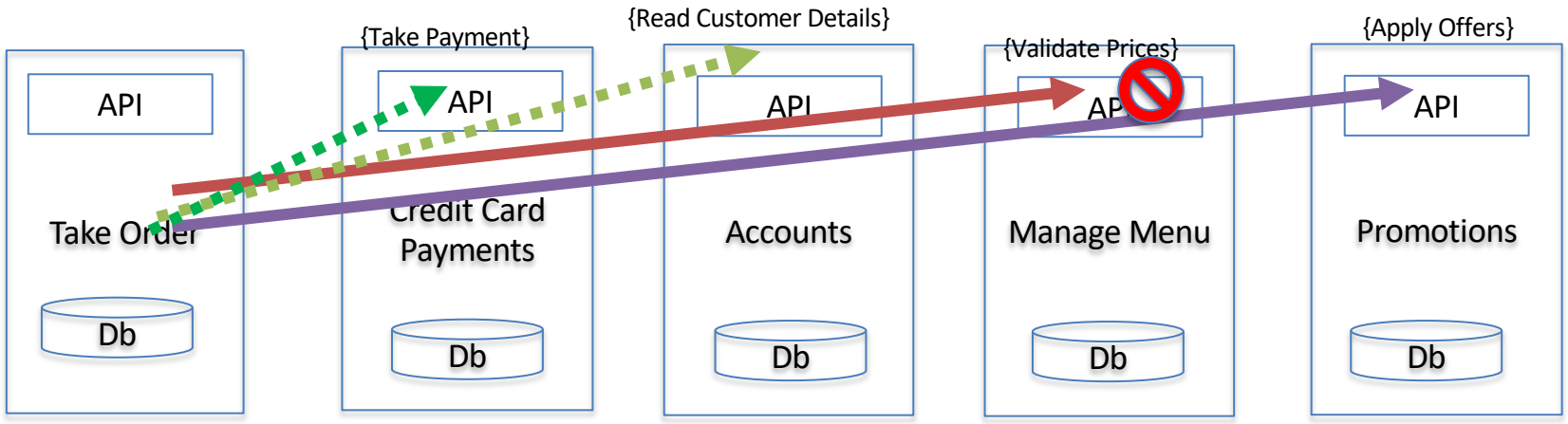
# Getting work done in a Monolith



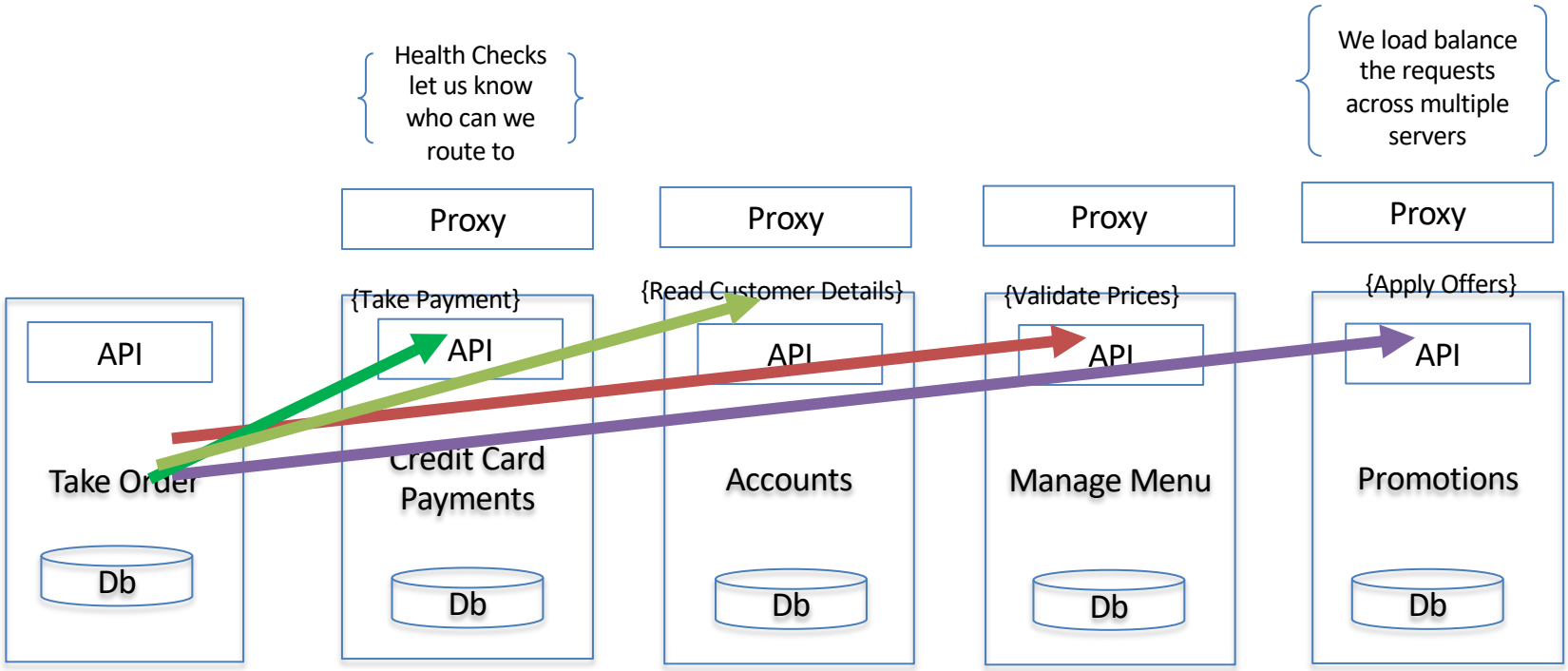
# Microservices



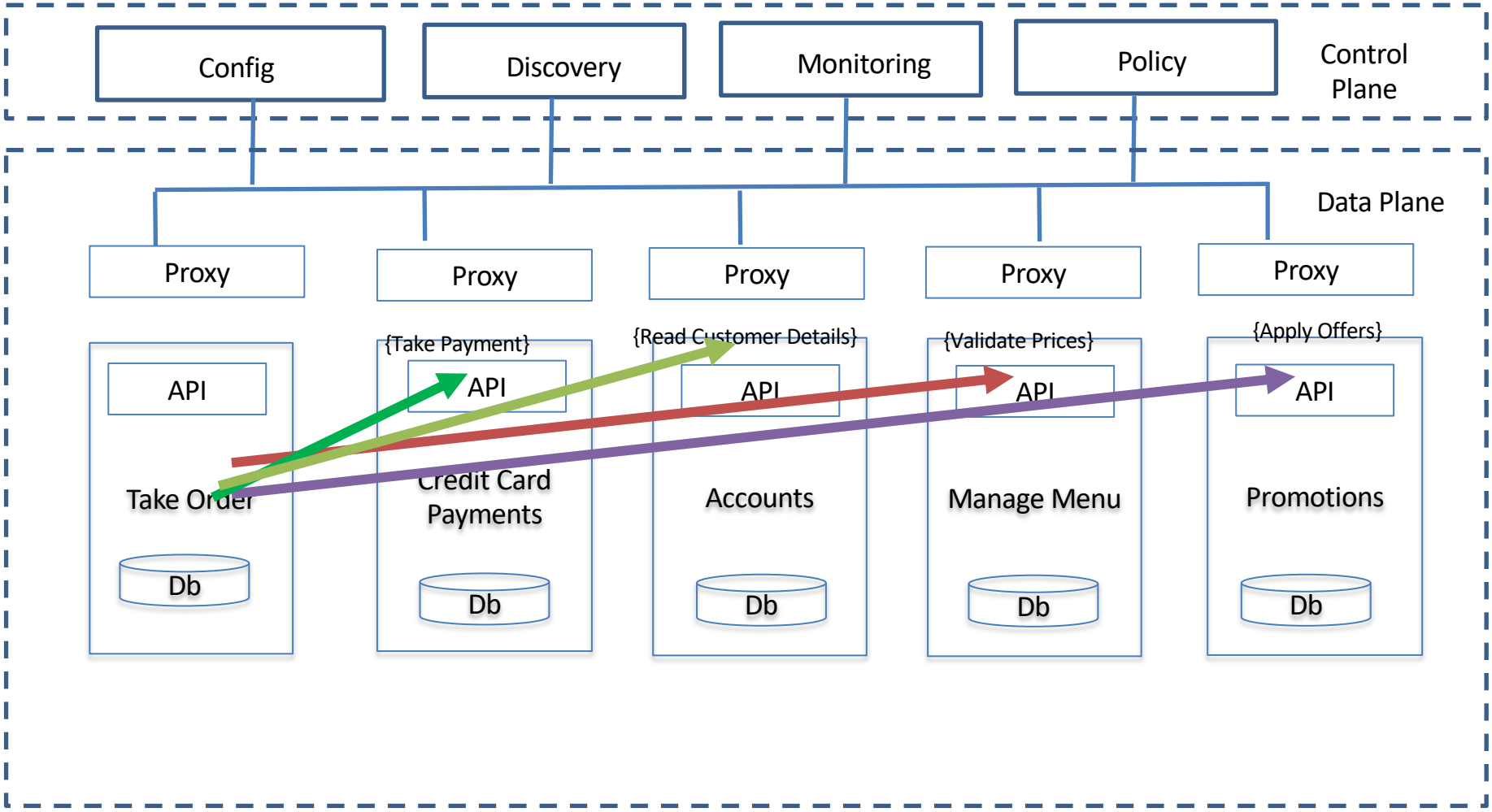
# Temporal Coupling



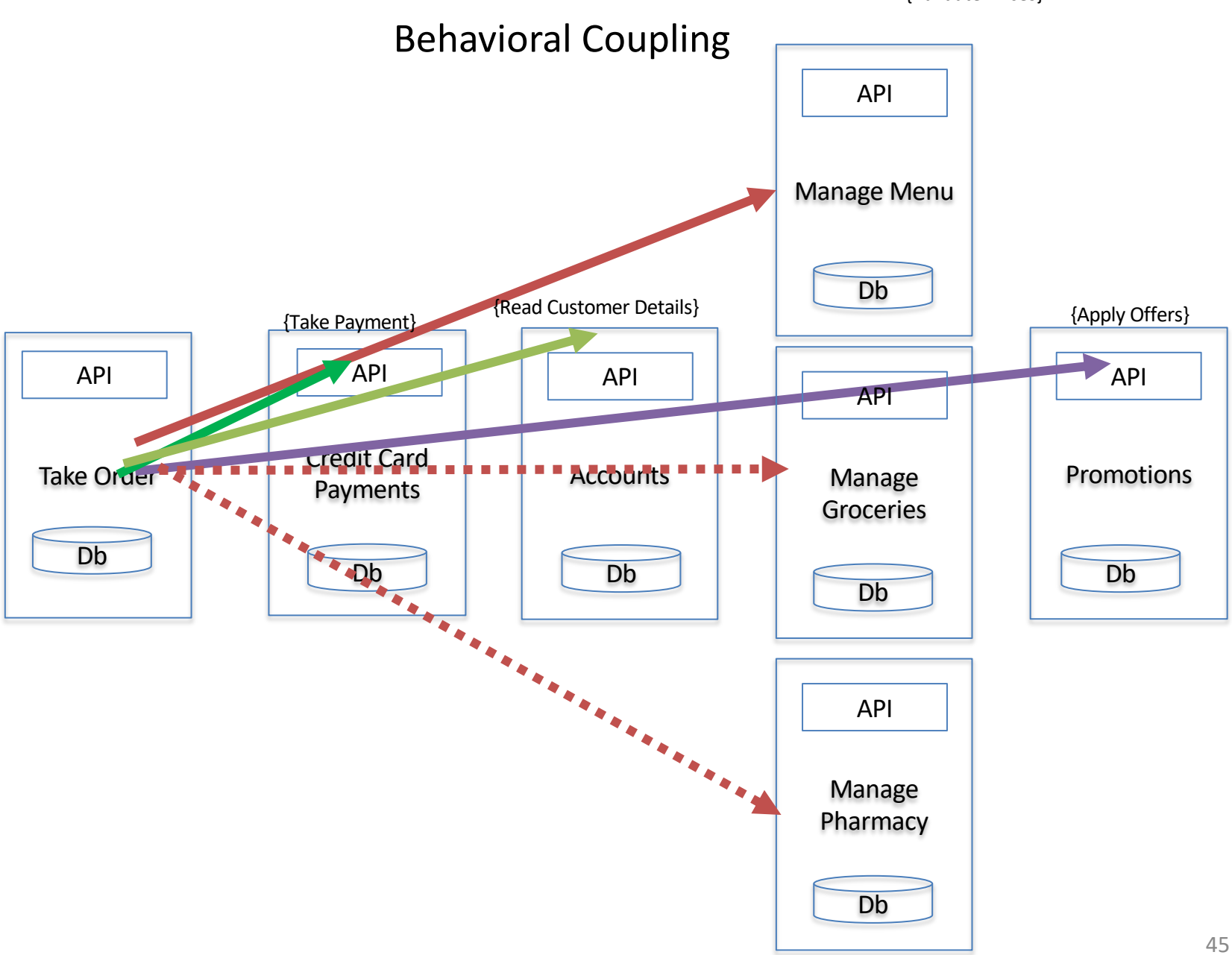
# Using A Proxy for Availability



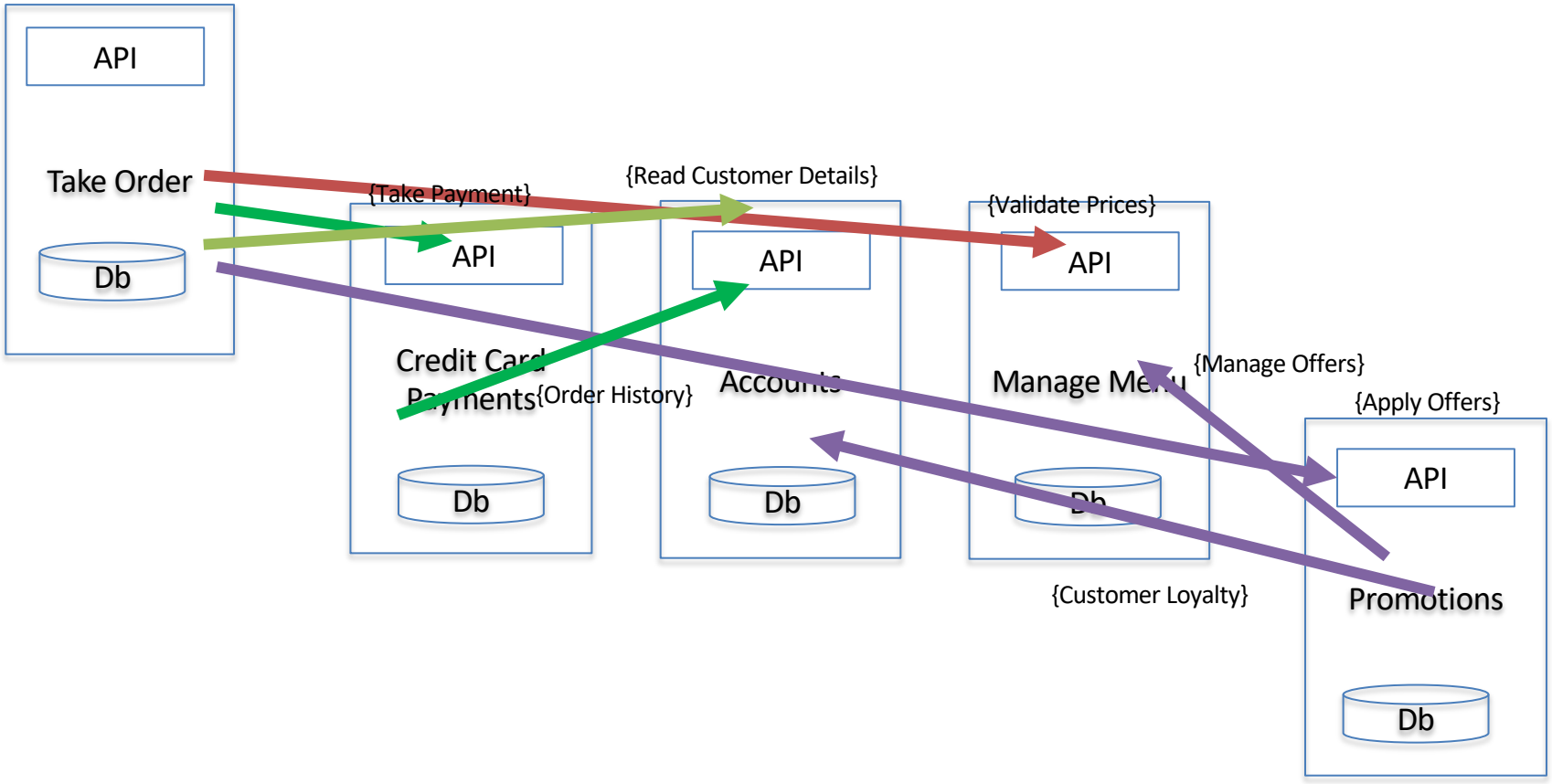
# Service Mesh



# Behavioral Coupling



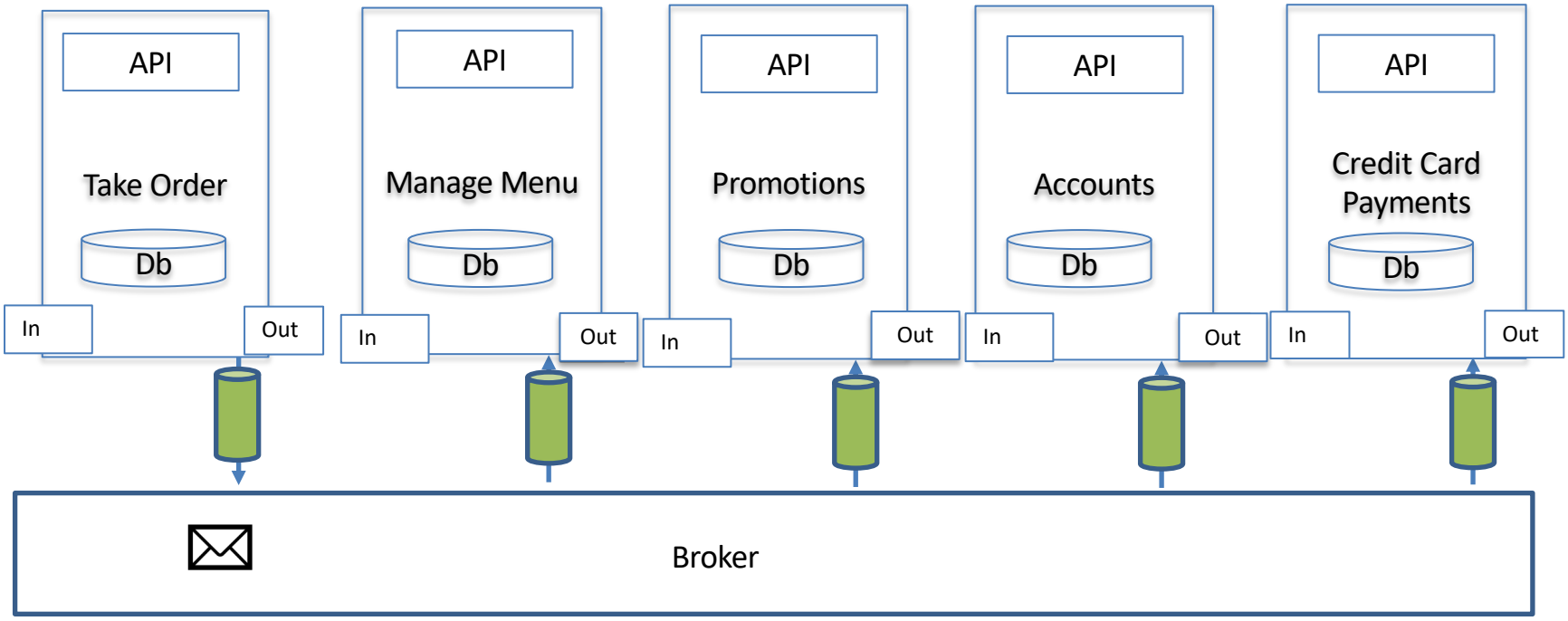
# N(N-1)/ 2 connections



Integrating using events

# 4.EVENT DRIVEN ARCHITECTURES

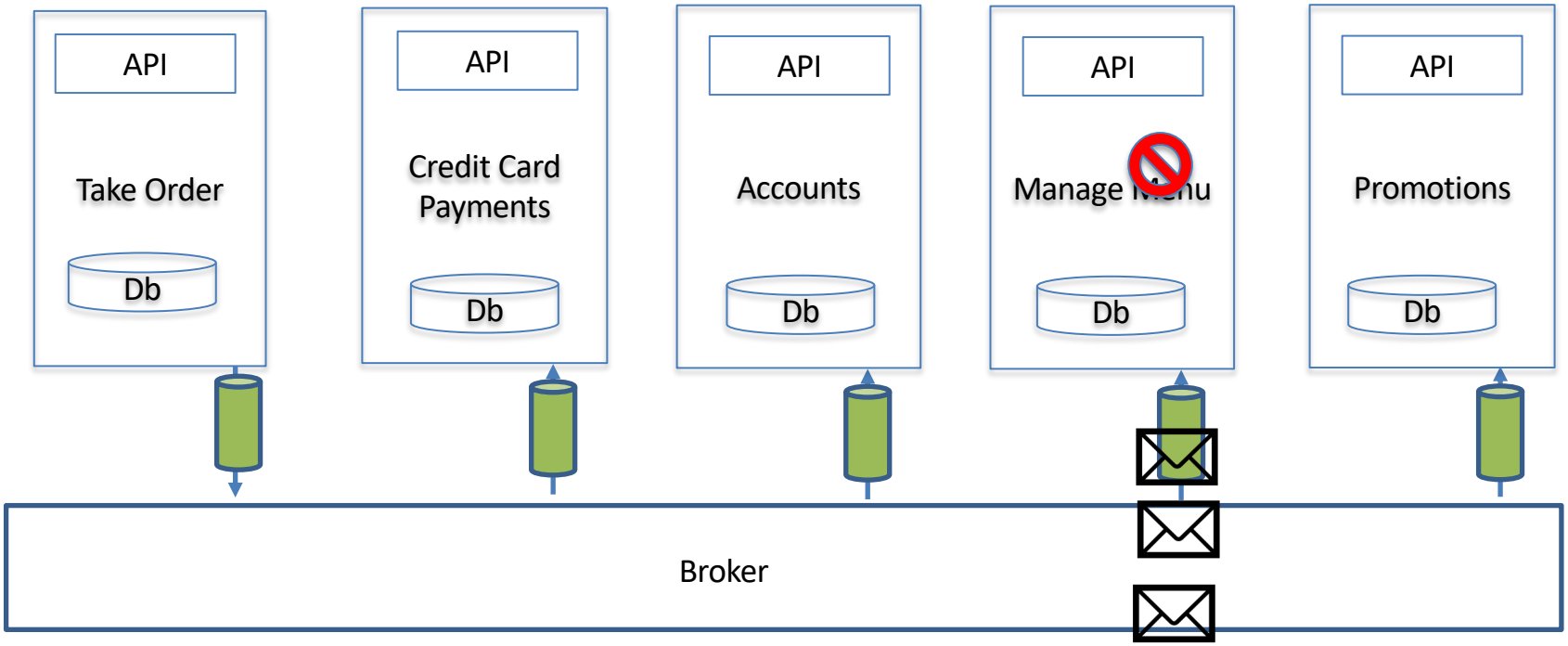
# Event Driven Architecture



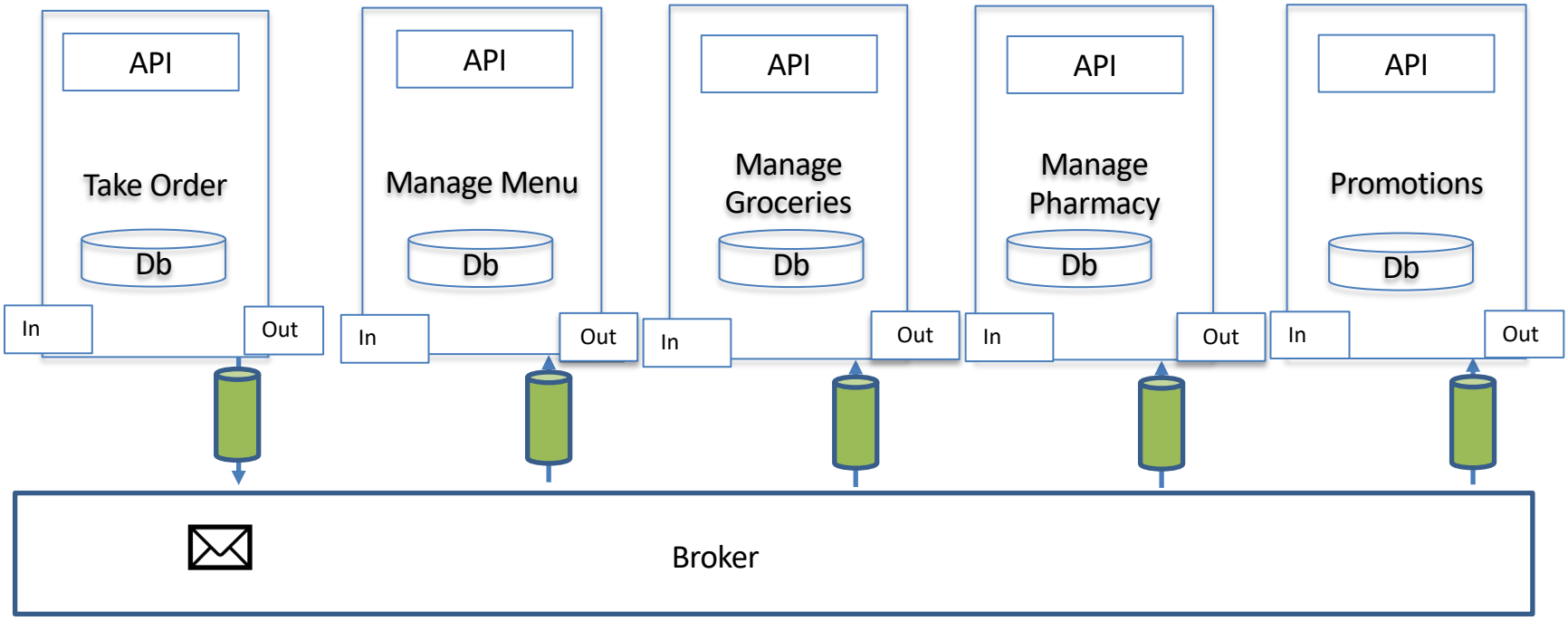

Consumer has no notion of producer, just a topic on the broker

Channels subscribes to a 'topic' on the broker

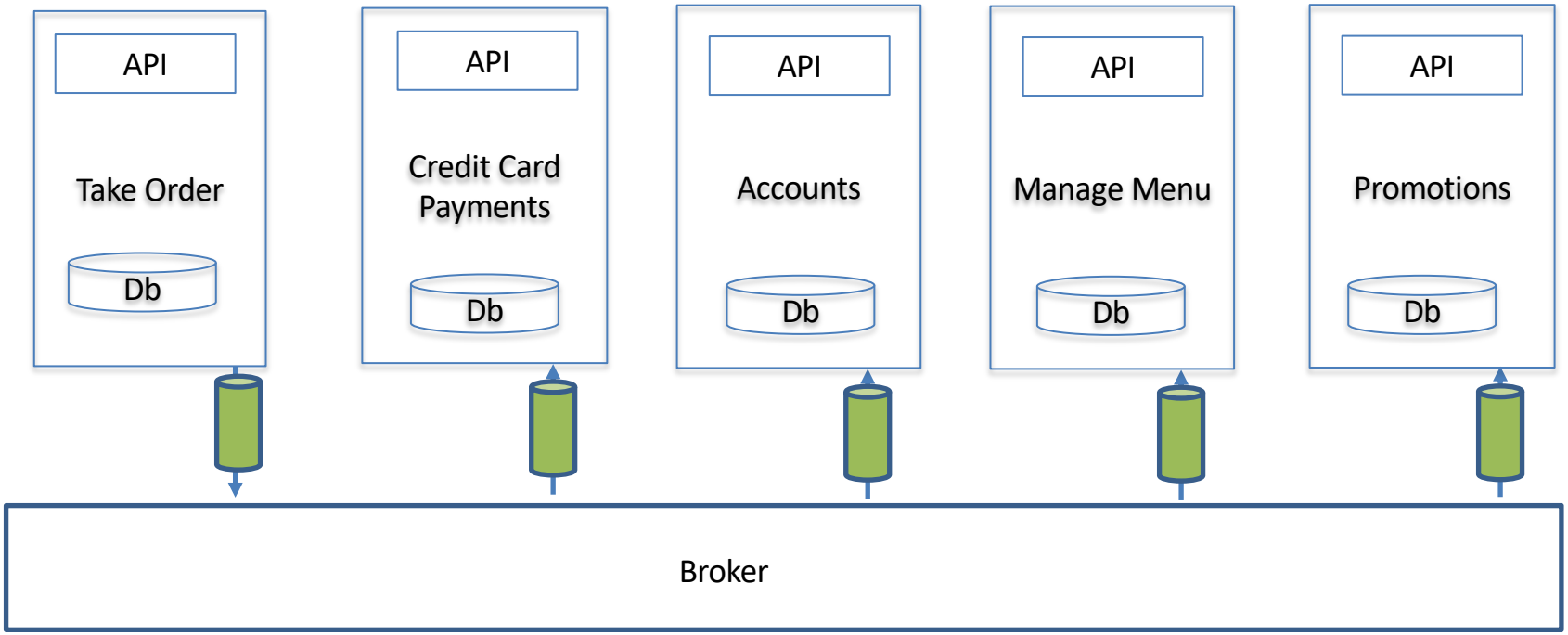
# Guaranteed, At Least Once



# Publish Subscribe

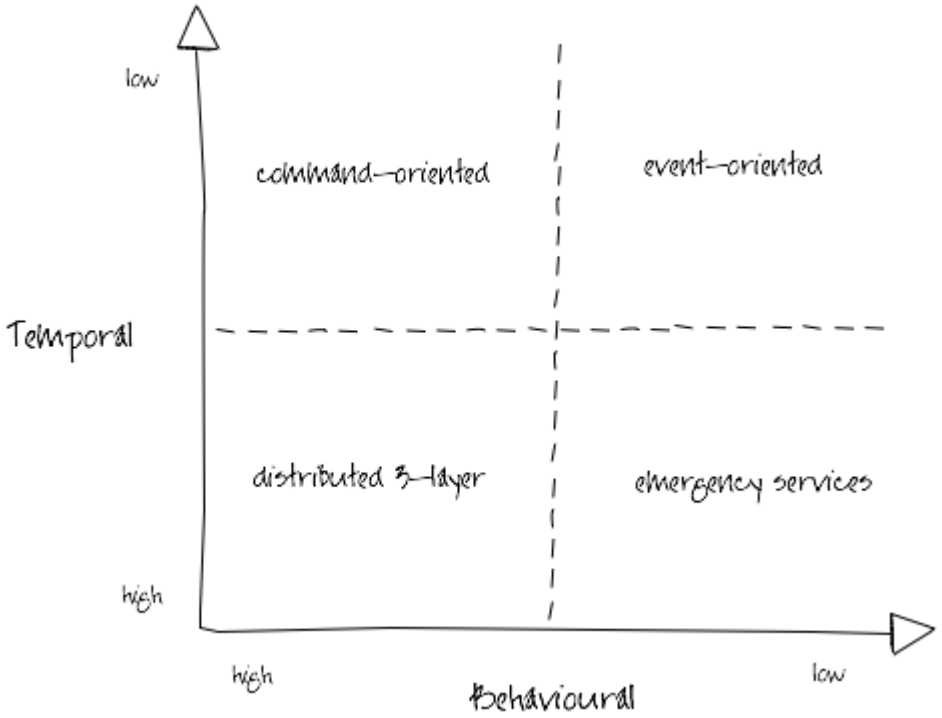


# Hub and Spoke



Integrating using events

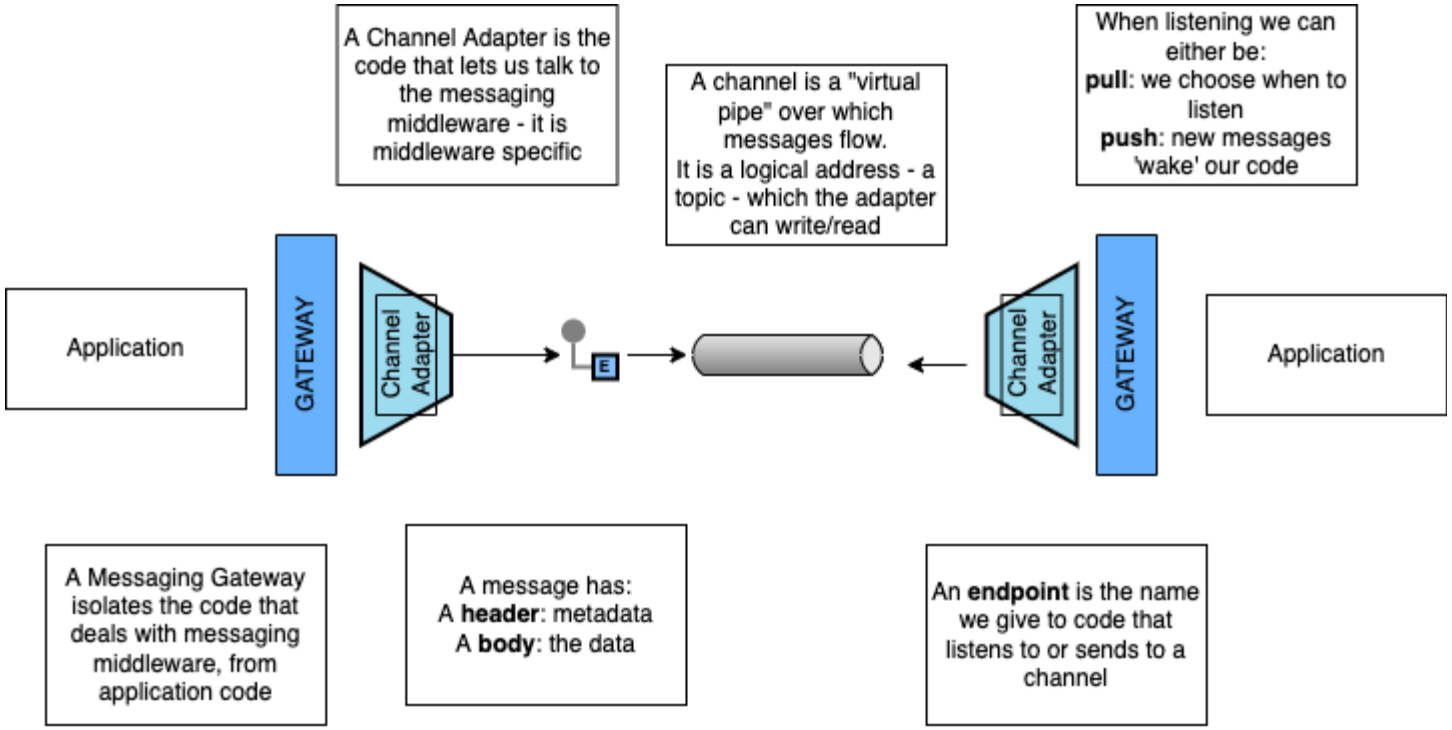
# CONTRASTING REQUESTS AND EVENTS



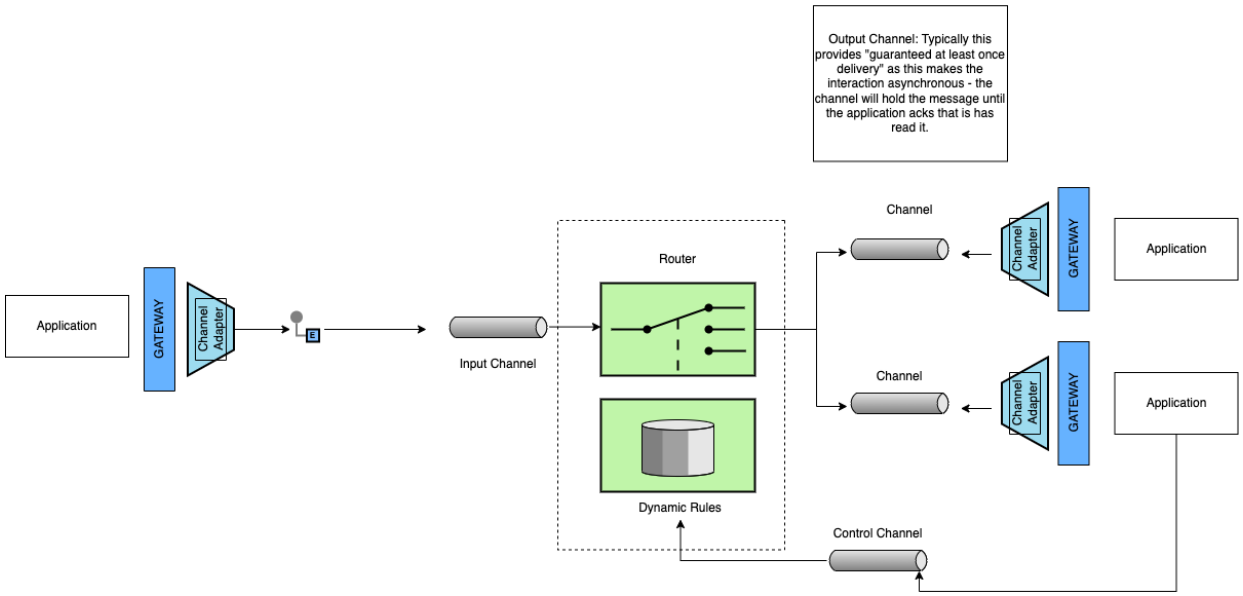

Ian Robinson: http://iansrobinson.com/2009/04/27/temporal-and-behavioural-coupling/

Integrating using events

# 5.MESSAGING PATTERNS


A Channel Adapter is the code that lets us talk to the messaging middleware - it is middleware specific
