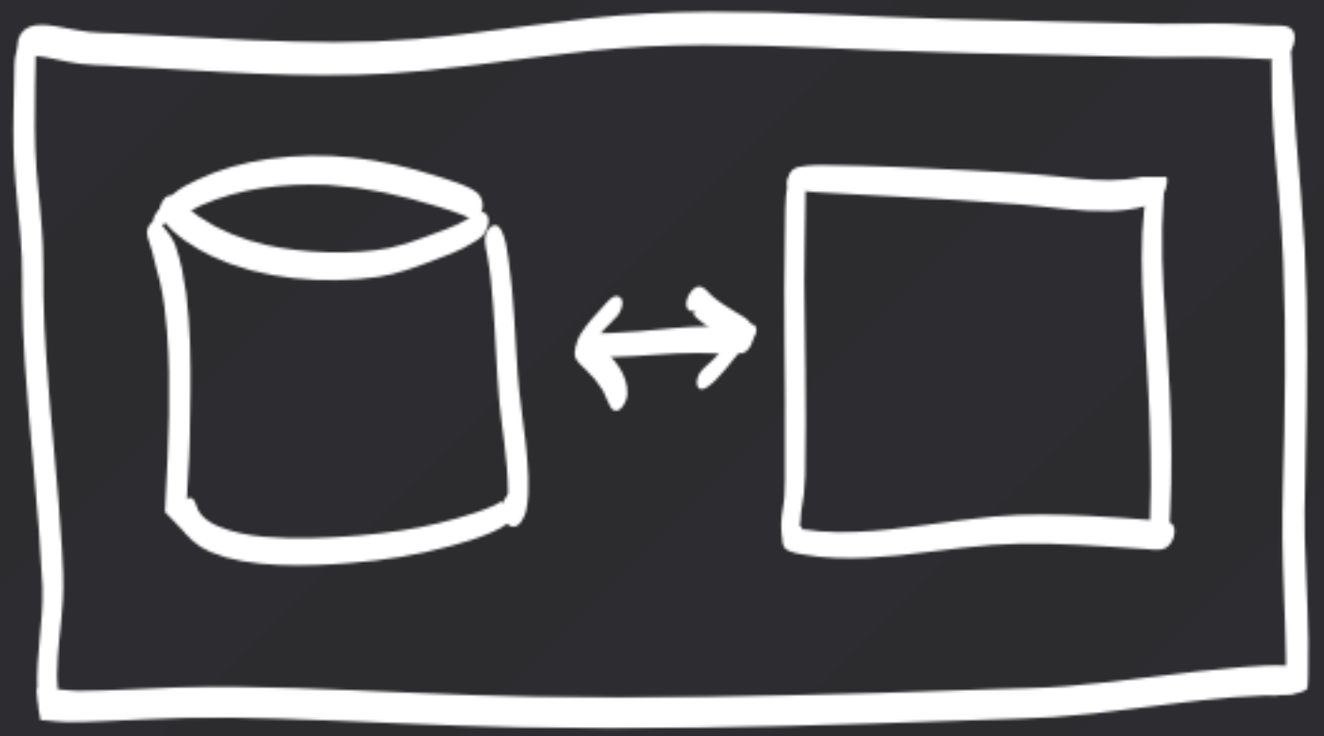
DISTRIBUTED SYSTEMS
with ZeroMQ and gevent

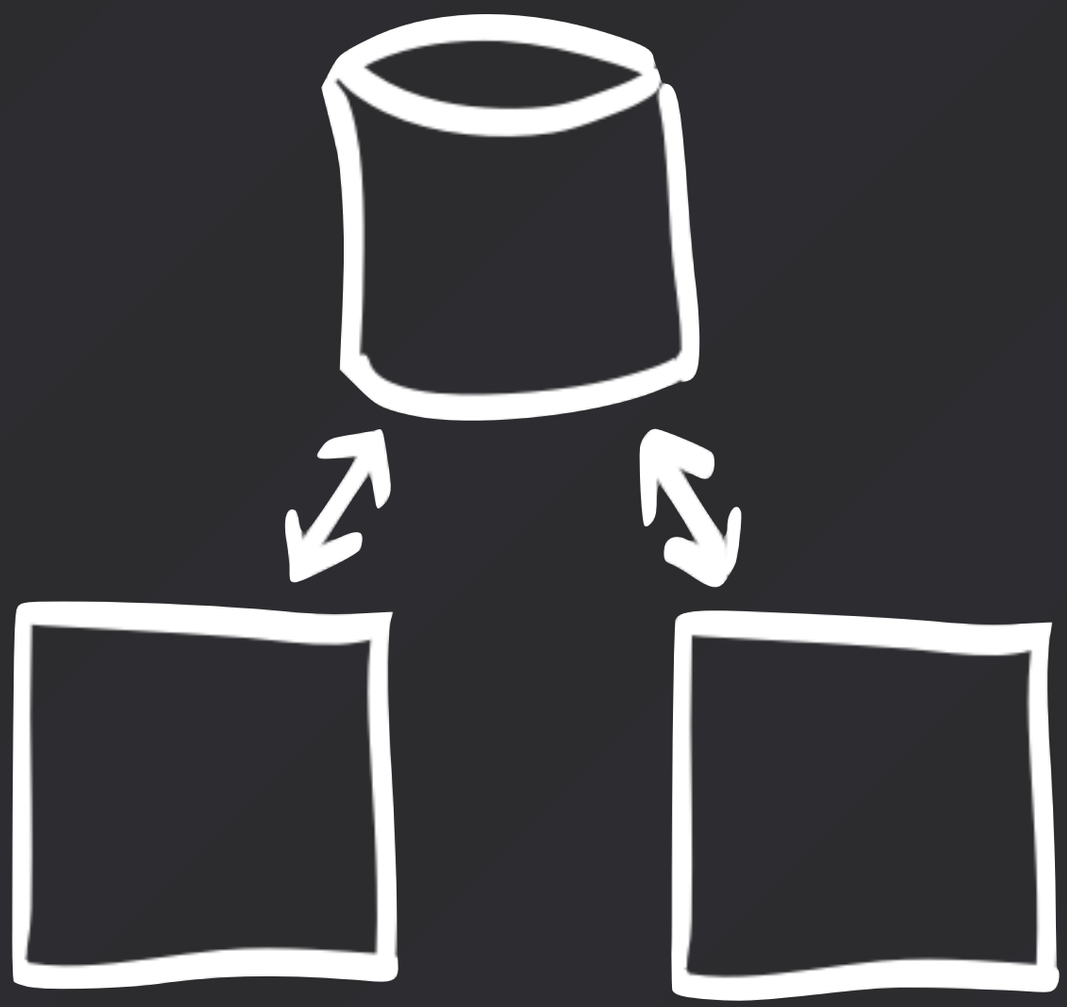
Jeff Lindsay
@progrium

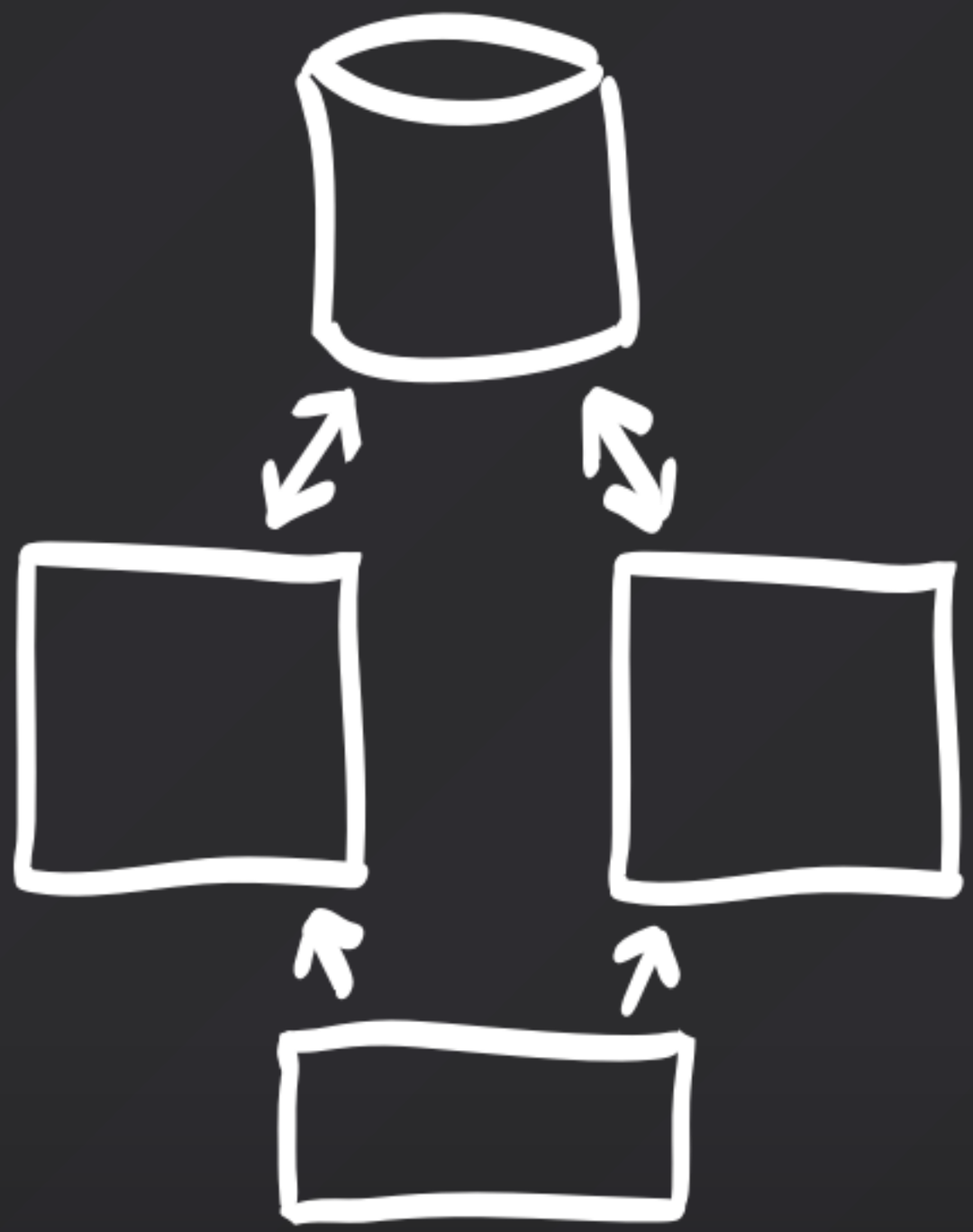
Why distributed systems?

