Take a Ride on the Bus

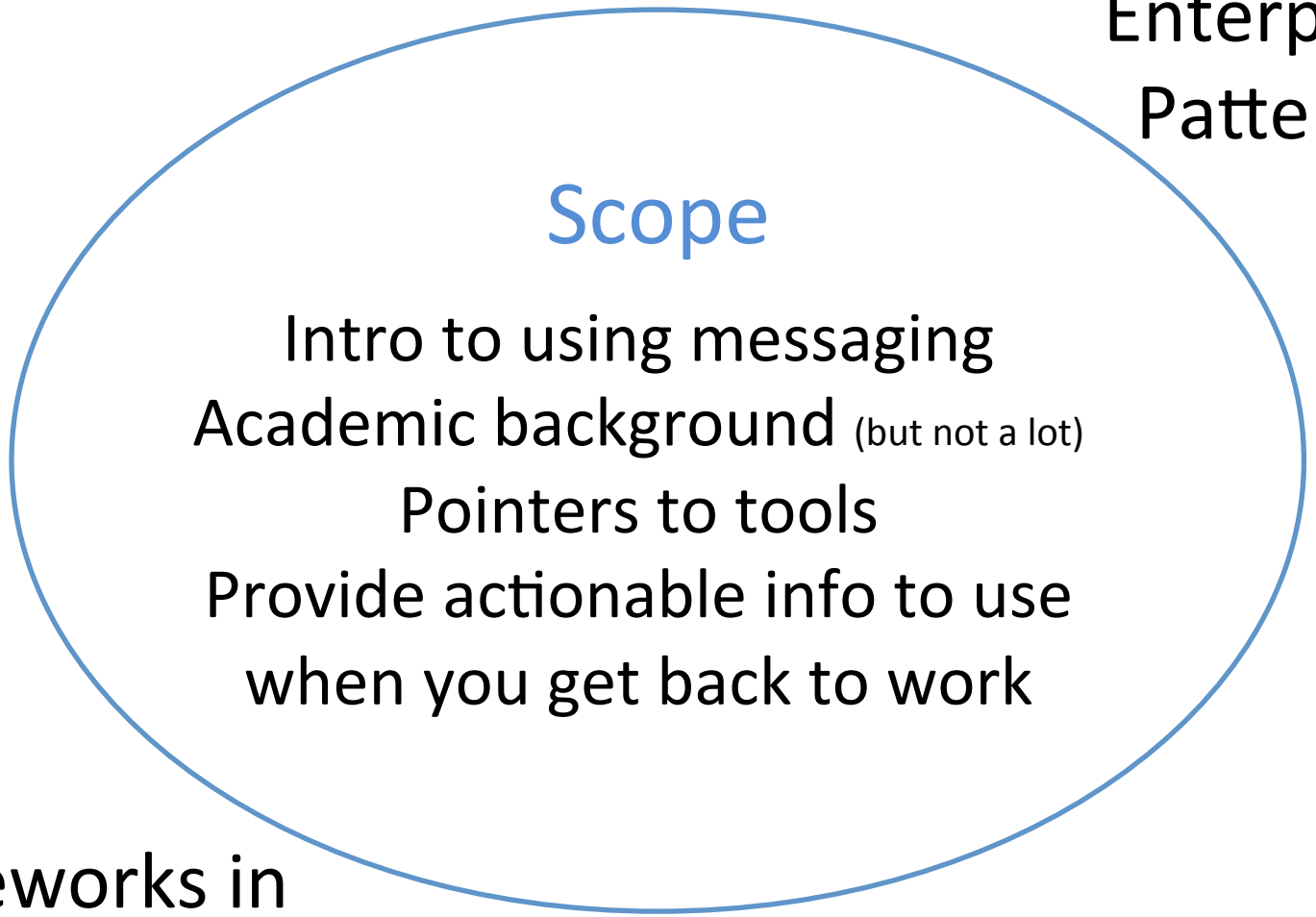
Practical Messaging and Queueing

Chris Meadows
Director of Software Engineering
& Senior Developer
Cloudswell
Email: meadoch1@gmail.com
Twitter: @meadoch1

Agenda

What’s messaging?
Why would I want to use it?
Examples
Q&A

Setting the Stage



Enterprise Patterns

ESB

Frameworks in depth

Fact or Fiction?

Use of messaging can make your applications easier to
build,
understand,
and maintain

What is Messaging?

"Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model, processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize." - from Wikipedia

What is Messaging?
(Practically Speaking)

A means of communication between two distinct programmatic constructs

Could be

- inter-process
- between simple objects
- asynchronous

Sound Familiar?

CORBA

.NET Remoting

Java RMI

DCOM

AMQP

Erlang inter-process calls

Web Services

Ruby method calls

SOAP

Smalltalk method calls

RESTful APIs

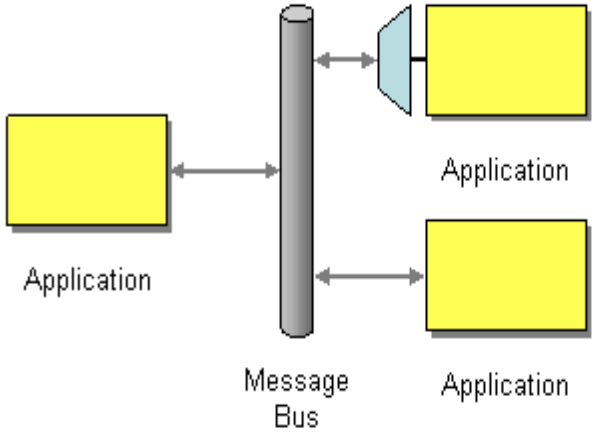
What is a message?

A defined series of bytes which can be encoded and decoded with consistent meaning

What is a message bus?