A channel is a "virtual pipe" over which messages flow.
It is a logical address - a topic - which the adapter can write/read
When listening we can either be:
**pull**: we choose when to listen
**push**: new messages 'wake' our code
Application
GATEWAY
Channel Adapter
Channel Adapter
GATEWAY
Application
A Messaging Gateway isolates the code that deals with messaging middleware, from application code
A message has:
A **header**: metadata
A **body**: the data
An **endpoint** is the name we give to code that listens to or sends to a channel

# Messaging with A Broker



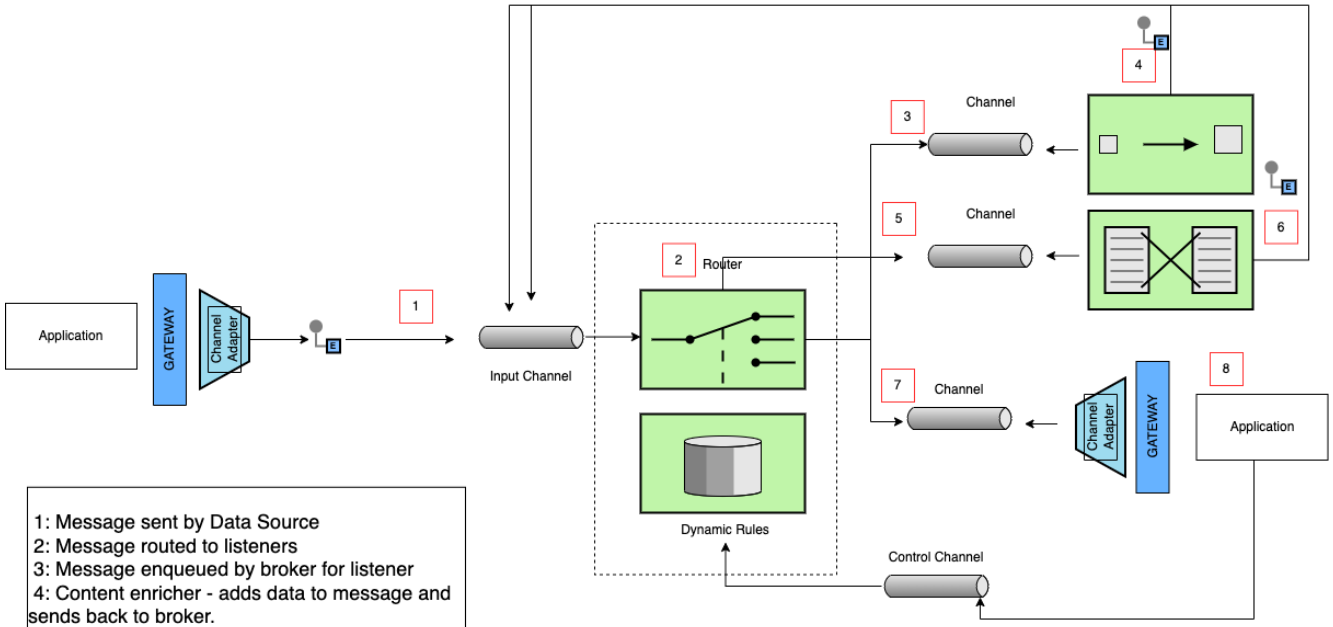

**Input Channel**: Typically applications connect to a broker via a TCP/IP channel that is it does not have "*guaranteed at least once delivery*"

Fundamentally a broker is a router, that supports one or more routing strategies. You send a message to a broker and it is then routed to consuming applications according to the rules. Thus the producer only needs to know the broker, not all the consumers

A broker is typically dynamic - that is we can use code to register our intent to receive messages according to a set of rules - this topic for example - at runtime.