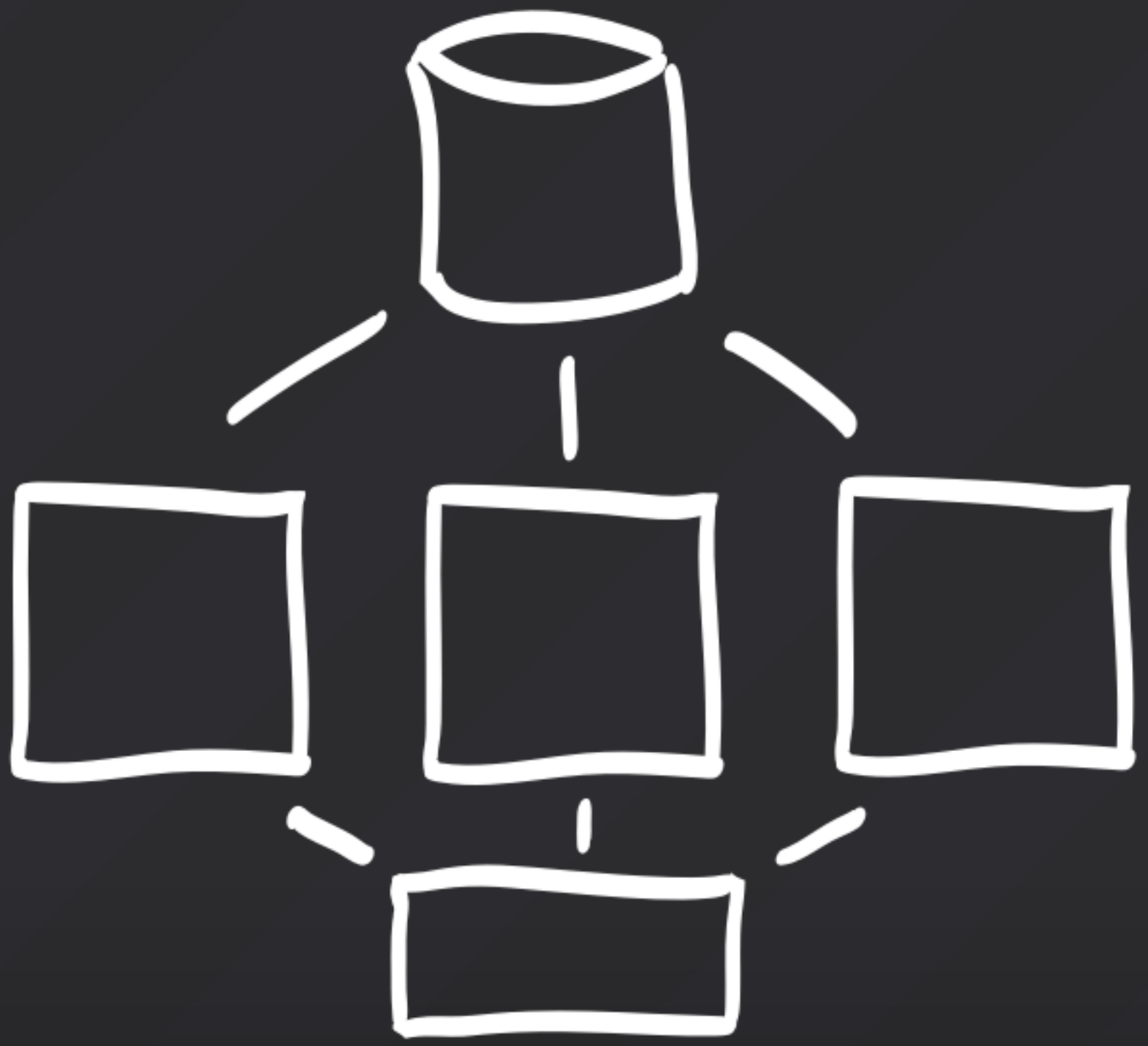
- Harness more CPUs and resources
- Run faster in parallel
- Tolerance of individual failures
- Better separation of concerns

Most web apps evolve into distributed systems

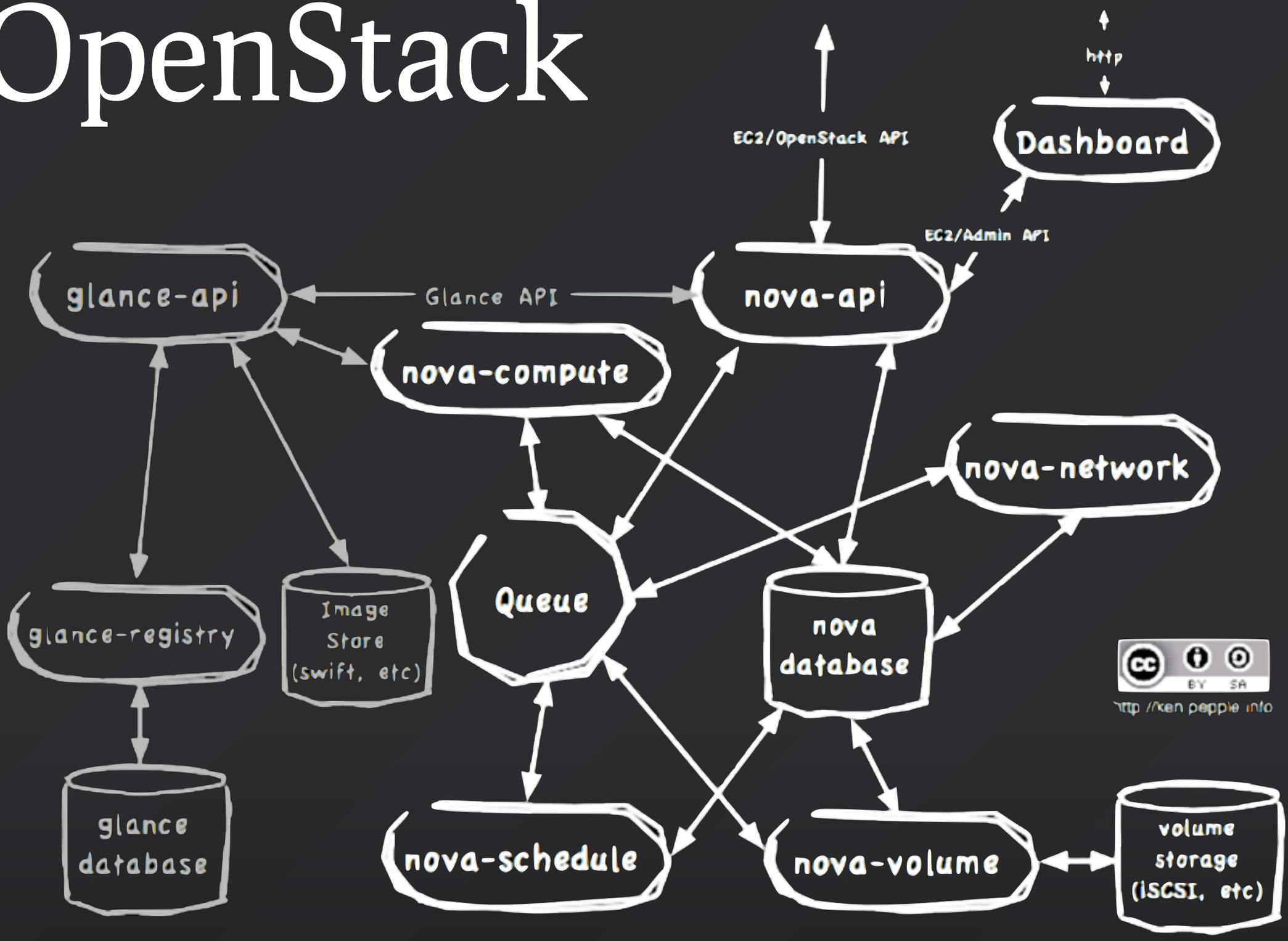








OpenStack


http
Dashboard
EC2/OpenStack API
EC2/Admin API
glance-api
Glance API
nova-api
nova-compute
nova-network
Queue
nova database
glance-registry
Image Store (swift, etc)
glance database
nova-schedule
nova-volume
volume storage (iSCSI, etc)
BY SA


Amazon AWS
Web
API
Client
TwiML
Provider
Provider
Provider

ZeroMQ + gevent

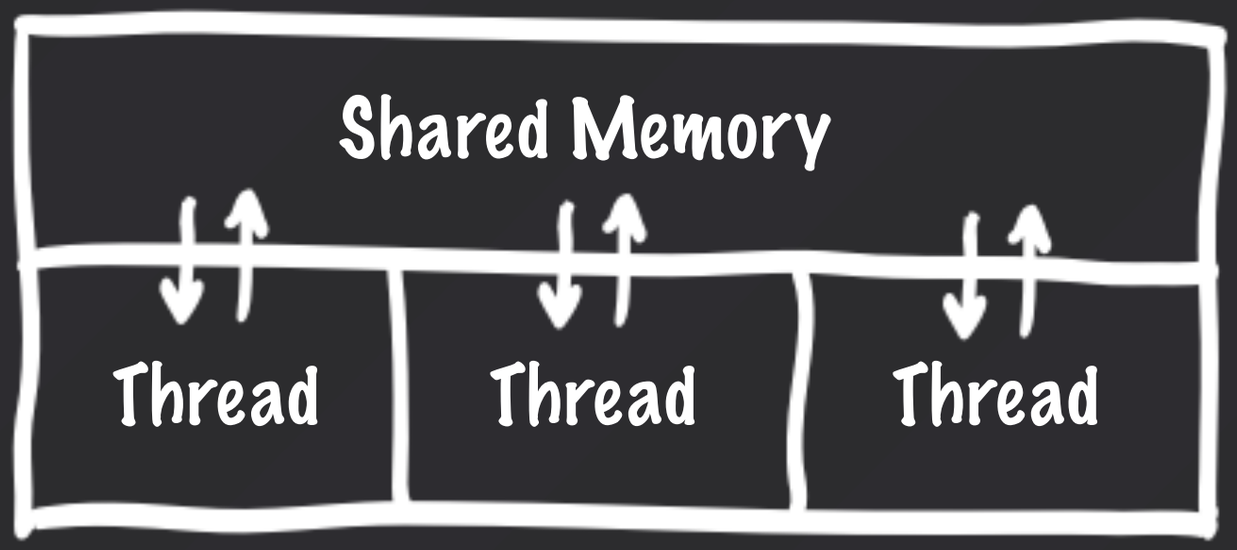
Two powerful and misunderstood tools

CONCURRENCY

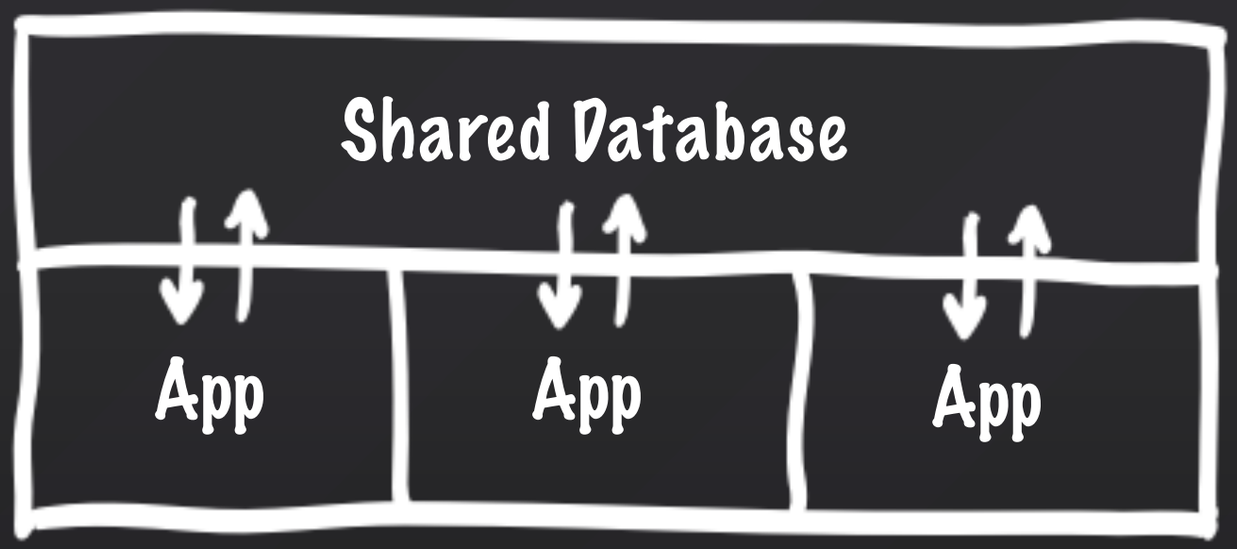
Heart of Distributed Systems

Distributed computing is just another flavor of local concurrency

Multithreading



Distributed system



Concurrency models

- Execution model
  - Defines the “computational unit”
- Communication model
  - Means of sharing and coordination

Concurrency models

- Traditional multithreading
  - OS threads
  - Shared memory, locks, etc
- Async or Evented I/O
  - I/O loop + callback chains
  - Shared memory, futures
- Actor model
  - Shared nothing “processes”
  - Built-in messaging

Examples

- Erlang
  - Actor model
- Scala
  - Actor model
- Go
  - Channels, Goroutines
- Everything else (Ruby, Python, PHP, Perl, C/C++, Java)
  - Threading
  - Evented

Erlang is special.