An enterprise contains several existing systems that must be able to share data and operate in a unified manner in response to a set of common business requests.

What is an architecture that enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?



Structure the connecting middleware between these applications as a *Message Bus* that enables them to work together using messaging.

A *Message Bus* is a combination of a common data model, a common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces. This is analogous to a communications bus in a computer system, which serves as the focal point for communication between the CPU, main memory, and peripherals. Just as in the hardware analogy, there are a number of pieces that come together to form the message bus:

from http://www.eaipatterns.com/MessageBus.html

Warning…

http://commons.wikimedia.org/wiki/File:Amargosa_desert.jpg

What is a message bus?

Characteristics

Anatomy

Patterns

What is a message bus?

Characteristics

Characteristics of a Message Bus

Reliable Transfer

Preservation of Order

Broadcast/Collection

Synchronous/Async

Characteristics of a Message Bus

Reliable Transfer

Do you need guarantees of delivery?

Maybe Yes	Maybe No
Request processing	Status update
System state change	“Tweets”
…	Heartbeats

Characteristics of a Message Bus

Preservation of Order

Do you need messages delivered in the same order as sent?

Maybe Yes	No
Status update	Voting
System state change	“Tweets”
…	Heartbeats

Characteristics of a Message Bus

Broadcast/Collection

How many senders are talking to how many recievers?

One – One

One – Many

Many – One

Many - Many

Characteristics of a Message Bus

Synchronous / Asynchronous

Do you need serial processing?

Synchronous	Asynchronous
RPC	Pub/Sub
Method invocation in OOP	Twitter
…	Facebook
	Parallel Processing

What is a message bus?

Anatomy

Anatomy of a Message Bus

Message Storage

Serialization Protocols

Transport Mechanism

Anatomy of a Message Bus

Message Storage

Frequently involves a Queue

Allows delayed processing

Can preserve order

Can offer fault tolerance

Anatomy of a Message Bus

Message Storage

Database Backed

File Backed

In Memory

Anatomy of a Message Bus
Message Storage

Database Backed

Pros
- Simple to set up
- Tools are usually at hand
- Easy view into your queues

Cons
- Prone to contention at high volumes
- Dependency on additional tool

File Backed

In Memory

Anatomy of a Message Bus
Message Storage

Database Backed

File Backed

Pros
- Simple to set up
- Easy view into your queues

Cons
- Prone to contention at high volumes
- Management of files can be difficult

In Memory

Anatomy of a Message Bus
Message Storage

Database Backed

File Backed

In Memory

Pros

- Very fast
- Can use caches to eliminate custom handling of memory
- Can make distributed

Cons

- Can demand a lot of memory
- Danger of running out of memory

Anatomy of a Message Bus

Serialization Protocol

Interoperability

Size

Speed of encoding

Ease of use

There are many good standards already

Protocol Buffers, Thrift, MessagePack,…

Anatomy of a Message Bus

Transport Mechanism

What connects sender and reciever?

TCP/IP

UDP

HTTP

…

Might be dictated by other decisions

(db backed -> db client)

Anatomy of a Message Bus

Build

- Pros
  - Flexibility
  - Deep knowledge
  - Fun
- Cons
  - A lot of work
  - Tricky to get right in all cases

“Buy”

- Pros
  - Many good choices
  - More “testers”
  - Support for getting it right
- Cons
  - Can be very expensive
  - Might impose trade-offs

Anatomy of a Message Bus

Many good Open Source Systems Exist

RabbitMQ
ActiveMQ
ZeroMQ
Resque
Delayed Job
Beanstalkd
JMS
Amazon SQS
NServiceBus
Mass Transit
…

What is a message bus?