A control channel just implies - you an register at runtime.

# Pipes and Filters




1: Message sent by Data Source
2: Message routed to listeners
3: Message enqueued by broker for listener
4: Content enricher - adds data to message and sends back to broker.
5: Message routed and enqueued by broker for listener
6: Message translated - changes shape and sends back to broker
7: Message routed and enqueued for data sink
8: Data sink processes the final message

# 5.1 TYPES OF MESSAGES

# Command Message

Use a Command Message to reliably invoke a procedure in another application

Uses the well-established pattern for encapsulating a request as an object. The Command pattern [GoF] turns a request into an object that can be stored and passed around.

# Document Message

Use a Document Message to reliably transfer a data structure between applications.

The receiver decides what, if anything, to do with the data

# Event Message

Use an Event Message for reliable, asynchronous event notification between applications.

The difference between an Event Message and a Document Message is a matter of timing and content. An event's contents are typically less important.

# Request-Reply

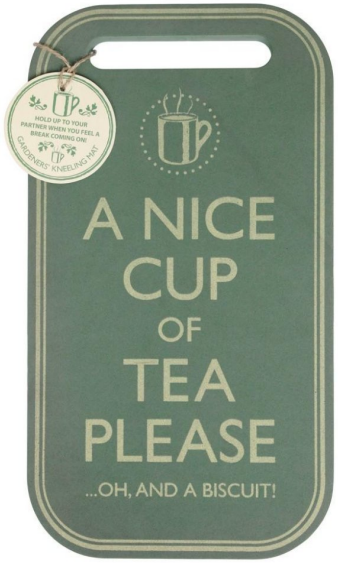
When an application sends a message, how can it get a response from the receiver?

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

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

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

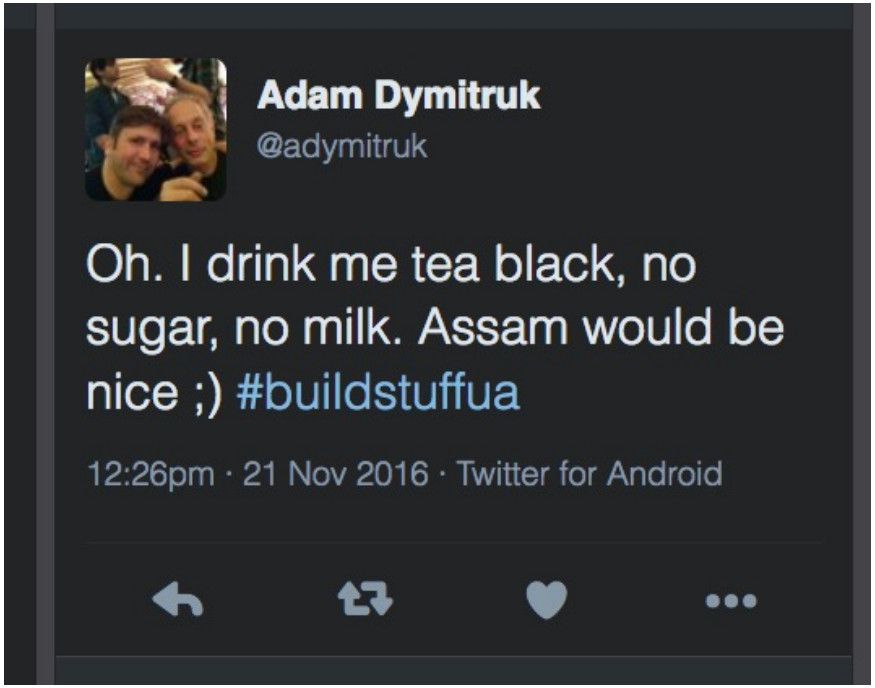
# Command or Event?



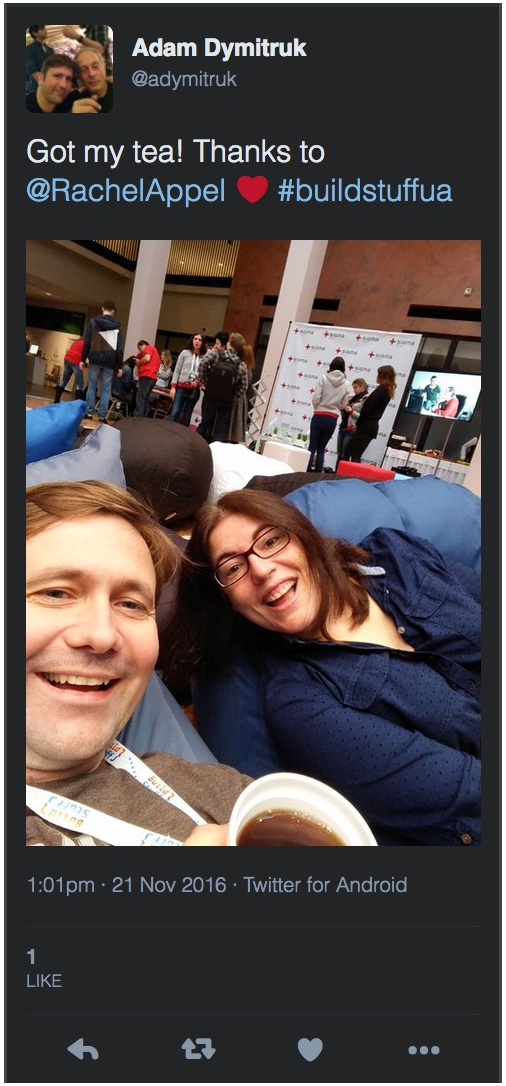

A Command


An Event


View original
Flag media
Adam Dymitruk @adymitruk
38m
I'm lying in the middle of the exhibition hall. Caught in a catch-22. Too tired/sleepy to get a strong tea. Somebody help? #buildstuffua


Adam Dymitruk
@adymitruk
Oh. I drink me tea black, no sugar, no milk. Assam would be nice ;) #buildstuffua
12:26pm · 21 Nov 2016 · Twitter for Android


Dylan Beattie @dylanbeattie 2h
Somebody @Buildstuffua please bring @adymitruk a black tea... he's in the middle of the exhibition hall and he won't stop complaining... :)
Adam Dymitruk @adymitruk
Oh. I drink me tea black, no sugar, no milk. Assam would be nice ;) #buildstuffua


Adam Dymitruk
@adymitruk
Got my tea! Thanks to
@RachelAppel ♥ #buildstuffua
1:01pm · 21 Nov 2016 · Twitter for Android
1
LIKE

What is a message?