Normally, the networking of distributed systems is tacked on to the local concurrency model.

MQ, RPC, REST, ...

Why not always use Erlang?

Why not always use Erlang?

- Half reasons
  - Weird/ugly language
  - Limited library ecosystem
  - VM requires operational expertise
  - Functional programming isn't mainstream

Why not always use Erlang?

- Half reasons
  - Weird/ugly language
  - Limited library ecosystem
  - VM requires operational expertise
  - Functional programming isn't mainstream
- Biggest reason
  - It's not always the right tool for the job


Amazon AWS
Web
API
Client
TwiML
Provider
Provider
Provider

Service Oriented Architecture

Multiple languages
Heterogeneous cluster

RPC

RPC

- Client / server

RPC

- Client / server
- Mapping to functions

RPC

- Client / server
- Mapping to functions
- Message serialization

RPC

- Client / server
- Mapping to functions
- Message serialization

Poor abstraction of what you really want

What you want are tools to help you get distributed actor model concurrency like Erlang ... without Erlang.

Even better if they're decoupled and optional.

Rarely will you build an application as part of a distributed system that does not also need local concurrency.

COMMUNICATION MODEL

How do we unify communications in local concurrency and distributed systems across languages?

EXECUTION MODEL

How do we get Erlang-style local concurrency without interfering with the language's idiomatic paradigm?

ZeroMQ

COMMUNICATION MODEL

Misconceptions

Misconceptions

- It's just another MQ, right?

Misconceptions

- It's just another MQ, right?
  - Not really.

Misconceptions

- It's just another MQ, right?
  - Not really.

Misconceptions

- It's just another MQ, right?
  - Not really.
- Oh, it's just sockets, right?

Misconceptions

- It's just another MQ, right?
  - Not really.
- Oh, it's just sockets, right?
  - Not really.

Misconceptions

- It's just another MQ, right?
  - Not really.
- Oh, it's just sockets, right?
  - Not really.

Misconceptions

- It's just another MQ, right?
  - Not really.
- Oh, it's just sockets, right?
  - Not really.
- Wait, isn't messaging a solved problem?

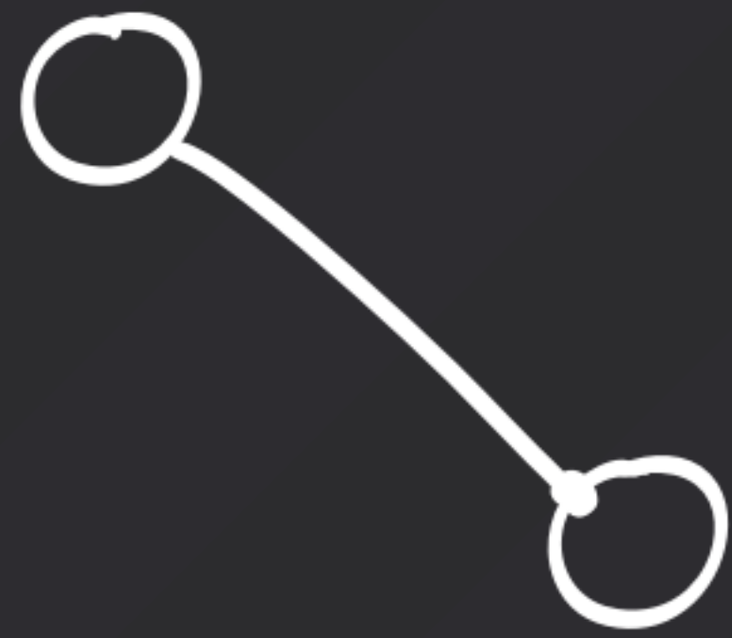
Misconceptions

- It's just another MQ, right?
  - Not really.
- Oh, it's just sockets, right?
  - Not really.
- Wait, isn't messaging a solved problem?
  - *sigh* ... maybe.

Regular Sockets

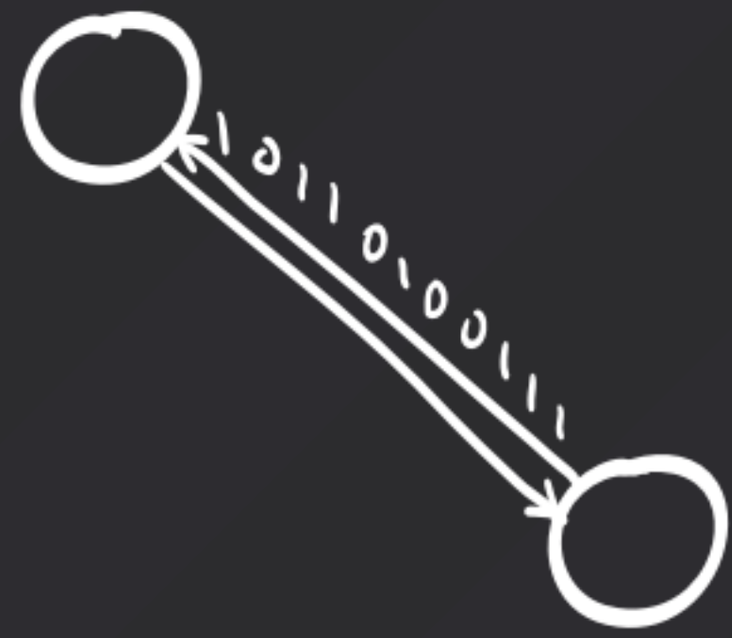
Regular Sockets

- Point to point



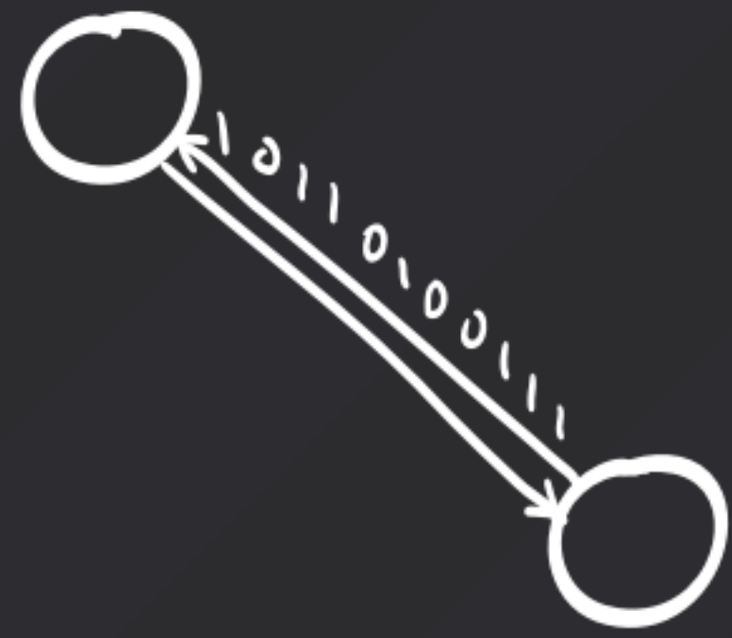
Regular Sockets

- Point to point
- Stream of bytes



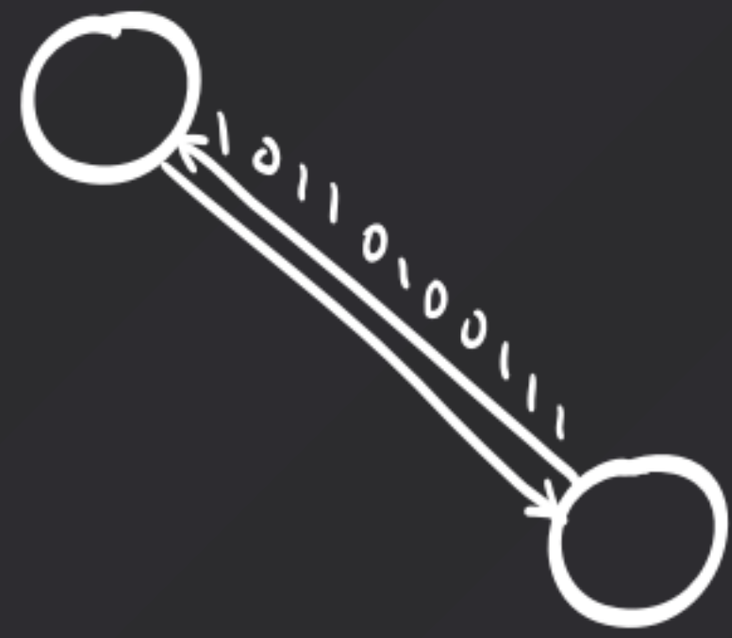
Regular Sockets

- Point to point
- Stream of bytes
- Buffering



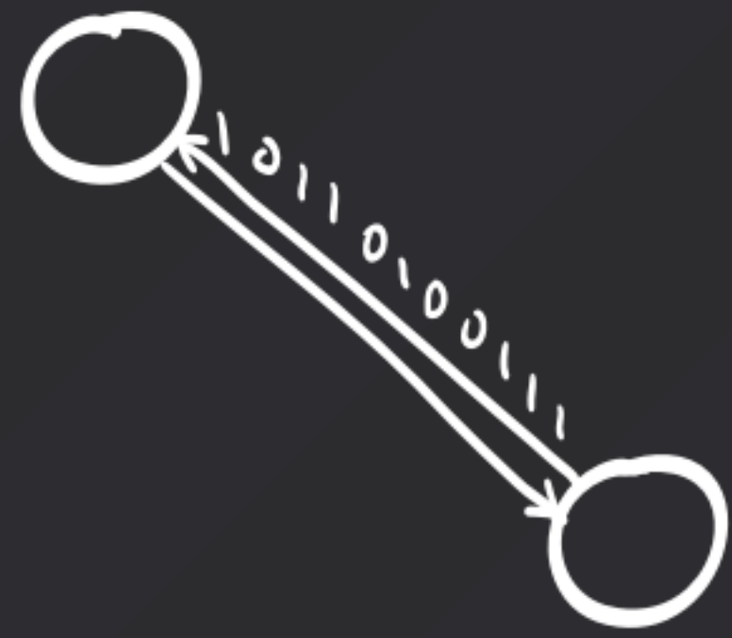
Regular Sockets

- Point to point
- Stream of bytes
- Buffering
- Standard API



Regular Sockets

- Point to point
- Stream of bytes
- Buffering
- Standard API
- TCP/IP or UDP, IPC