Patterns

Patterns of Usage of a Message Bus

Exclusive Pair

Request – Reply

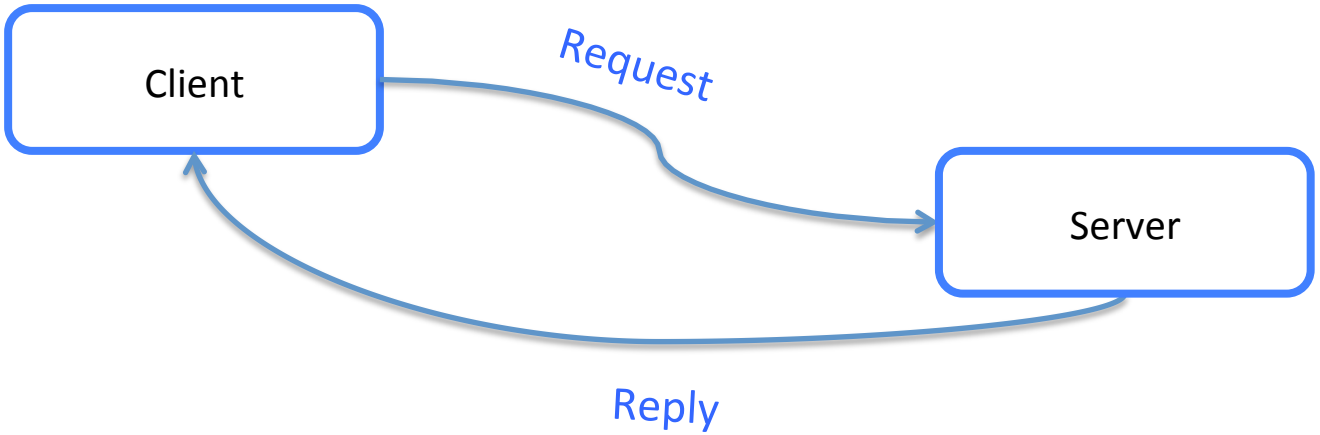
Competing Consumer

Pub / Sub

Fan-in

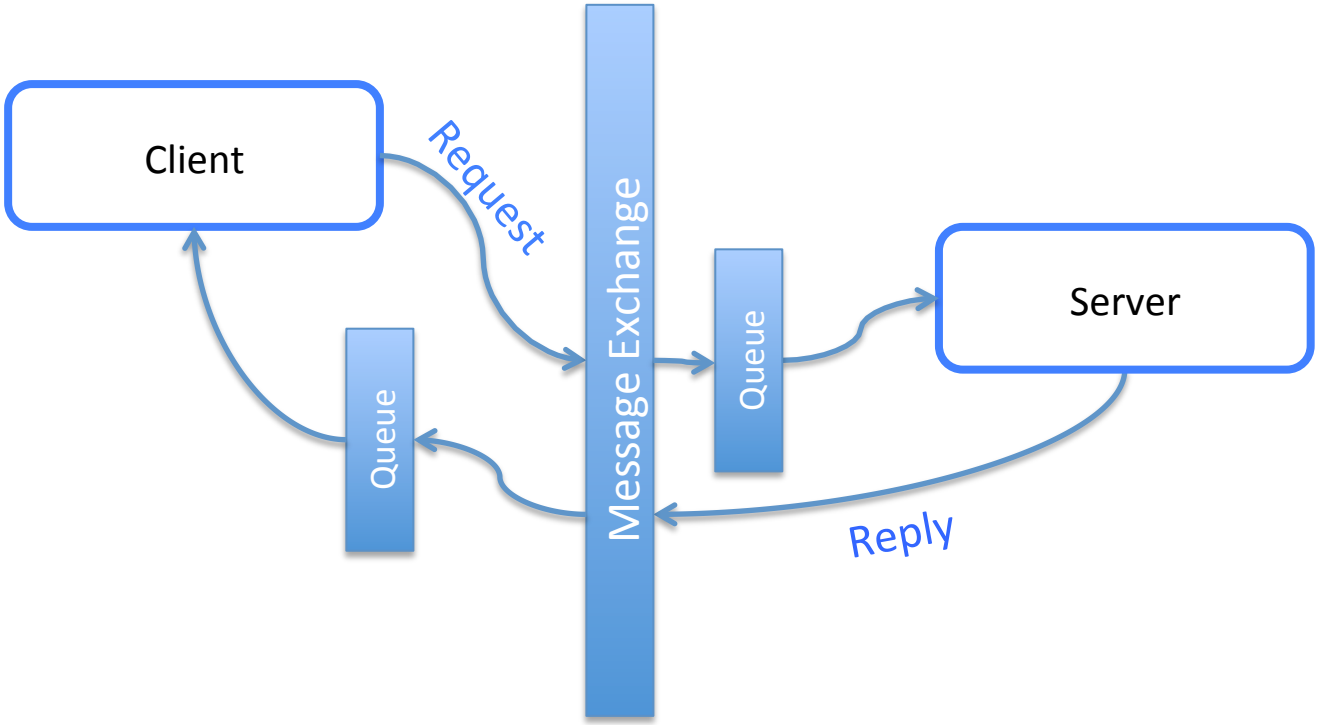
Patterns of Usage of a Message Bus

Direct Pair