# 5.2 A MESSAGE

# Message Construction

A message has a header and body

The body contains data for the consumer

The header contains metadata for any *filter* in the pipeline.

The header should indicate the format of the body

Break a large message into pieces as a Message Sequence

# 5.3 CHANNELS

# Channels

A virtual pipe that connects producer and consumer

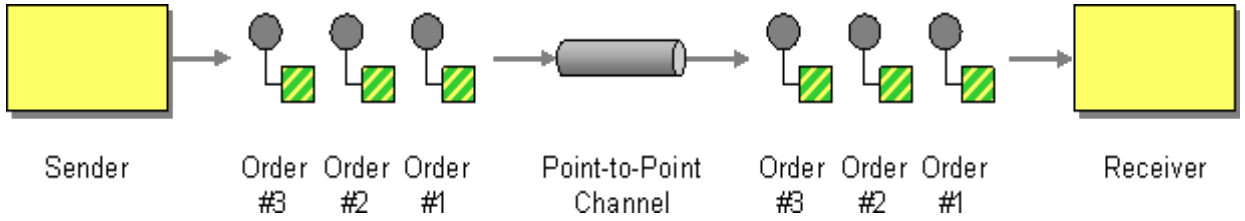
Logical Address (Topic or Routing Key)

Unidirectional

One-to-One or One-to-Many

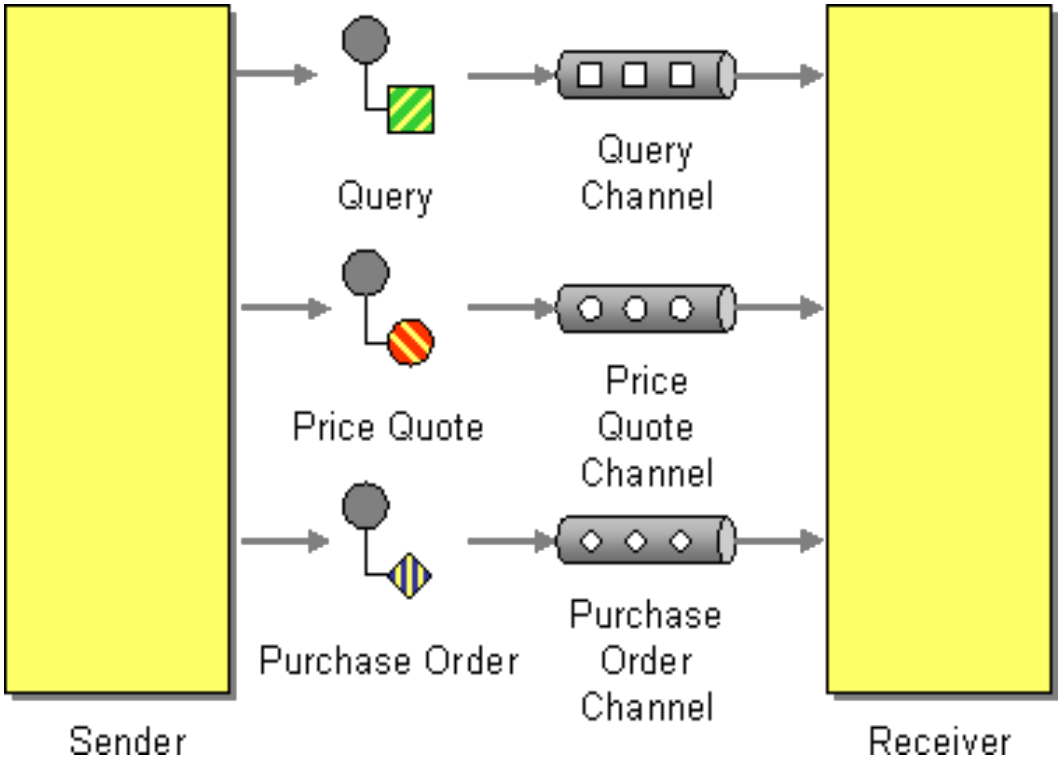
Messaging is a ‘pipe’ not a ‘bucket’.

# Point-to-Point Channel



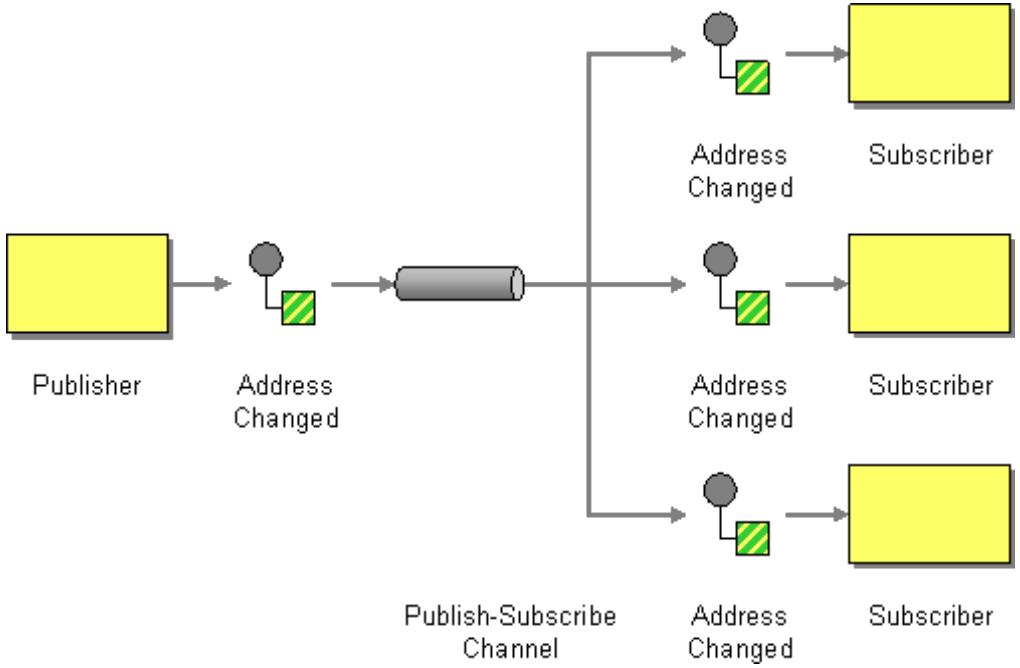

http://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html

# Datatype Channel



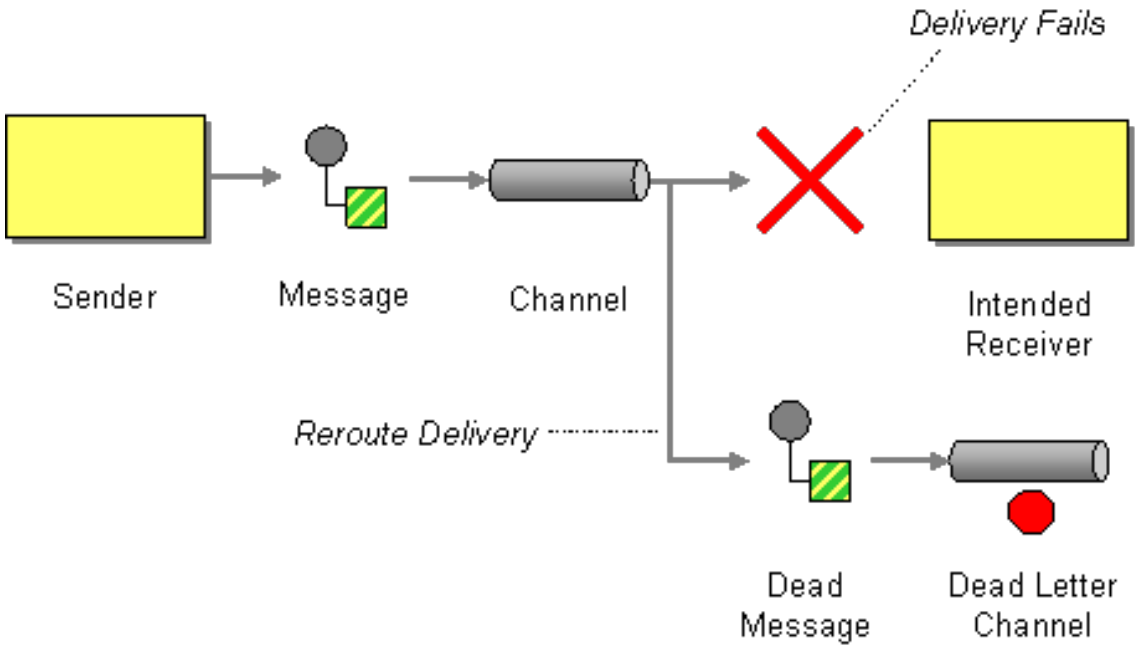

http://www.enterpriseintegrationpatterns.com/patterns/messaging/DatatypeChannel.html

# Publish-Subscribe Channel



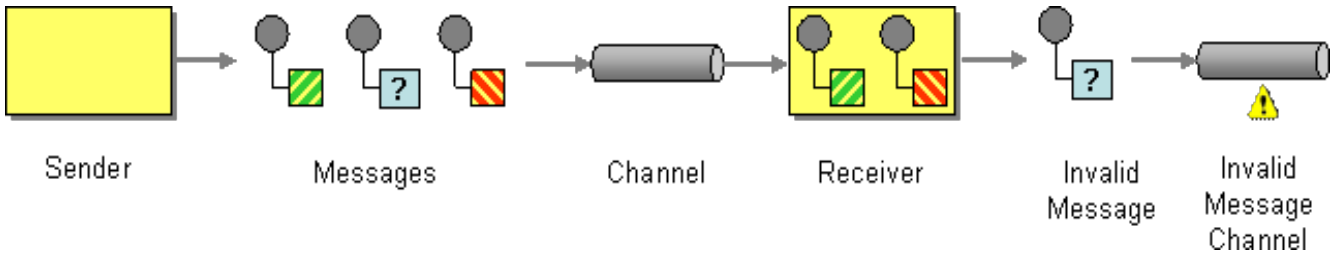

http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html

# Dead Letter Channel




http://www.enterpriseintegrationpatterns.com/patterns/messaging/DeadLetterChannel.html

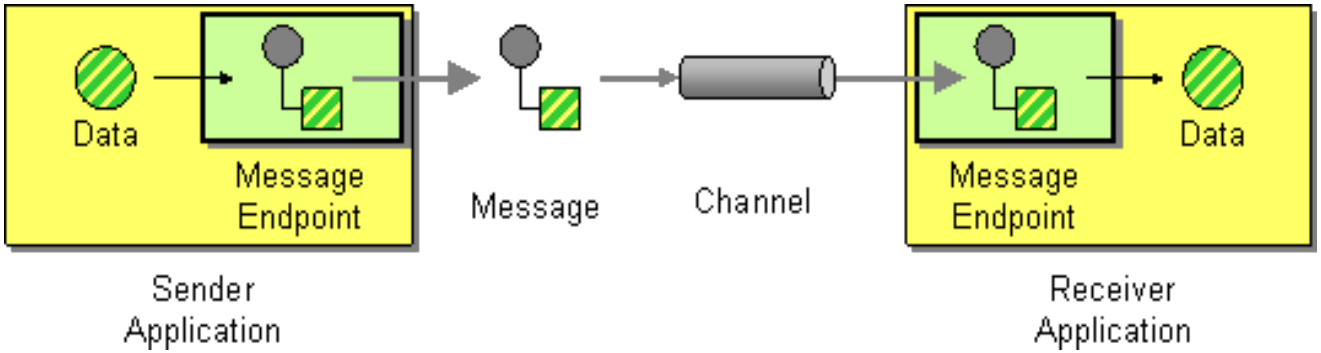
# Invalid Message Channel




http://www.enterpriseintegrationpatterns.com/patterns/messaging/InvalidMessageChannel.html