Messaging

Messaging

Messages are atomic

Messaging

Messages are atomic

Messaging

Messages are atomic

Messaging

Messages are atomic

Messaging

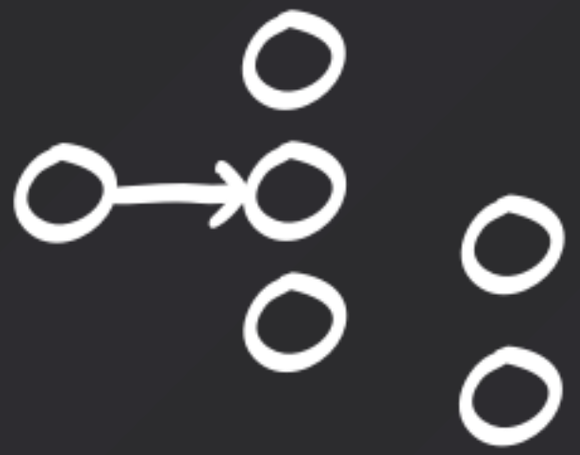
Messages are atomic

Messages can be routed

Messaging

Messages are atomic

Messages can be routed



Messaging

Messages are atomic

Messages can be routed

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messaging

Messages are atomic

Messages can be routed

Messages may sit around

Messages are delivered

Messaging

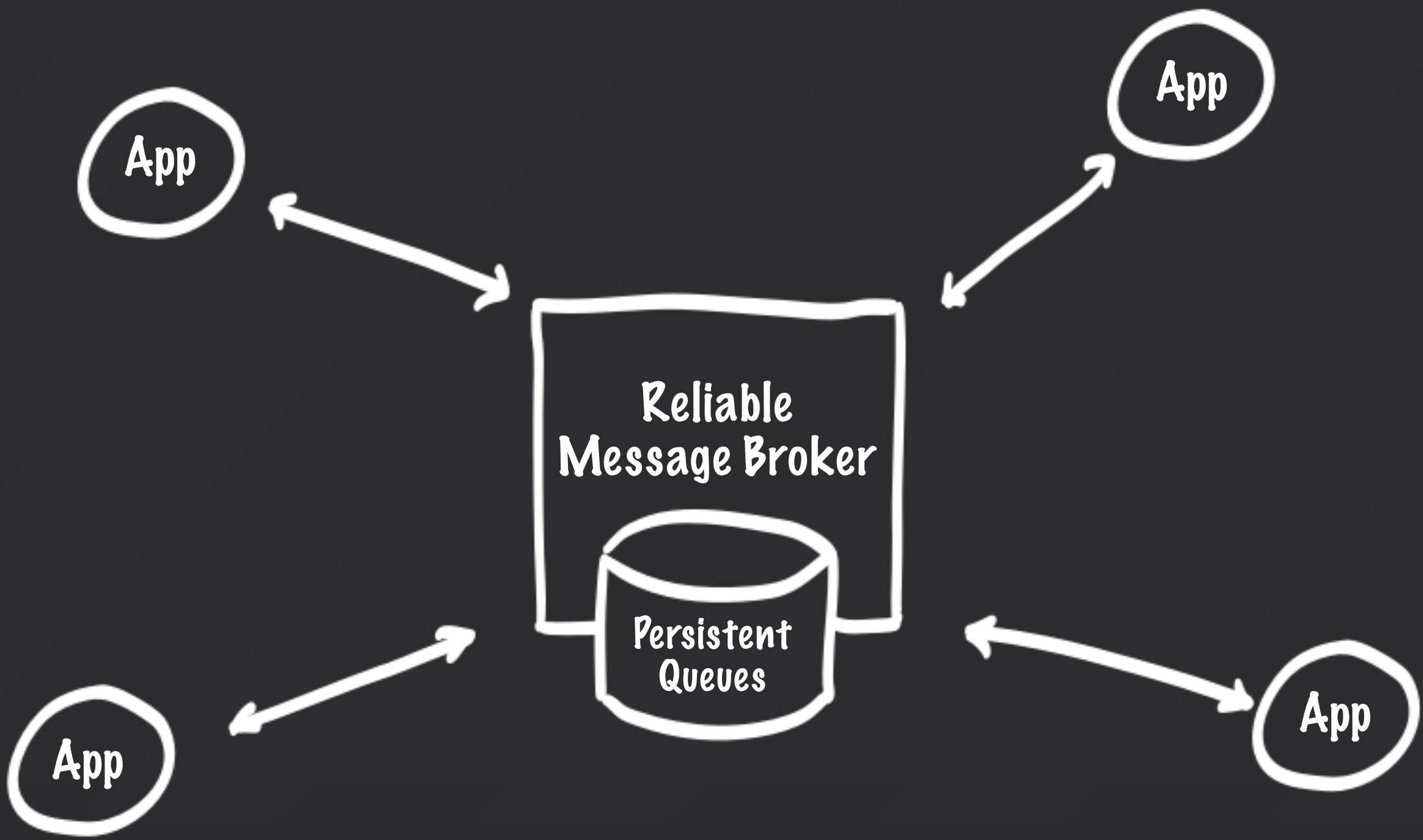
Messages are atomic

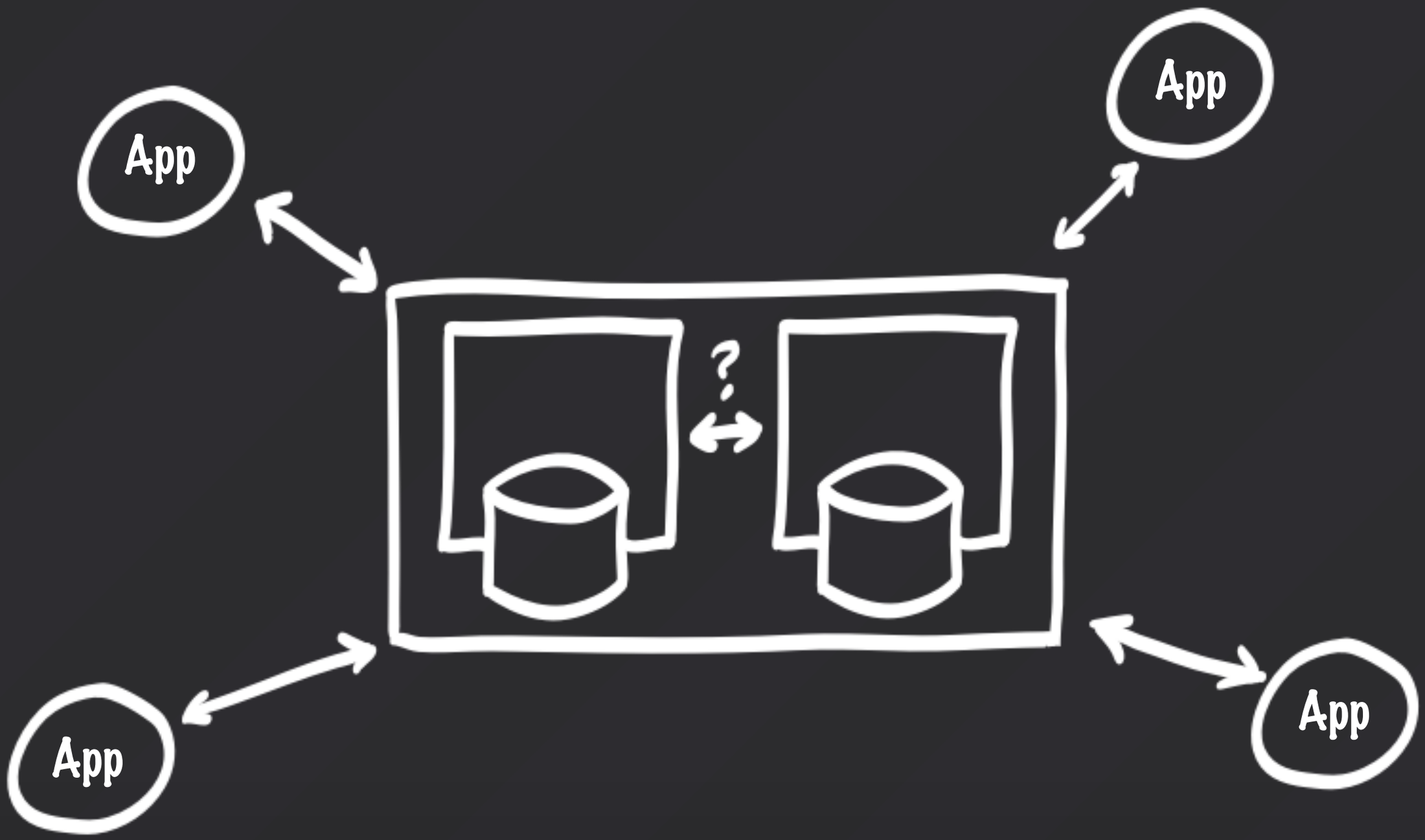
Messages can be routed

Messages may sit around

Messages are delivered

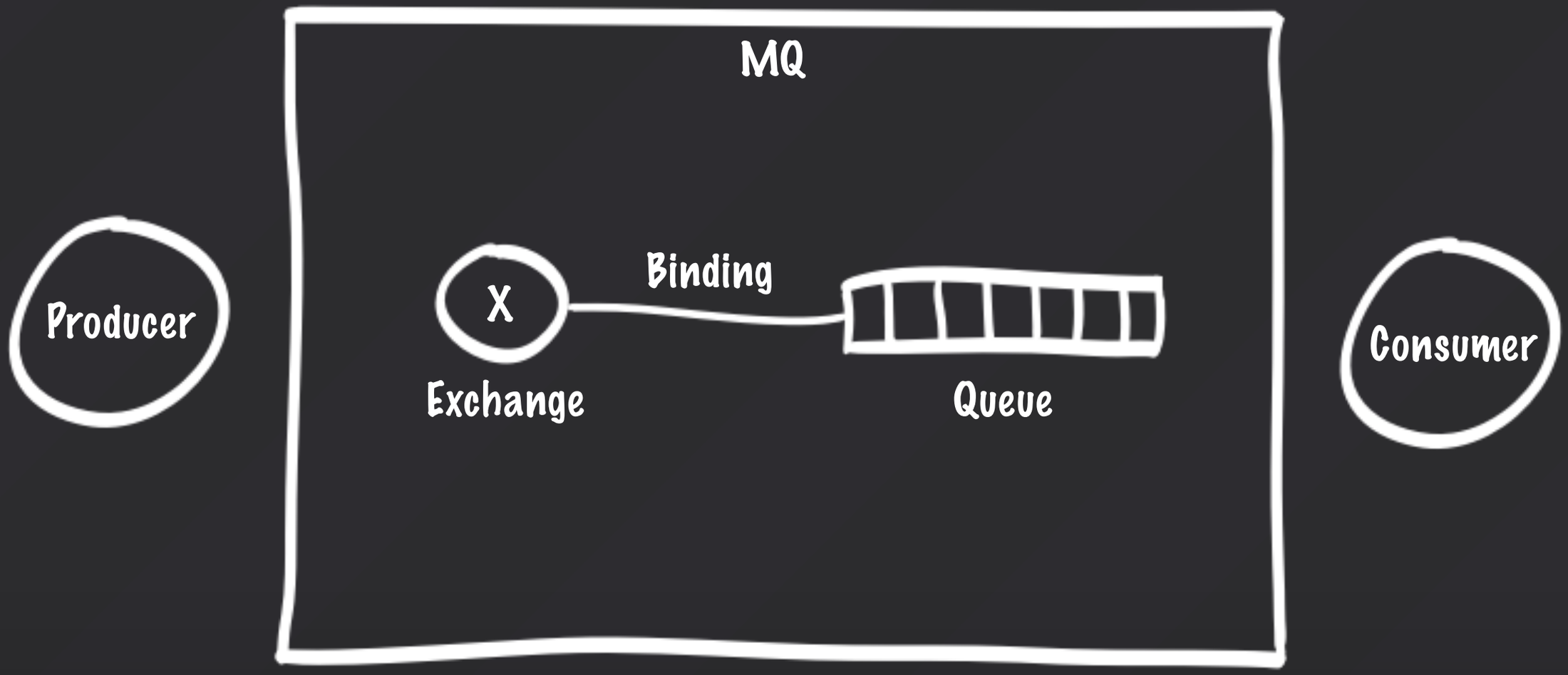
Rise of the Big MQ


App
App
Reliable
Message Broker
Persistent
Queues
App
App

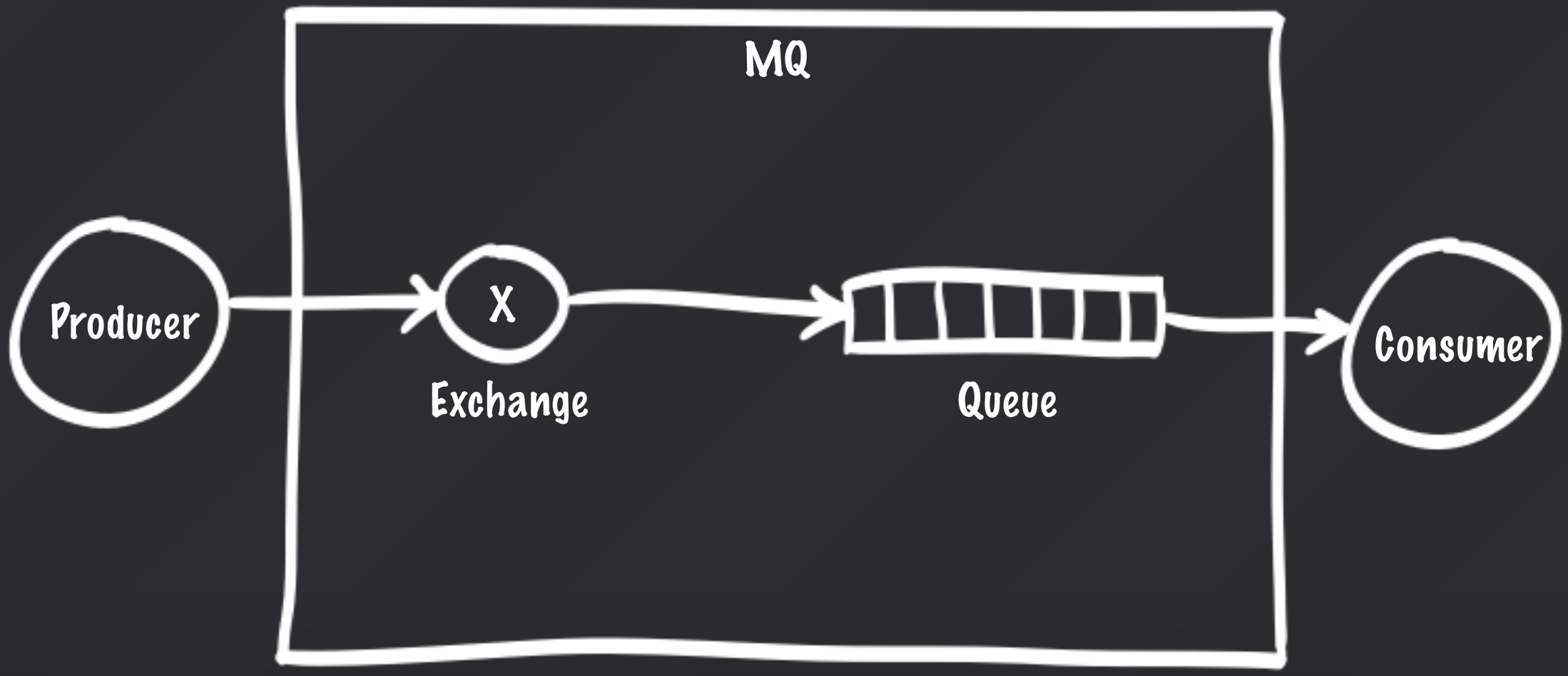

App
App
?
App
App

AMQP

AMQP



AMQP



AMQP Recipes

AMQP Recipes

Work queues

Distributing tasks among workers

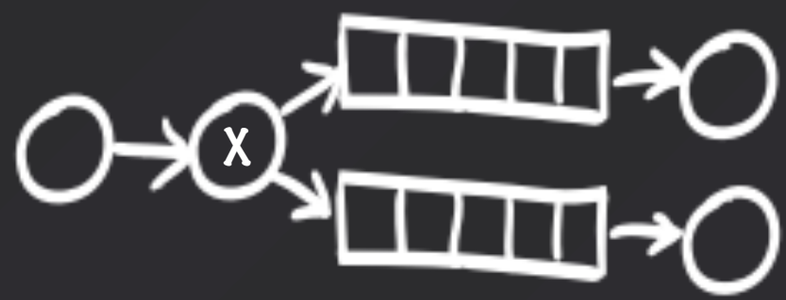
AMQP Recipes

Work queues

Distributing tasks among workers

Publish/Subscribe

Sending to many consumers at once



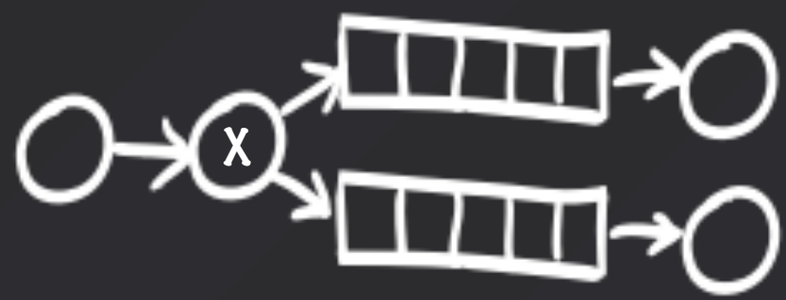
AMQP Recipes

Work queues

Distributing tasks among workers

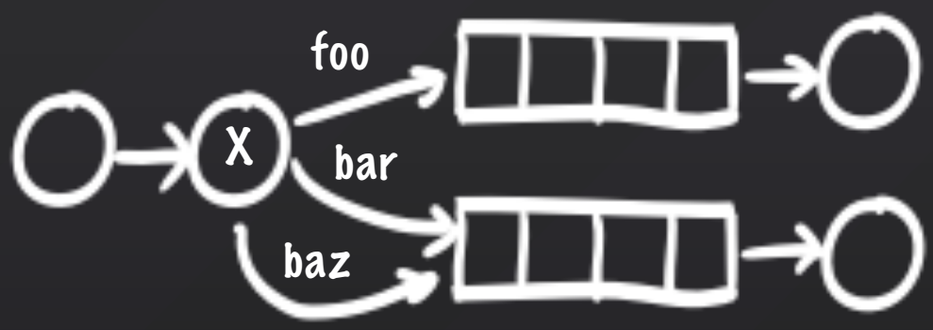
Publish/Subscribe

Sending to many consumers at once



Routing

Receiving messages selectively



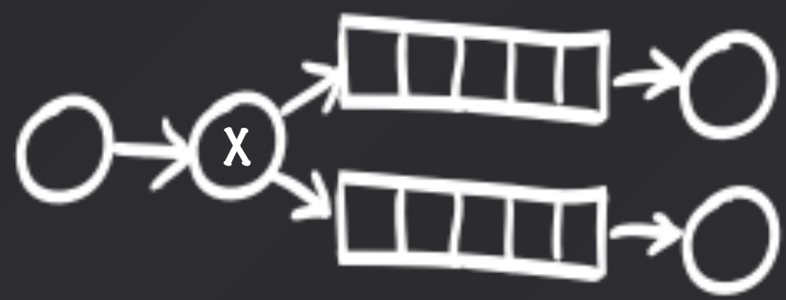
AMQP Recipes

Work queues

Distributing tasks among workers

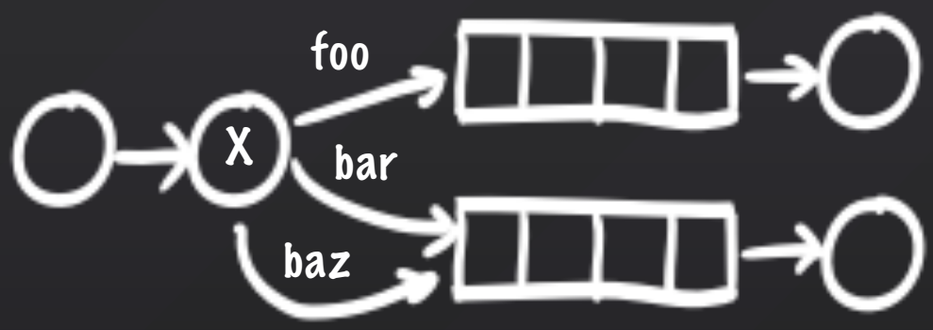
Publish/Subscribe

Sending to many consumers at once



Routing

Receiving messages selectively



RPC

Remote procedure call implementation

Drawbacks of Big MQ

- Lots of complexity
- Queues are heavyweight
- HA is a challenge
- Poor primitives

Enter ZeroMQ

"Float like a butterfly, sting like a bee"