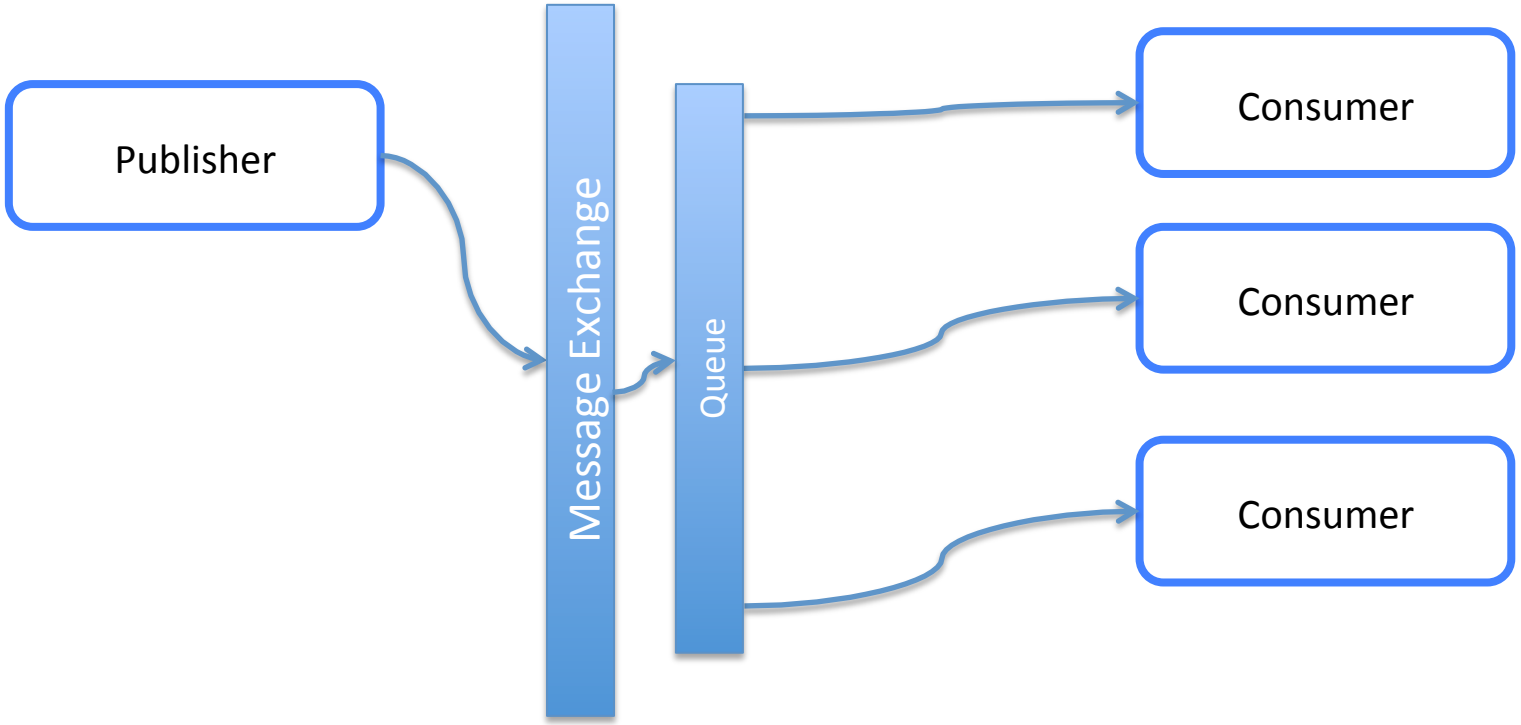
Patterns of Usage of a Message Bus

Request - Reply



Patterns of Usage of a Message Bus

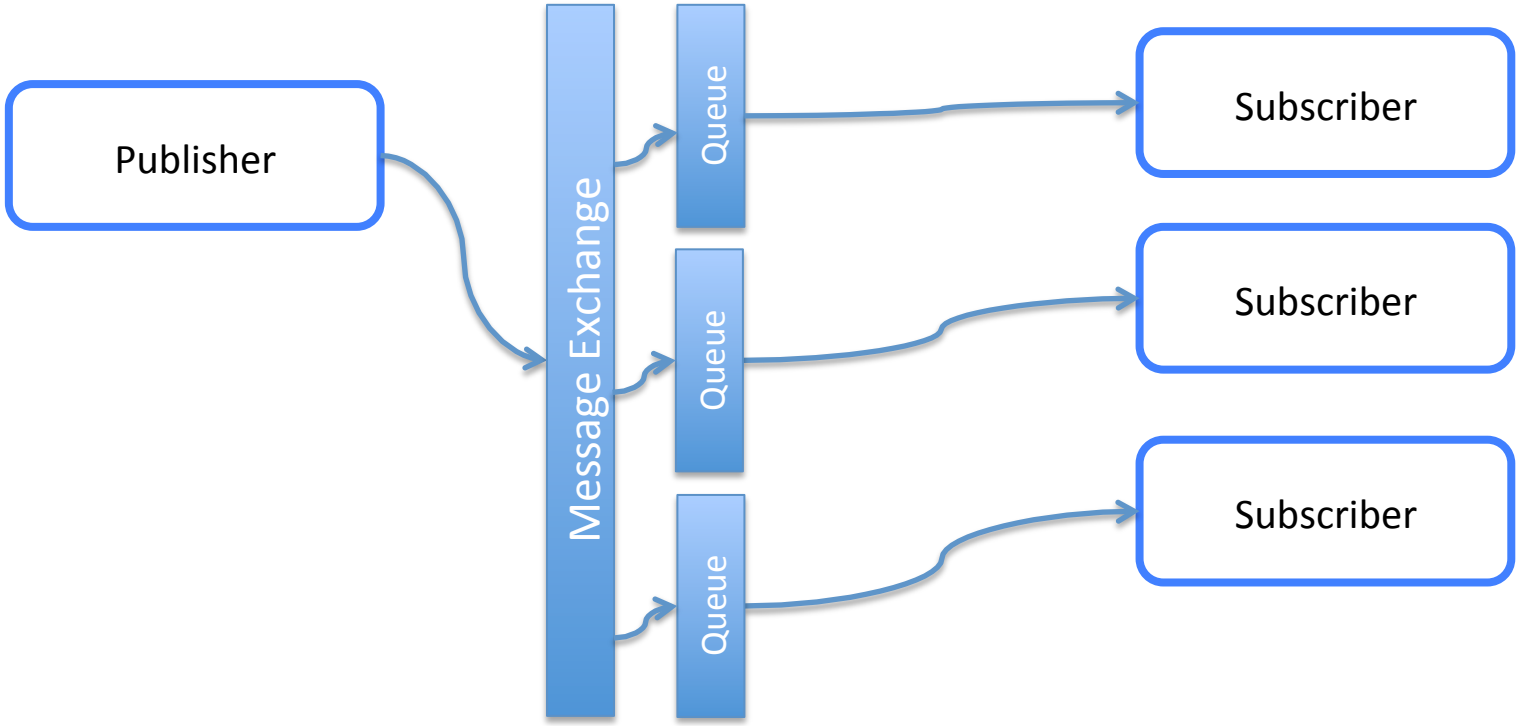
Competing Consumer



Only one consumer receives a given message
"first come, first served"