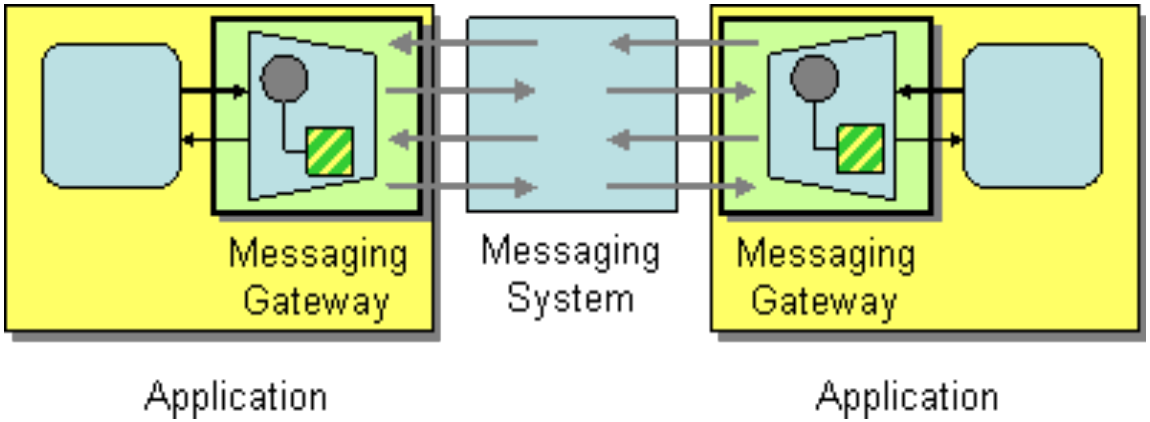
# 5.4 ENDPOINTS

# Message Endpoint



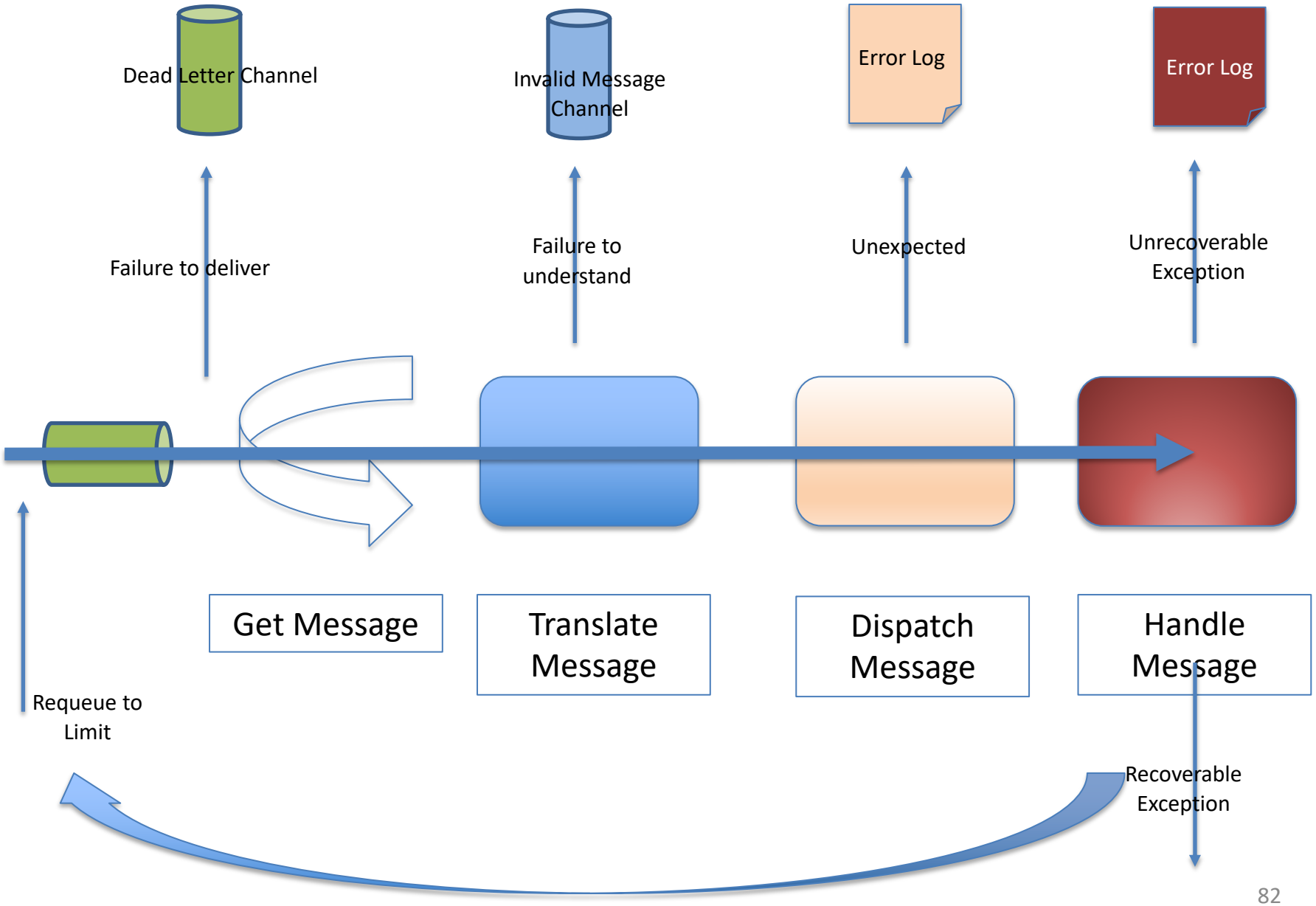

http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html

# Messaging Gateway

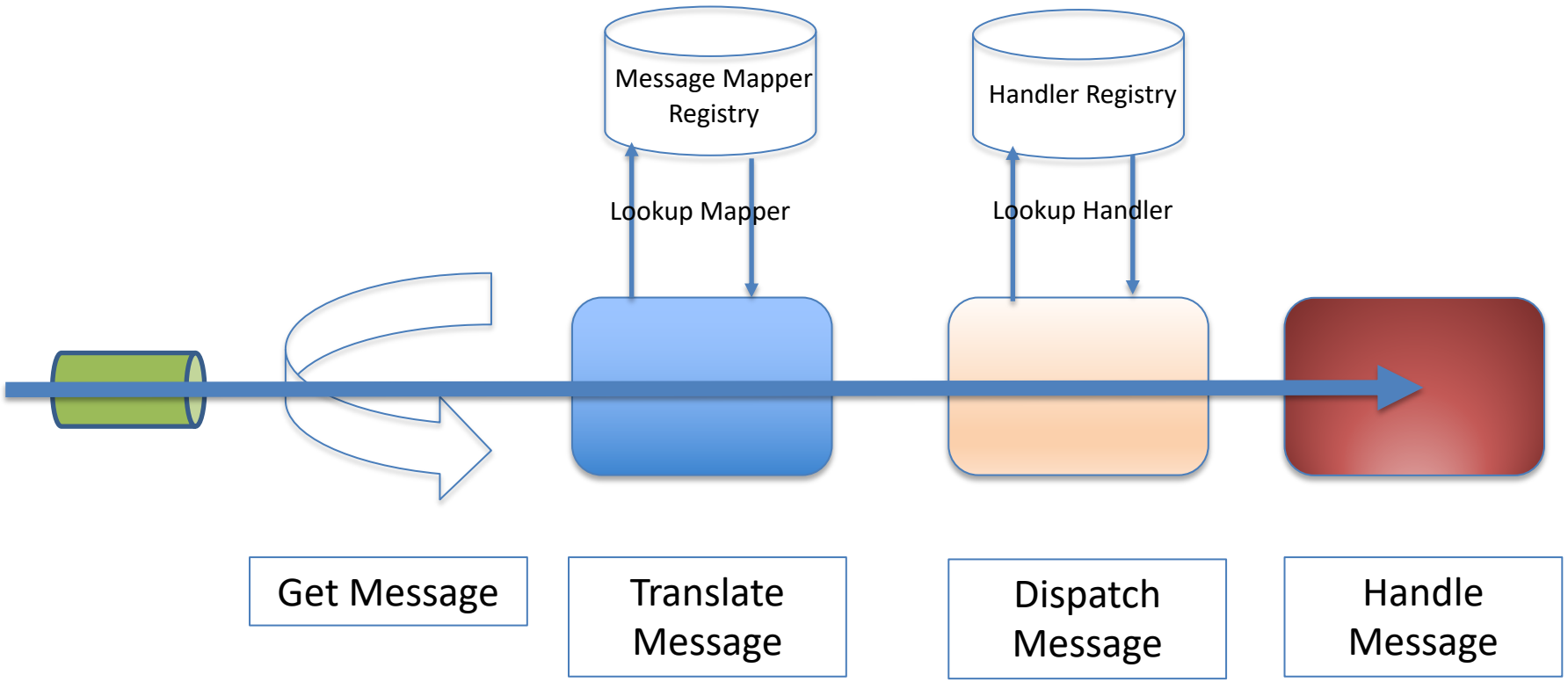



http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingGateway.html

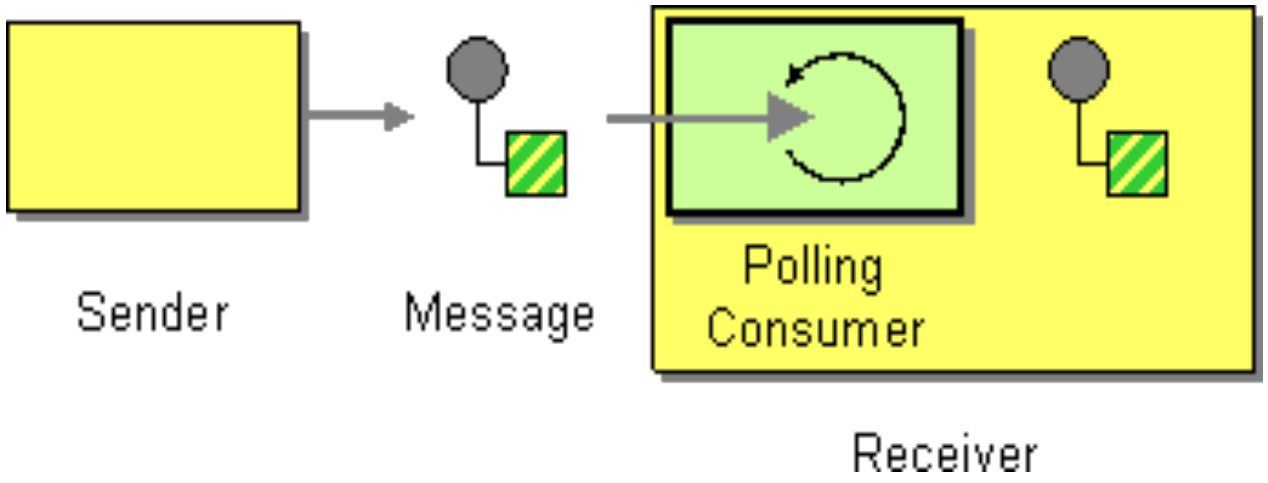
# 5.5 THE MESSAGE PUMP


Dead Letter Channel
Invalid Message Channel
Error Log
Error Log
Failure to deliver
Failure to understand
Unexpected
Unrecoverable Exception
Get Message
Translate Message
Dispatch Message
Handle Message
Requeue to Limit
Recoverable Exception