Echo in Python

Server

```
1 import zmq
2 context = zmq.Context()
3 socket = context.socket(zmq.REP)
4 socket.bind("tcp://127.0.0.1:5000")
5
6 while True:
7     msg = socket.recv()
8     print "Received", msg
9     socket.send(msg)
```

Client

```
1 import zmq
2 context = zmq.Context()
3 socket = context.socket(zmq.REQ)
4 socket.connect("tcp://127.0.0.1:5000")
5
6 for i in range(10):
7     msg = "msg %s" % i
8     socket.send(msg)
9     print "Sending", msg
10    reply = socket.recv()
```

Echo in Ruby

Server

```ruby
require "zmq"
context = ZMQ::Context.new(1)
socket = context.socket(ZMQ::REP)
socket.bind("tcp://127.0.0.1:5000")

loop do
    msg = socket.recv
    puts "Received #{msg}"
    socket.send(msg)
end
```

Client

```ruby
require "zmq"
context = ZMQ::Context.new(1)
socket = context.socket(ZMQ::REQ)
socket.connect("tcp://127.0.0.1:5000")

(0...10).each do |i|
    msg = "msg #{i}"
    socket.send(msg)
    puts "Sending #{msg}"
    reply = socket.recv
end
```

Echo in PHP

Server

```php
<?php
$context = new ZMQContext();
$socket = $context->getSocket(ZMQ::SOCKET_REP);
$socket->bind("tcp://127.0.0.1:5000");

while (true) {
    $msg = $socket->recv();
    echo "Received {$msg}";
    $socket->send($msg);
}
?>
```

Client

```php
<?php
$context = new ZMQContext();
$socket = $context->getSocket(ZMQ::SOCKET_REQ);
$socket->connect("tcp://127.0.0.1:5000");

foreach (range(0, 9) as $i) {
    $msg = "msg {$i}";
    $socket->send($msg);
    echo "Sending {$msg}";
    $reply = $socket->recv();
}
?>
```

Bindings

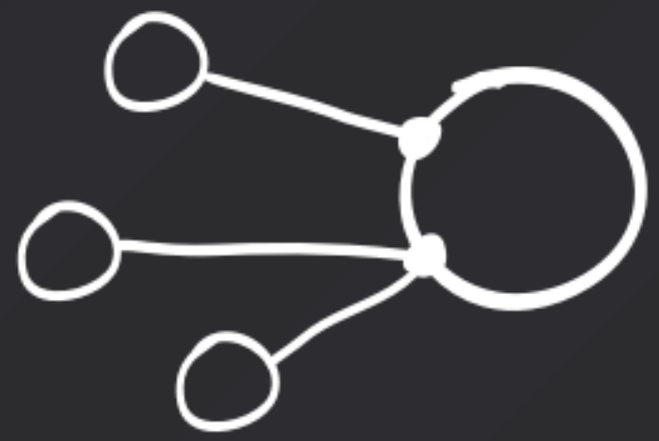
ActionScript, Ada, Bash, Basic, C, Chicken Scheme, Common Lisp, C#, C++, D, Erlang, F#, Go, Guile, Haskell, Haxe, Java, JavaScript, Lua, Node.js, Objective-C, Objective Caml, ooc, Perl, PHP, Python, Racket, REBOL, Red, Ruby, Smalltalk