Patterns of Usage of a Message Bus

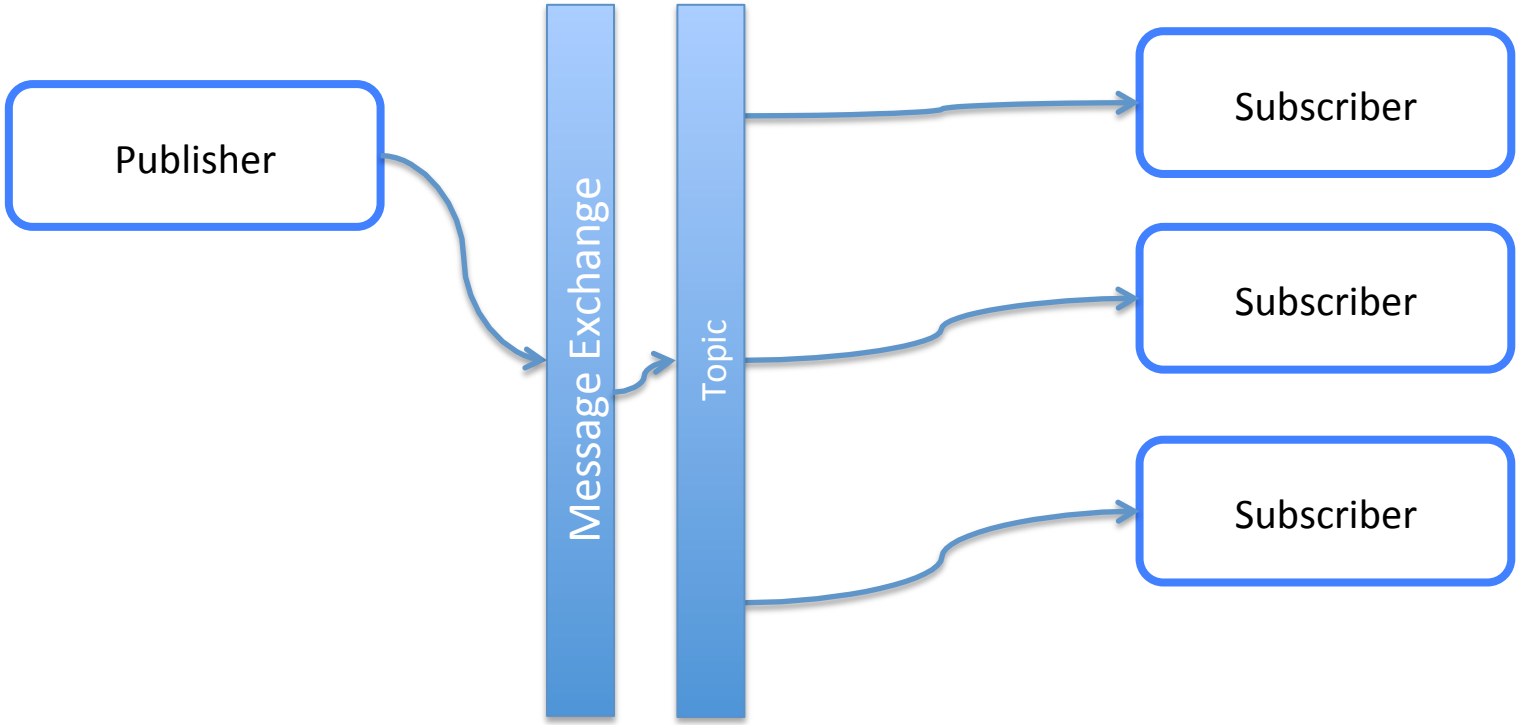
Pub / Sub



One message is copied to many subscribers/consumers

Patterns of Usage of a Message Bus

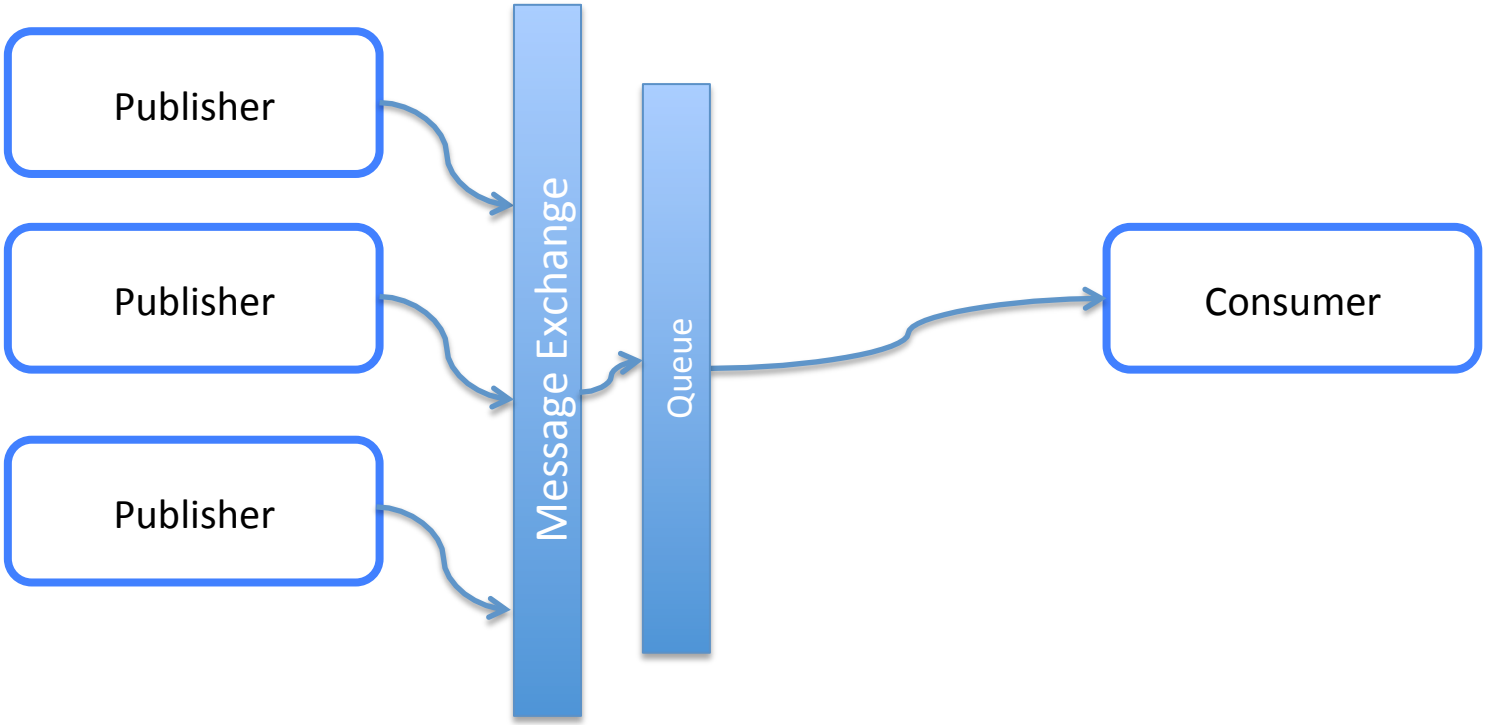
Pub / Sub



One message is copied to many subscribers/consumers

Patterns of Usage of a Message Bus

Fan-in



Why Messaging?