# Translate and Dispatch

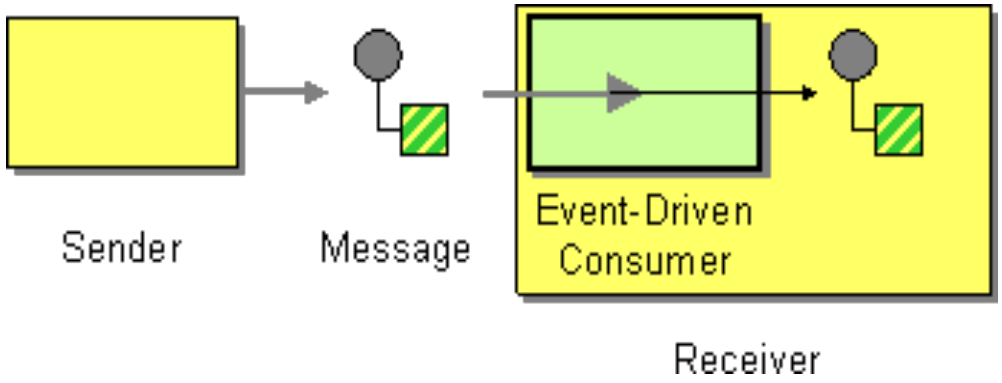


# Polling Consumer



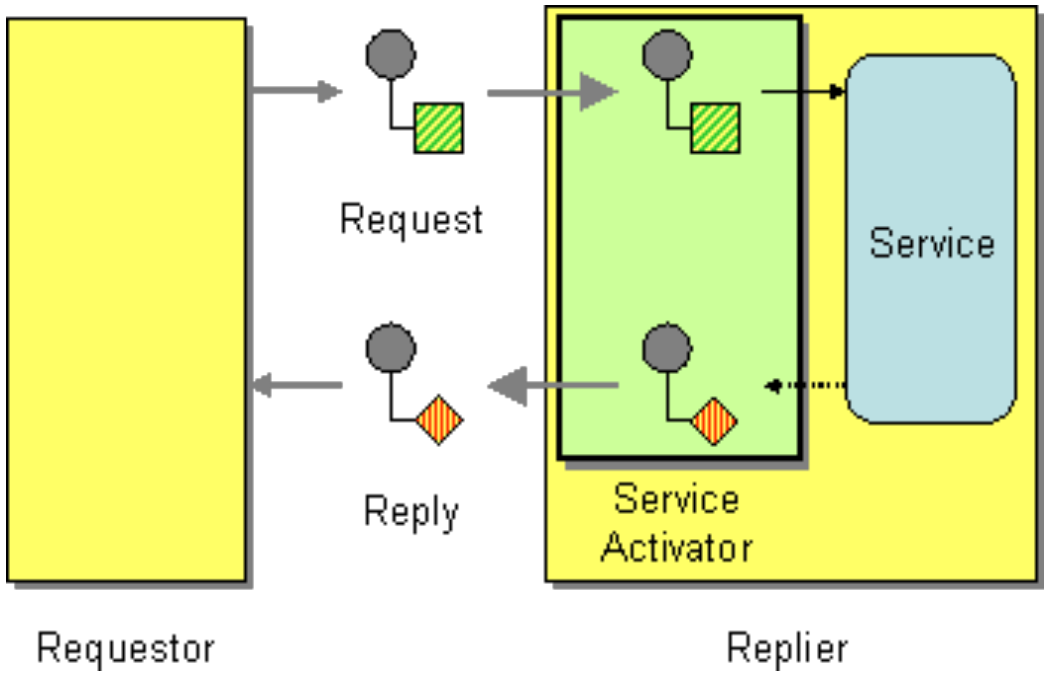

http://www.enterpriseintegrationpatterns.com/patterns/messaging/PollingConsumer.html

# Event Driven Consumer



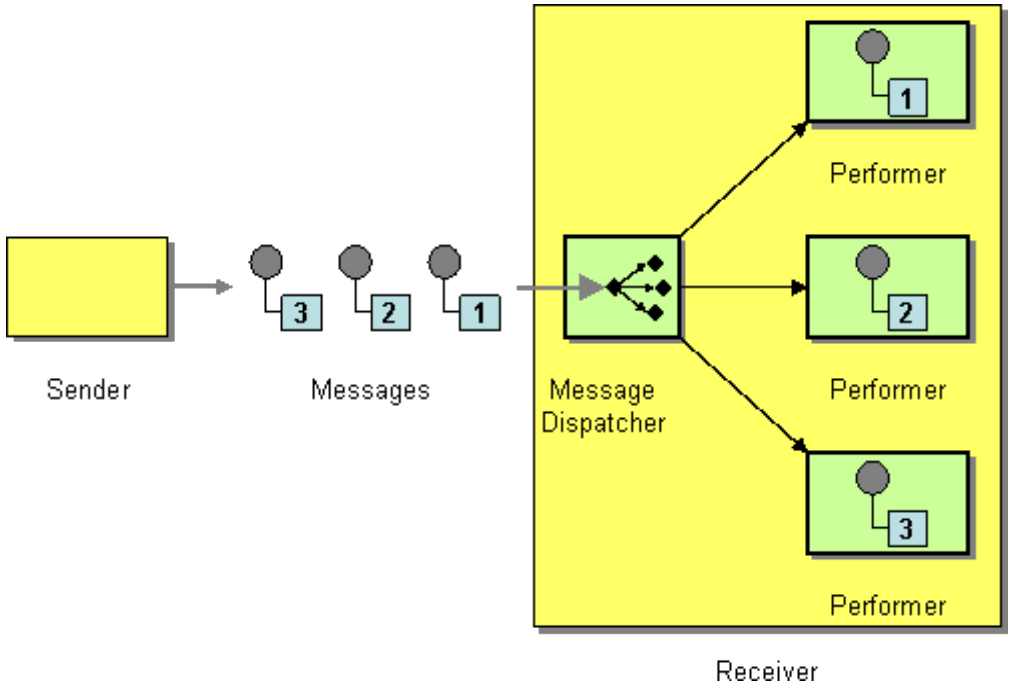

http://www.enterpriseintegrationpatterns.com/patterns/messaging/EventDrivenConsumer.html

# Service Activator




http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingAdapter.html

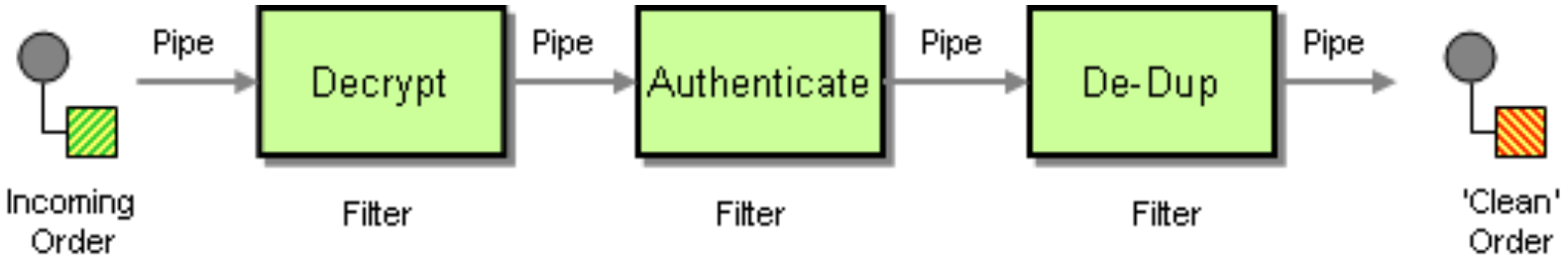
# Competing Consumers




http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageDispatcher.html

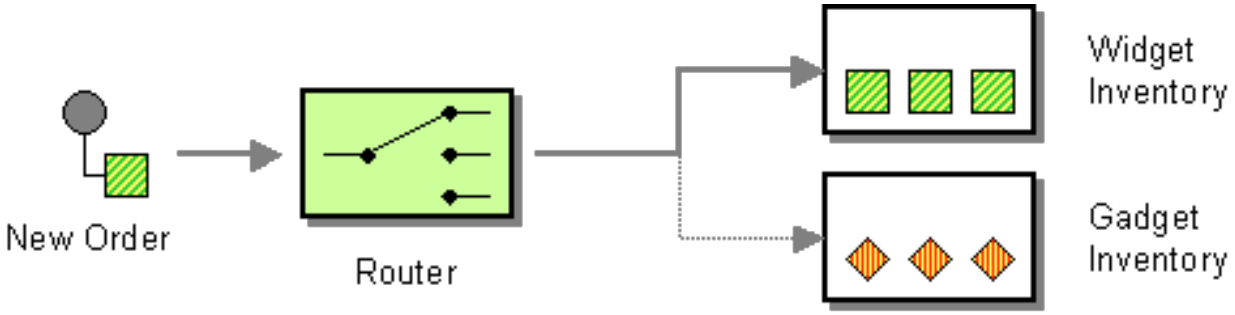
# 5.6 PIPELINES

# Pipes and Filters



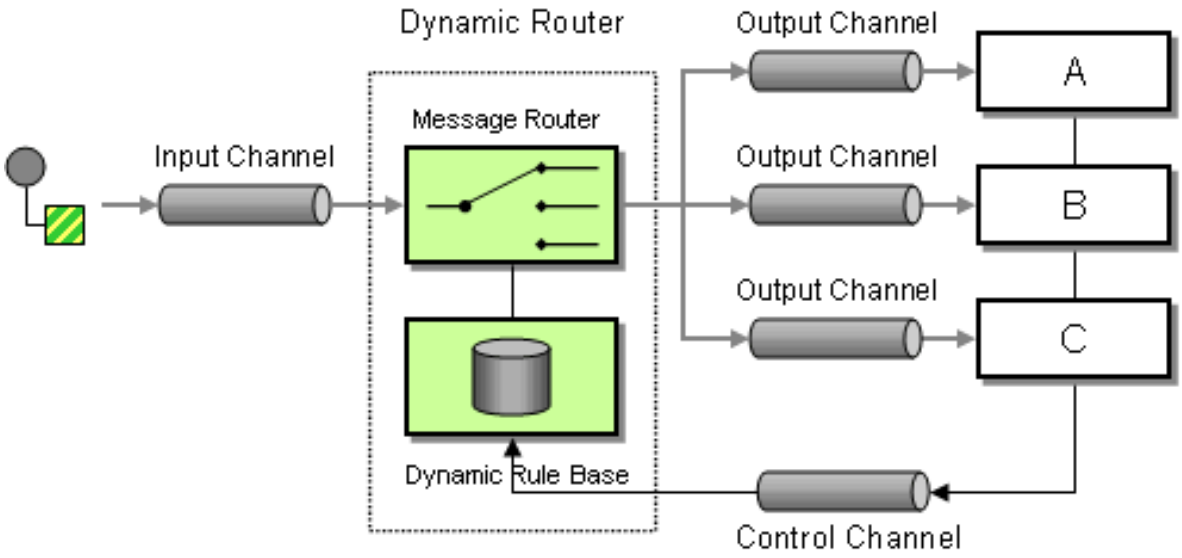

http://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html

# Content Based Router



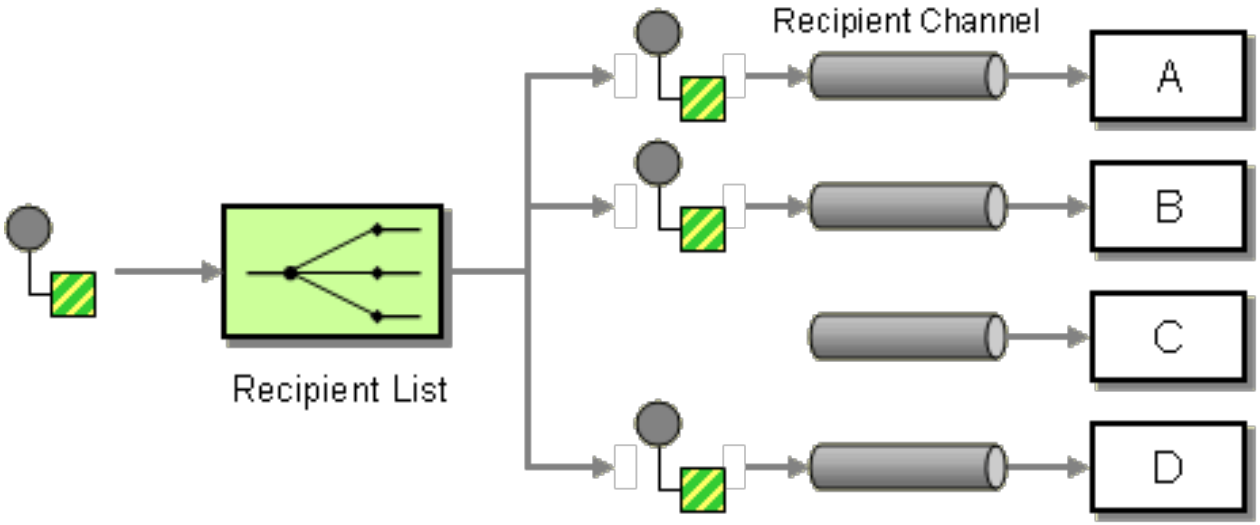

http://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html

# Dynamic Router



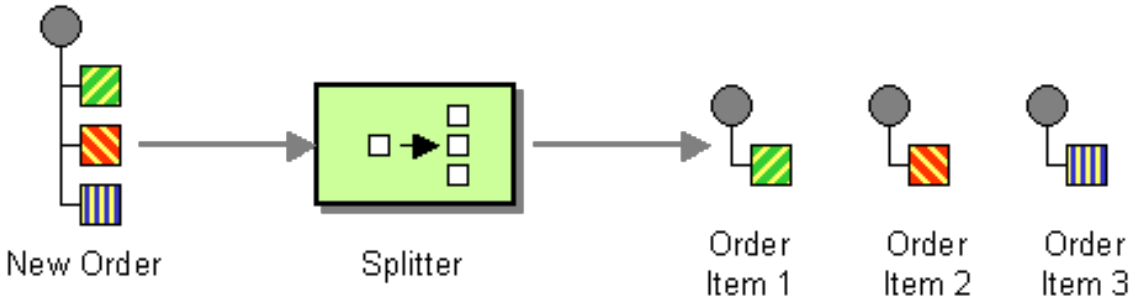

http://www.enterpriseintegrationpatterns.com/patterns/messaging/DynamicRouter.html