Plumbing

Plumbing

Plumbing

Plumbing



Plumbing

Plumbing

- inproc
- ipc
- tcp
- multicast

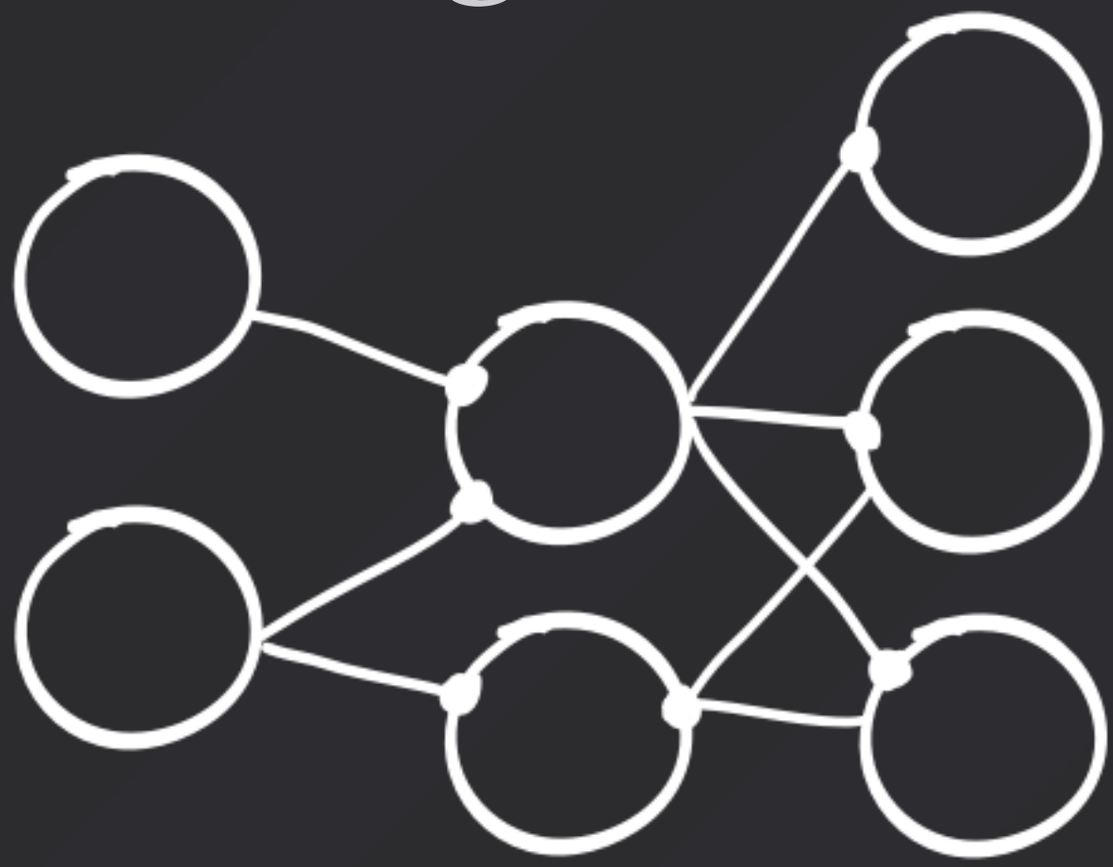
Plumbing

- inproc
- ipc
- tcp
- multicast

```
socket.bind("tcp://localhost:5560")
socket.bind("ipc:///tmp/this-socket")
socket.connect("tcp://10.0.0.100:9000")
socket.connect("ipc:///tmp/another-socket")
socket.connect("inproc://another-socket")
```

Plumbing

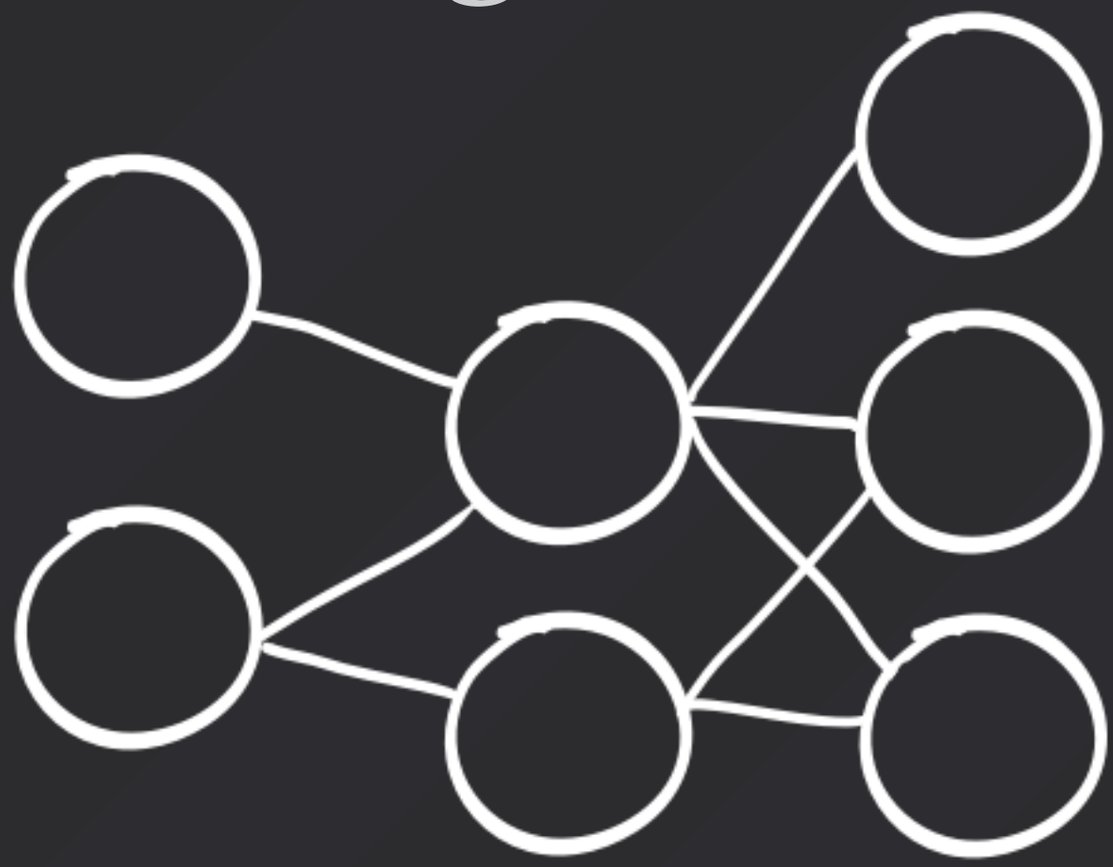
- inproc
- ipc
- tcp
- multicast



```
socket.bind("tcp://localhost:5560")
socket.bind("ipc:///tmp/this-socket")
socket.connect("tcp://10.0.0.100:9000")
socket.connect("ipc:///tmp/another-socket")
socket.connect("inproc://another-socket")
```

Plumbing

- inproc
- ipc
- tcp
- multicast



```
socket.bind("tcp://localhost:5560")
socket.bind("ipc:///tmp/this-socket")
socket.connect("tcp://10.0.0.100:9000")
socket.connect("ipc:///tmp/another-socket")
socket.connect("inproc://another-socket")
```