Why Messaging?

Offload Work

Distribute Work

SOA

Easy Expansion

Easier to Conceptualize than Threading

Why Messaging?
"Normal" Alternatives

Threading vs. Messaging

- Pros
  - Visualization is easier
  - No shared state
  - Distribution
  - Resiliency
- Cons
  - New tools to learn
  - Not native to language (most of the time)

Why Messaging?

"Normal" Alternatives

RPC vs. Messaging

- Pros
  - Loose coupling
  - Easier distribution
  - Resiliency
  - Can have lower overhead to set up (WCF)
- Cons
  - New tools to learn
  - Not native to language (most of the time)

Why Messaging?
"Normal" Alternatives

Load Balancers vs. Messaging

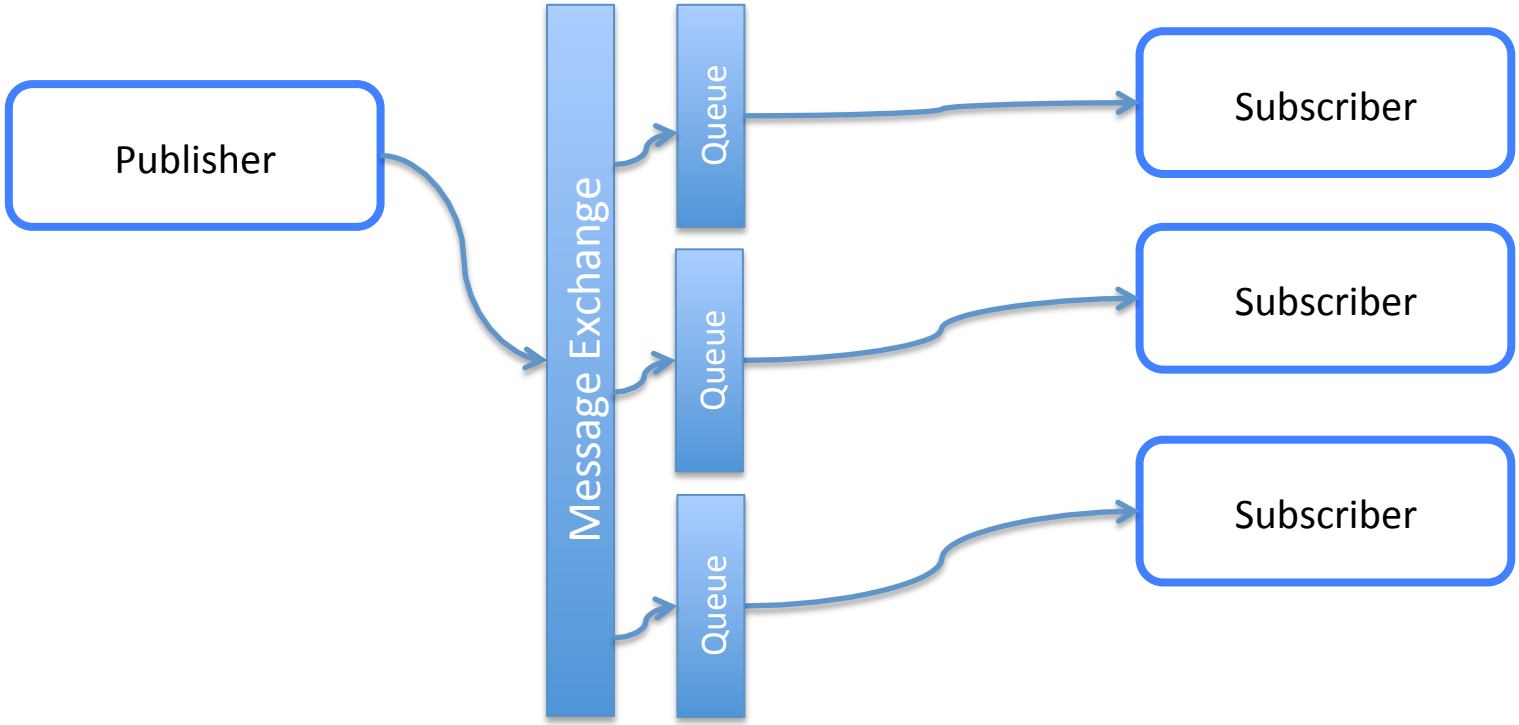
- Pros
  - Doesn't require expensive LB hardware
  - LB & failover becomes an application concern vs. networking
  - Control and tuning
- Cons
  - Requires planning
  - Can't just "stand it up"
  - May require server(s) to do the work

You made it!

http://commons.wikimedia.org/wiki/File:Hanauma_bay_(3).jpg

Ready for some examples?