# Recipient List



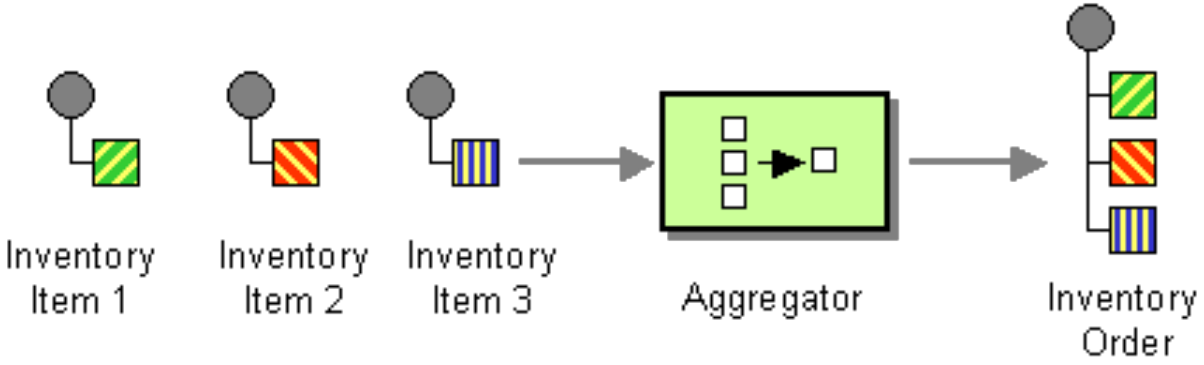

http://www.enterpriseintegrationpatterns.com/patterns/messaging/RecipientList.html

# Splitter



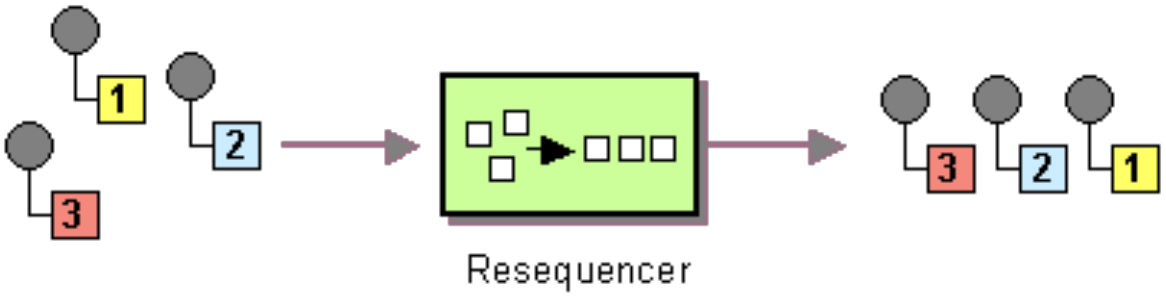

http://www.enterpriseintegrationpatterns.com/patterns/messaging/Sequencer.html

# Aggregator




http://www.enterpriseintegrationpatterns.com/patterns/messaging/Aggregator.html

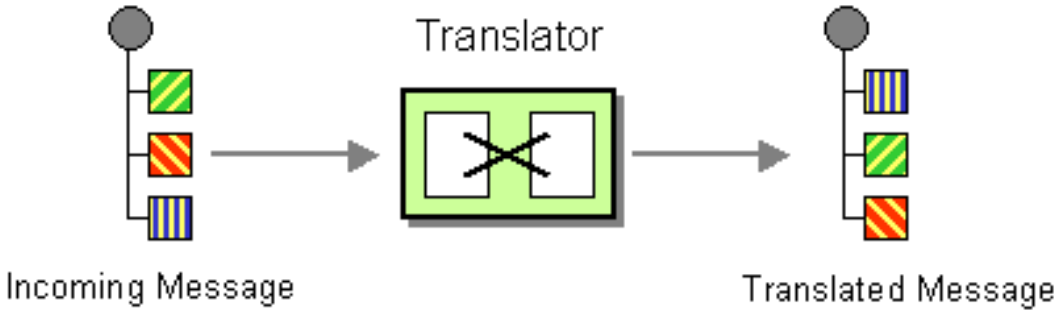
# Resequencer




http://www.enterpriseintegrationpatterns.com/patterns/messaging/Resequencer.html

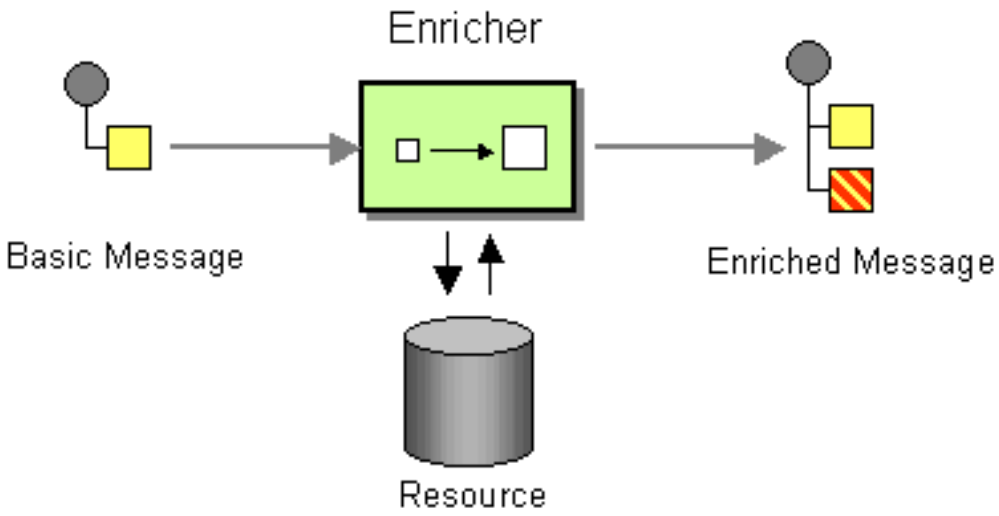
# 5.7 TRANSFORMATION

# Message Translator




http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html

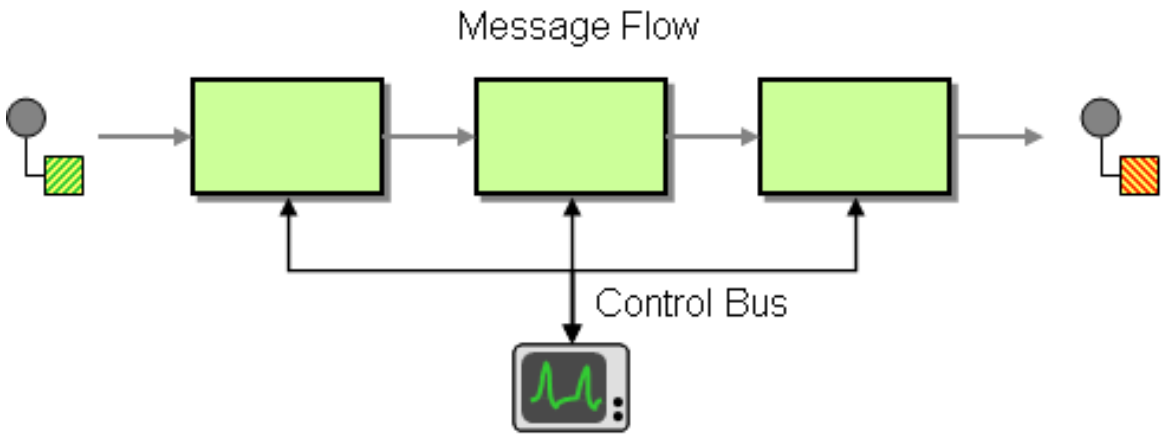
# Content Enricher




http://www.enterpriseintegrationpatterns.com/patterns/messaging/DataEnricher.html

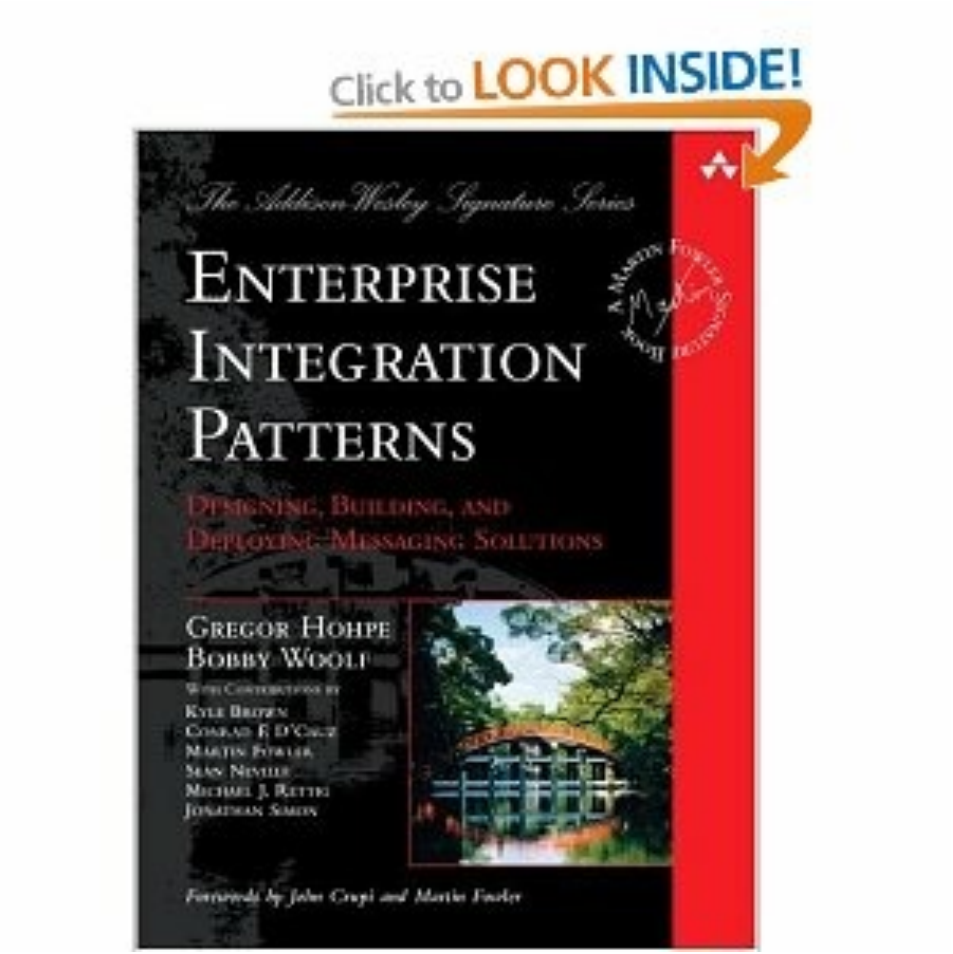
# 5.8 MANAGEMENT

# Control Bus




http://www.enterpriseintegrationpatterns.com/patterns/messaging/ControlBus.html

# Further Reading



# Q&A