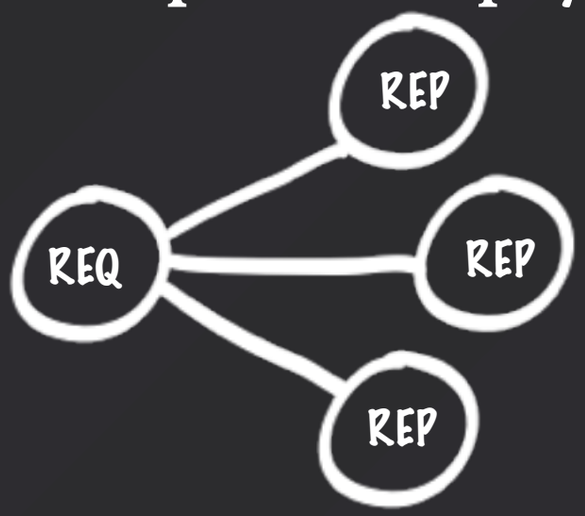
Message Patterns

Message Patterns

Request-Reply

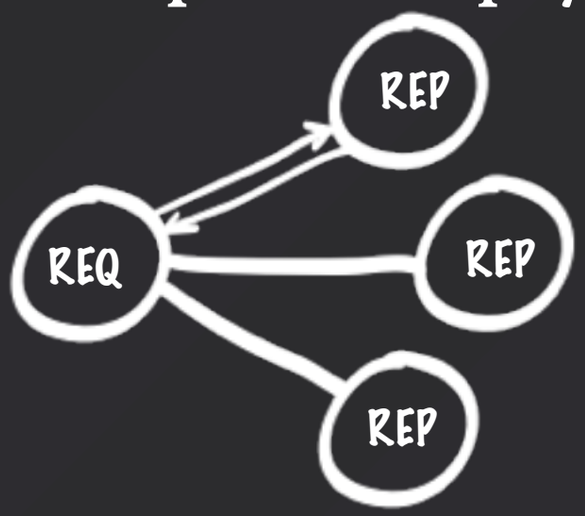
Message Patterns

Request-Reply



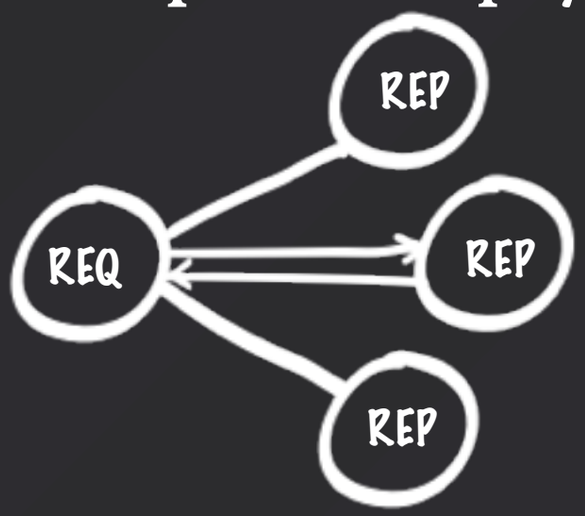
Message Patterns

Request-Reply



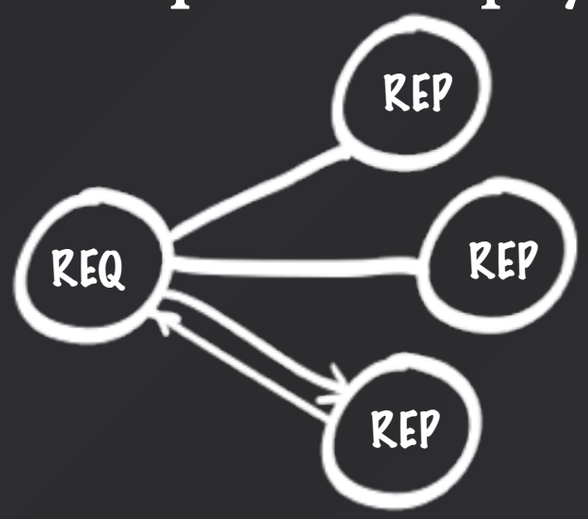
Message Patterns

Request-Reply



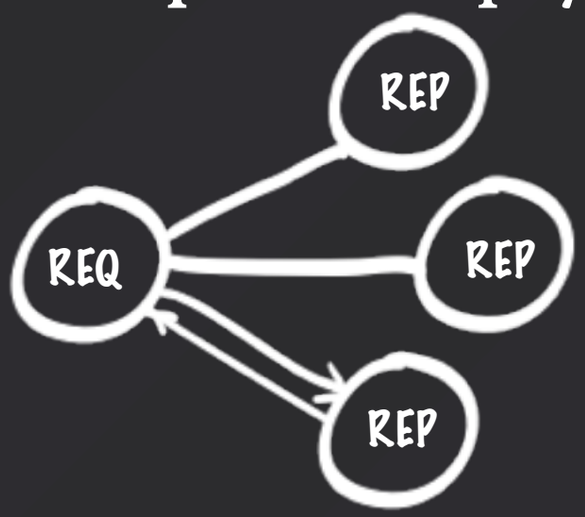
Message Patterns

Request-Reply

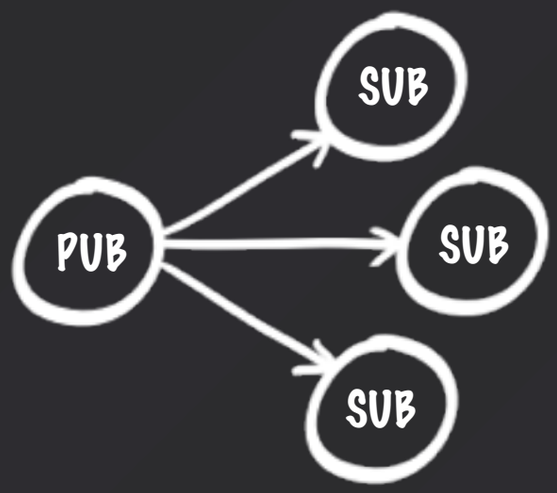


Message Patterns

Request-Reply

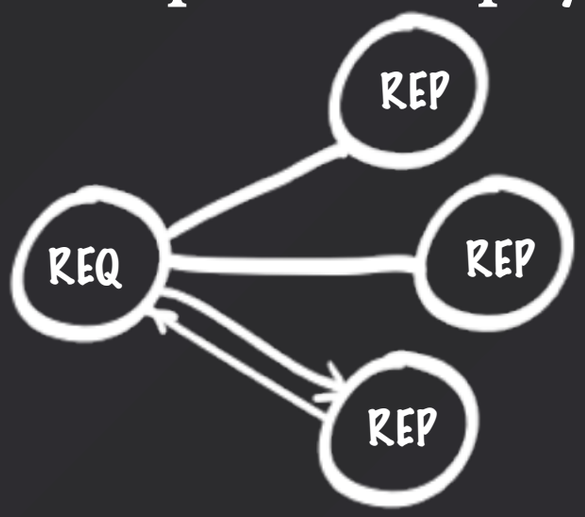


Publish-Subscribe

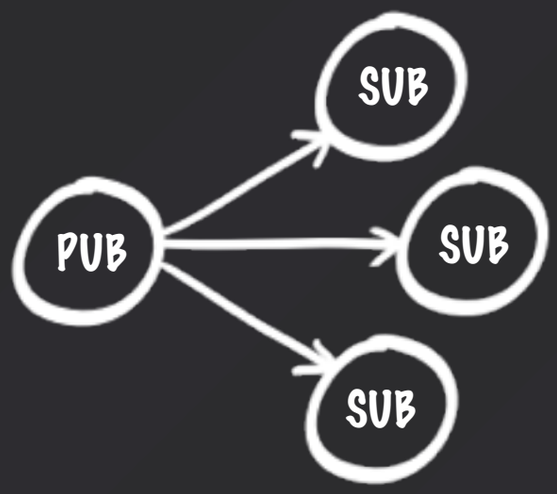


Message Patterns

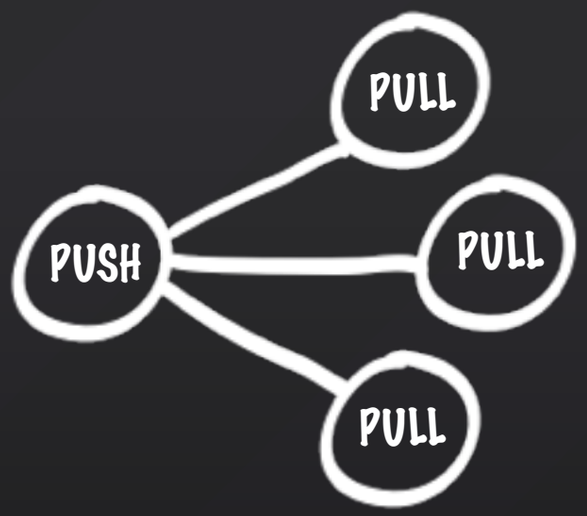
Request-Reply



Publish-Subscribe

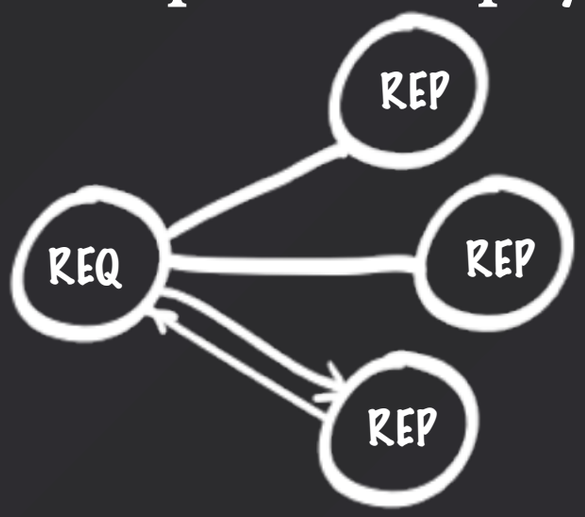


Push-Pull (Pipelining)

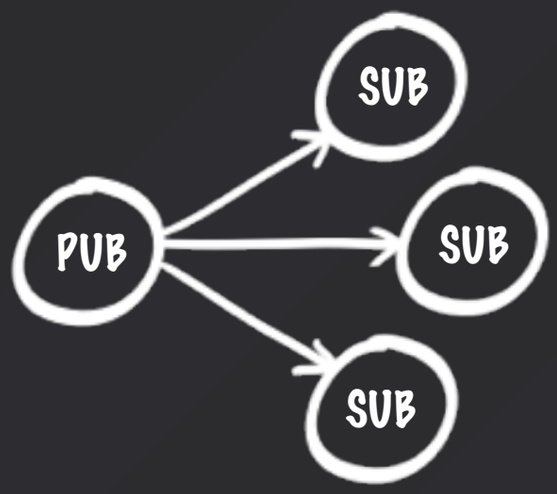


Message Patterns

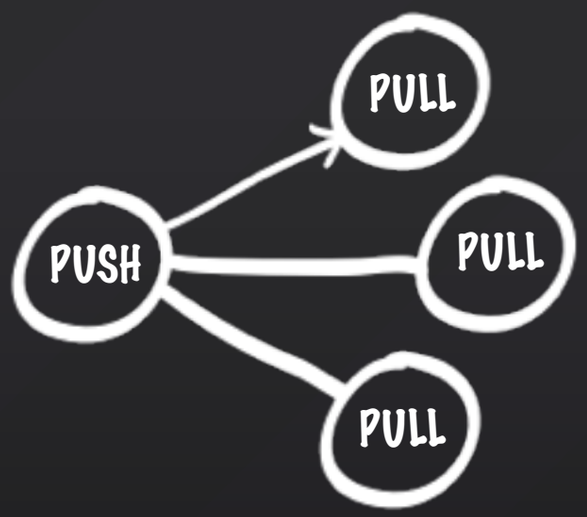
Request-Reply



Publish-Subscribe

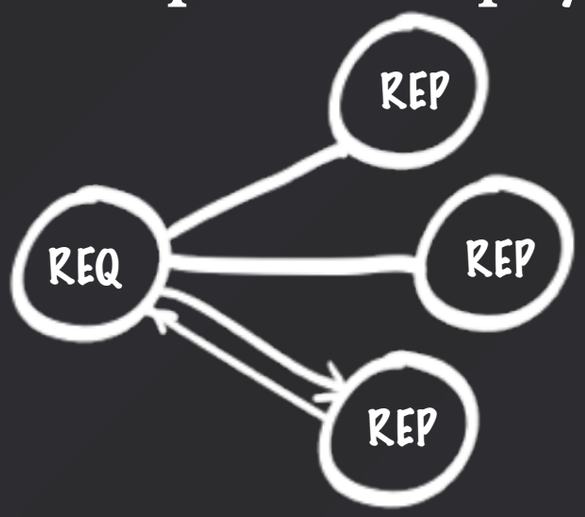


Push-Pull (Pipelining)

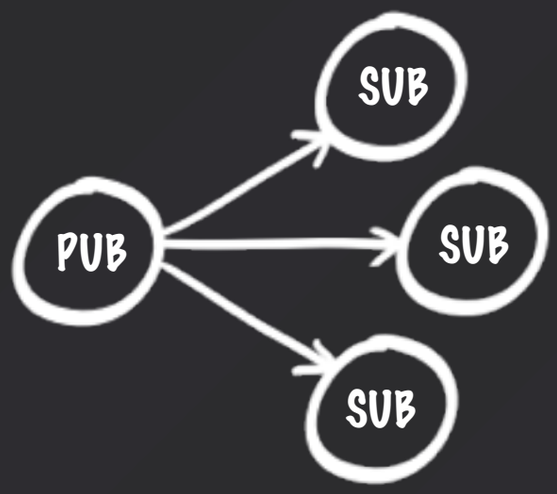


Message Patterns

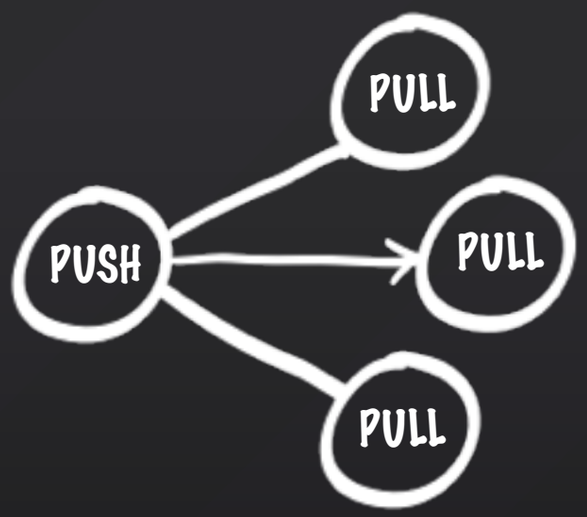
Request-Reply



Publish-Subscribe

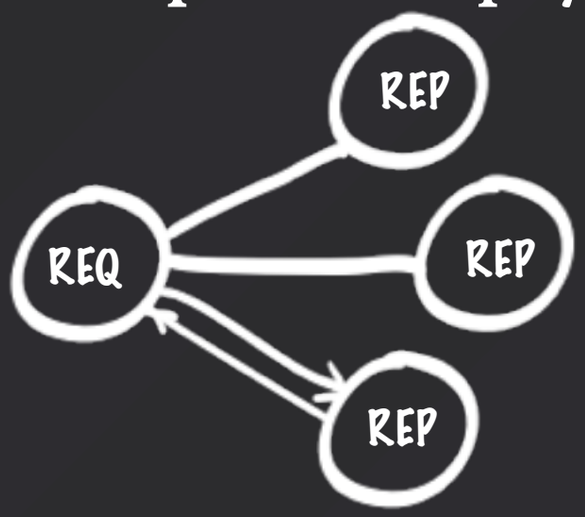


Push-Pull (Pipelining)

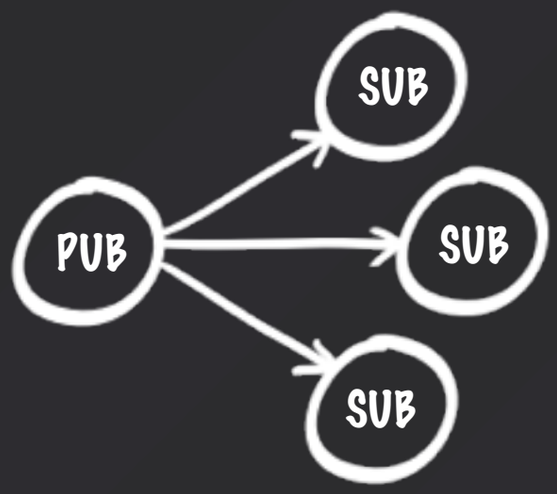


Message Patterns

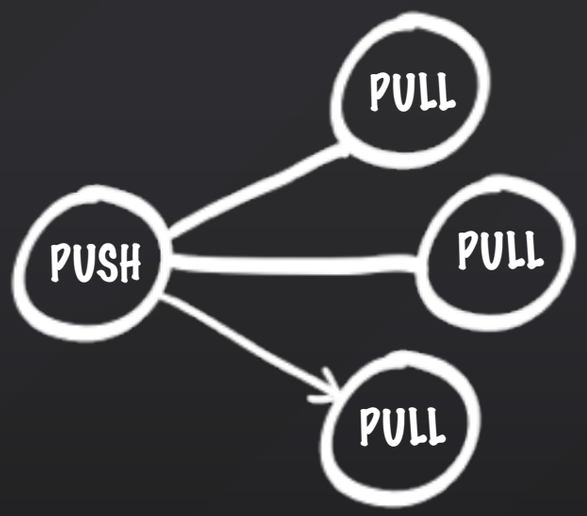
Request-Reply



Publish-Subscribe

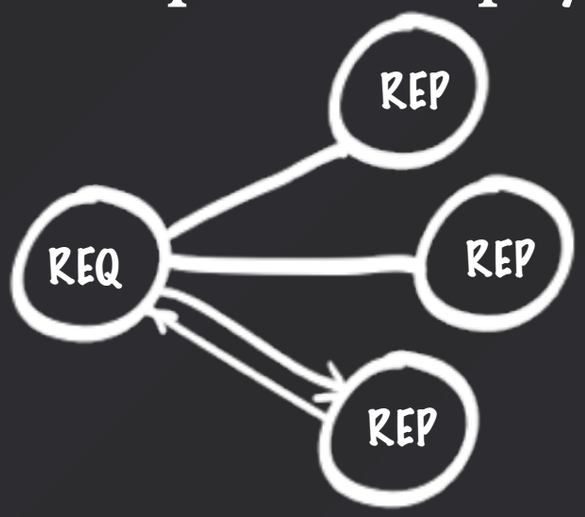


Push-Pull (Pipelining)