Simple Pub / Sub



Simple Pub/Sub

Publisher

```ruby
require 'stomp'

cli = Stomp::Client.new("admin","password","localhost",61613)
5.times do |i|
    puts "Publishing message #{i}"
    cli.publish("/topic/pubsub1","Pub/sub msg #{i}")
end
```

Subscriber

```ruby
require 'stomp'

num = ARGV[0]
cli = Stomp::Client.new("admin","password","localhost",61613)
puts "Listening for messages"
cli.subscribe("/topic/pubsub1", {"id" => "t#{num}"}) { |m|
    puts "Listener #{num} Recieved a message: #{m.body}"
}

while true
    sleep 5
end
```

Files: example1_publisher.rb & example1_subscriber.rb

Event Broadcast

Pub/Sub works great

Good for

- Logging
- Notifying system components of events as they occur
- Monitoring
- System diagnostics

Event Broadcast

Publisher

```ruby
require 'stomp'

locs = ["NewYork", "Washington", "Orlando", "Phoenix", "Austin", "Chattanooga" ]
infos = ["Weather", "Traffic", "Forecast"]

cli = Stomp::Client.new("admin","password","localhost",61613)

#first clear the client screens
cli.publish("/topic/reset","Clear for new run")

5.times do |i|
    msg = "#{locs.sample}.#{infos.sample}"
    puts "Publishing message #{i} to #{msg}"
    cli.publish("/topic/#{msg}","Pub/sub msg #{i}")
end
```

Listener

```ruby
require 'stomp'

mask = ARGV[0]
system("clear")

cli = Stomp::Client.new("admin","password","localhost",61613)

puts "Listening for messages related to #{mask}"
cli.subscribe("/topic/#{mask}", {"id" => "Listening"}) { |m|
    puts "Listener #{mask} Recieved a message - Topic: #{m.headers['destination']} Body: #{m.body}"
    cli.publish("/queue/resp","Done working on #{m.body}")
}

cli.subscribe("/topic/reset", {"id" => "reset"}) { |m|
    system("clear")
    puts "Listener #{mask} Recieved a clear message: #{m.body}"
}
while true
    sleep 5
end
```

Files: example3_publisher.rb, example3_subscriber.rb, example3_subscriber.erl

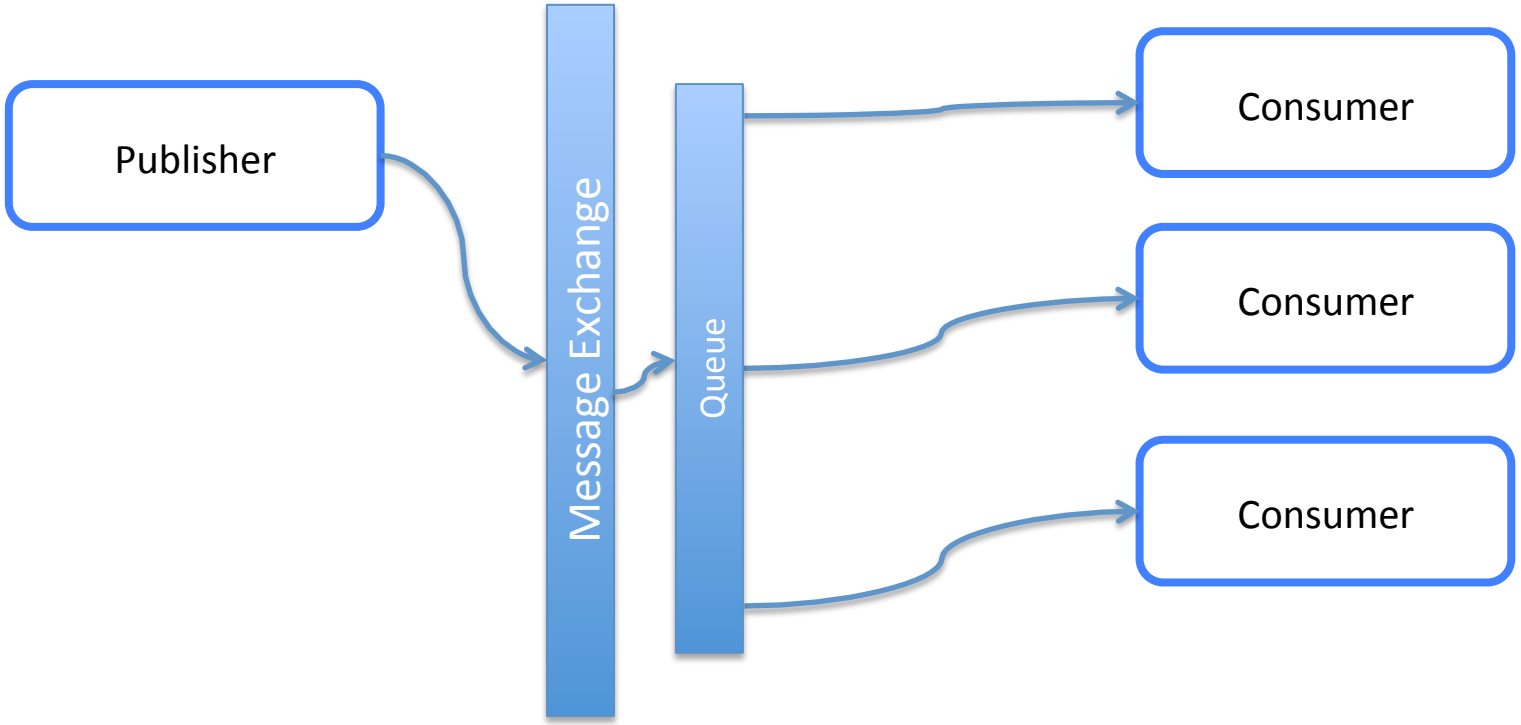
Work Distribution

Competing consumer works great

Good for

- Email sending
- Long running job tasking

Work Distribution



Only one consumer receives a given message
"first come, first served"

Work Distribution

Publisher

```ruby
require 'stomp'

cli = Stomp::Client.new("admin","password","localhost",61613)
5.times do |i|
    puts "Publishing message #{i}"
    cli.publish("/queue/cc1","Do work! msg #{i}")
end

```

Worker

```ruby
require 'stomp'

num = ARGV[0]
cli = Stomp::Client.new("admin","password","localhost",61613)
system("clear")
puts "Listening for messages"
cli.subscribe("/queue/cc1", {"id" => "t#{num}"}) { |m|
    puts "Worker #{num} Recieved a message: #{m.body}"
}

while true
    sleep 5
end
```

Files: example2_publisher.rb & example2_consumer.rb

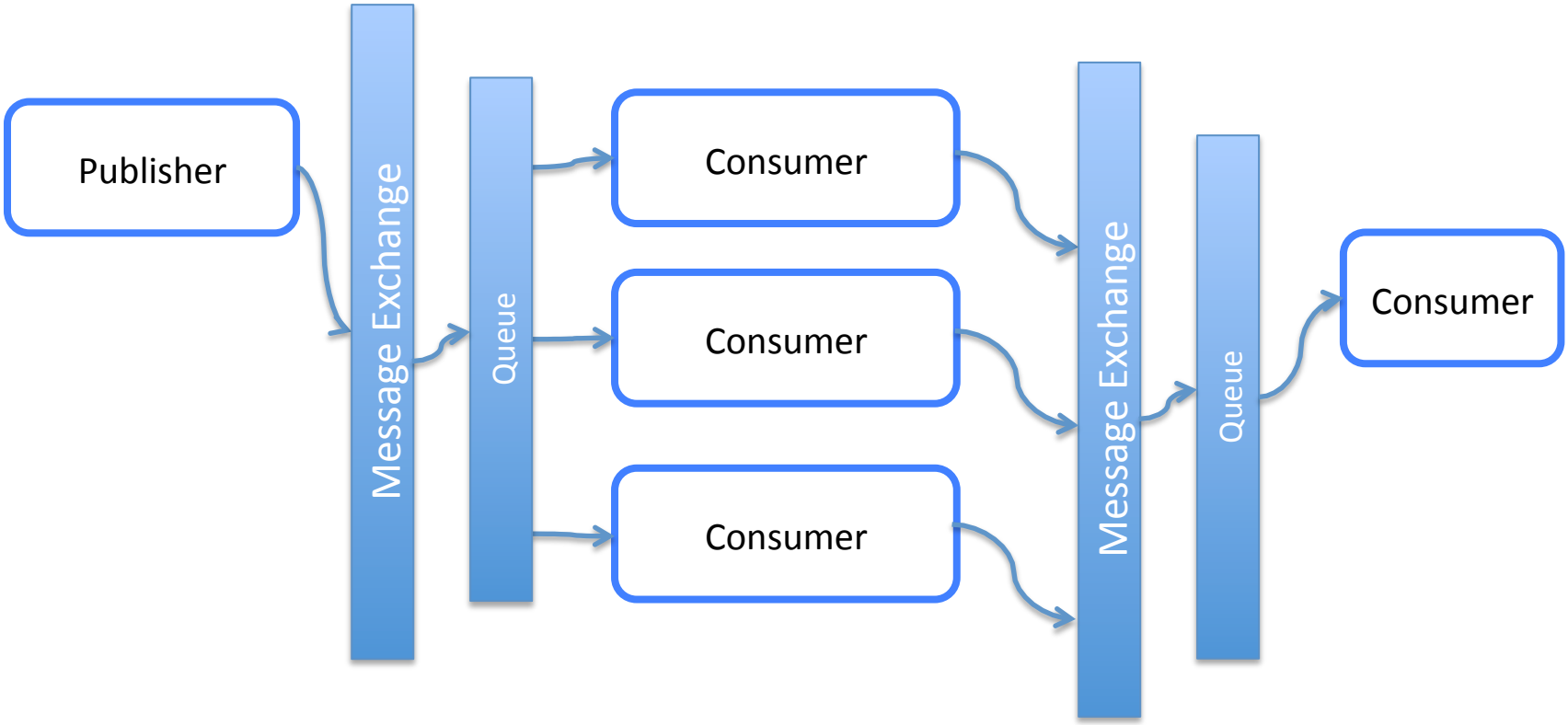
Work Distribution & Consolidation

Adds one more step of complexity

Good for

- Distributed workflows
- Asynchronous workflows

Work Distribution



Work Distribution

Publisher

```ruby
require 'stomp'

cli = Stomp::Client.new("admin","password","localhost",61613)

#first clear the client screens
cli.publish("/topic/reset","Clear for new run")

5.times do |i|
    puts "Publishing message #{i} to do work"
    cli.publish("/queue/workdist","Do work msg #{i}")
end
```

Final Worker

```ruby
require 'stomp'

cli = Stomp::Client.new("admin","password","localhost",61613)

cli.subscribe("/queue/resp", {"id" => "resp"}) { |m|
    puts "Recieved finished task #{m.body}"
}

while true
    sleep 5
end
```

Worker

```ruby
require 'stomp'

system("clear")

cli = Stomp::Client.new("admin","password","localhost",61613)

puts "Listening for messages to do work"
cli.subscribe("/queue/workdist", {"id" => "Listening"}) { |m|
    load = [0,1,2,3].sample

    puts "Recieved a message - Body: #{m.body}"
    puts "Working for #{load} secs"
    sleep load
    cli.publish("/queue/resp","Done working on #{m.body}")
}

cli.subscribe("/topic/reset", {"id" => "reset"}) { |m|
    system("clear")
    puts "Recieved a clear message: #{m.body}"
}
while true
    sleep 5
end
```

Files: example4_publisher.rb, example4_worker & example4_final.rb

Questions?

More info

We just scratched the surface

Google

I'd be glad to talk

Chris Meadows
meadoch1@gmail.com
@meadoch1 (Twitter)

https://github.com/meadoch1/PracticalMessaging