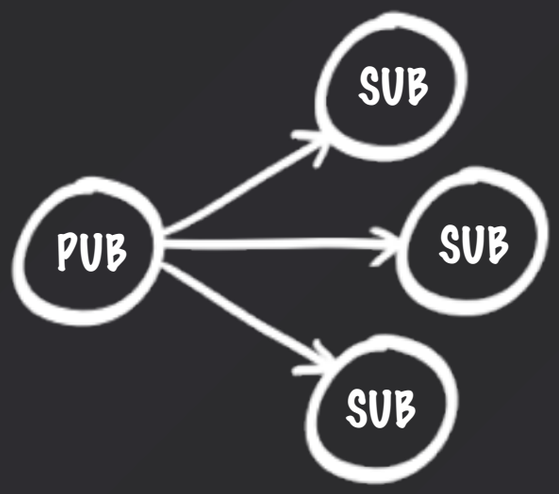


Message Patterns

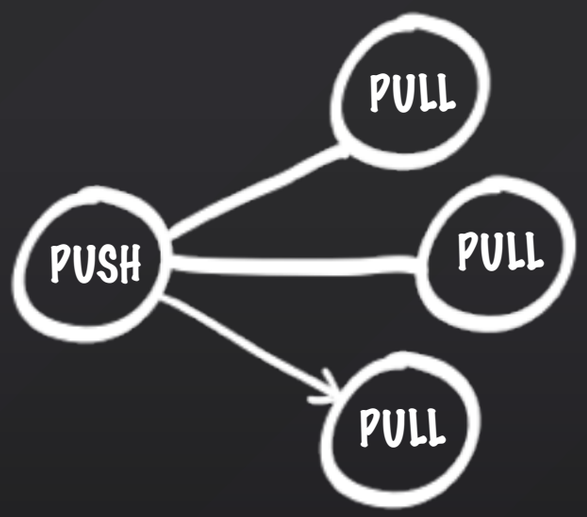
Request-Reply



Publish-Subscribe



Push-Pull (Pipelining)



Pair

Devices

Queue

Forwarder

Streamer

Design architectures around devices.

Devices

Queue Forwarder Streamer

Design architectures around devices.

Devices

Queue Forwarder Streamer

Design architectures around devices.

Devices

Queue Forwarder Streamer

Design architectures around devices.

Performance

Performance

- Orders of magnitude faster than most MQs

Performance

- Orders of magnitude faster than most MQs
- Higher throughput than raw sockets

Performance

- Orders of magnitude faster than most MQs
- Higher throughput than raw sockets
  - Intelligent message batching

Performance

- Orders of magnitude faster than most MQs
- Higher throughput than raw sockets
  - Intelligent message batching
  - Edge case optimizations

Concurrency?

"Come for the messaging, stay for the easy concurrency"

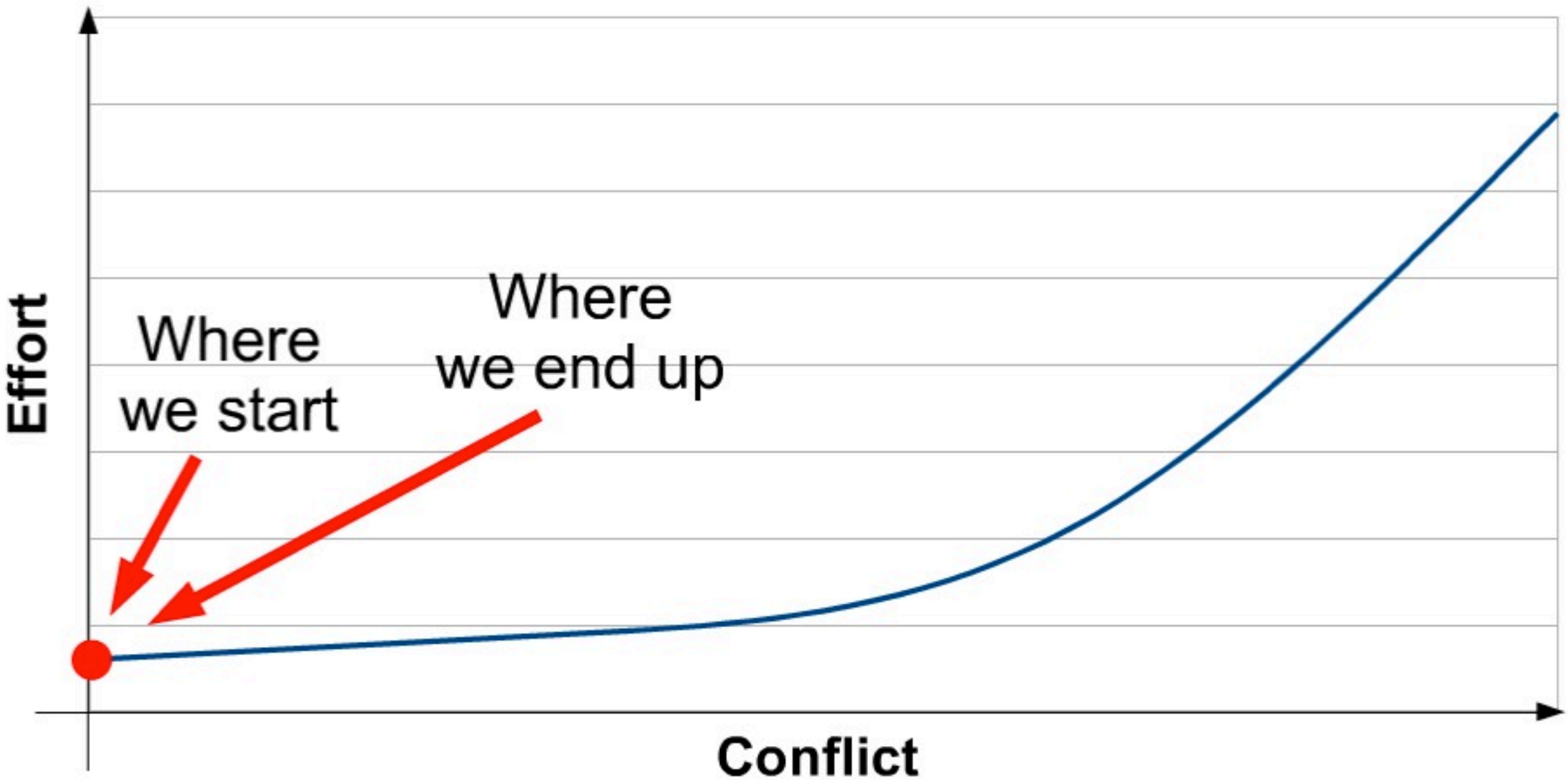
Hintjens' Law of Concurrency

$$E = MC^2$$

E is effort, the pain that it takes
M is mass, the size of the code
C is conflict, when C threads collide

Hintjens' Law of Concurrency

Hintjens' Law of Concurrency



Hintjens' Law of Concurrency

ZeroMQ:

$E=MC^2$, FOR C=1

ZeroMQ

- Easy ... familiar socket API
- Cheap ... lightweight queues in a library
- Fast ... higher throughput than raw TCP
- Expressive ... maps to your architecture

Messaging toolkit for concurrency and distributed systems.

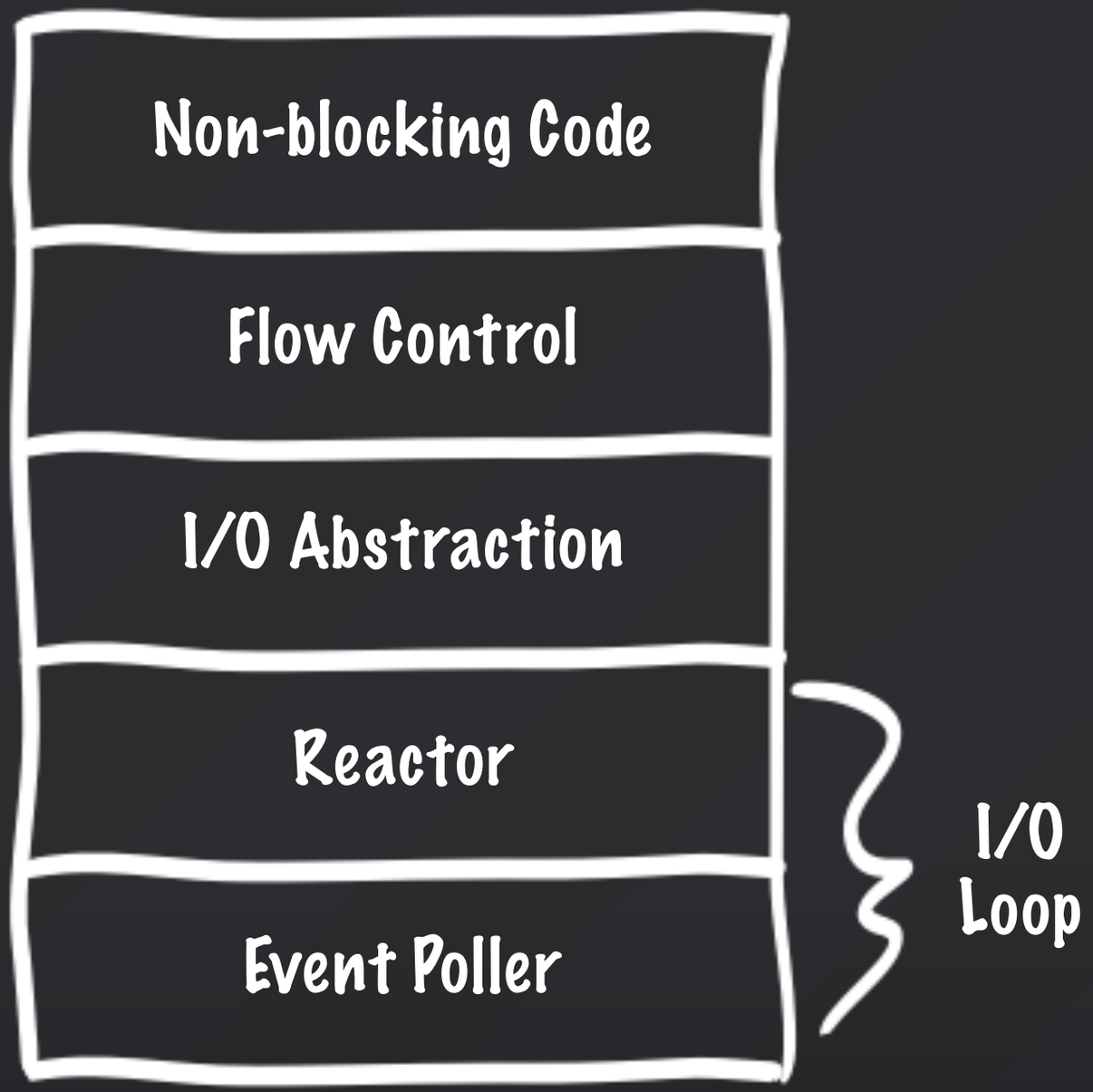
gevent

EXECUTION MODEL

Threading vs Evented

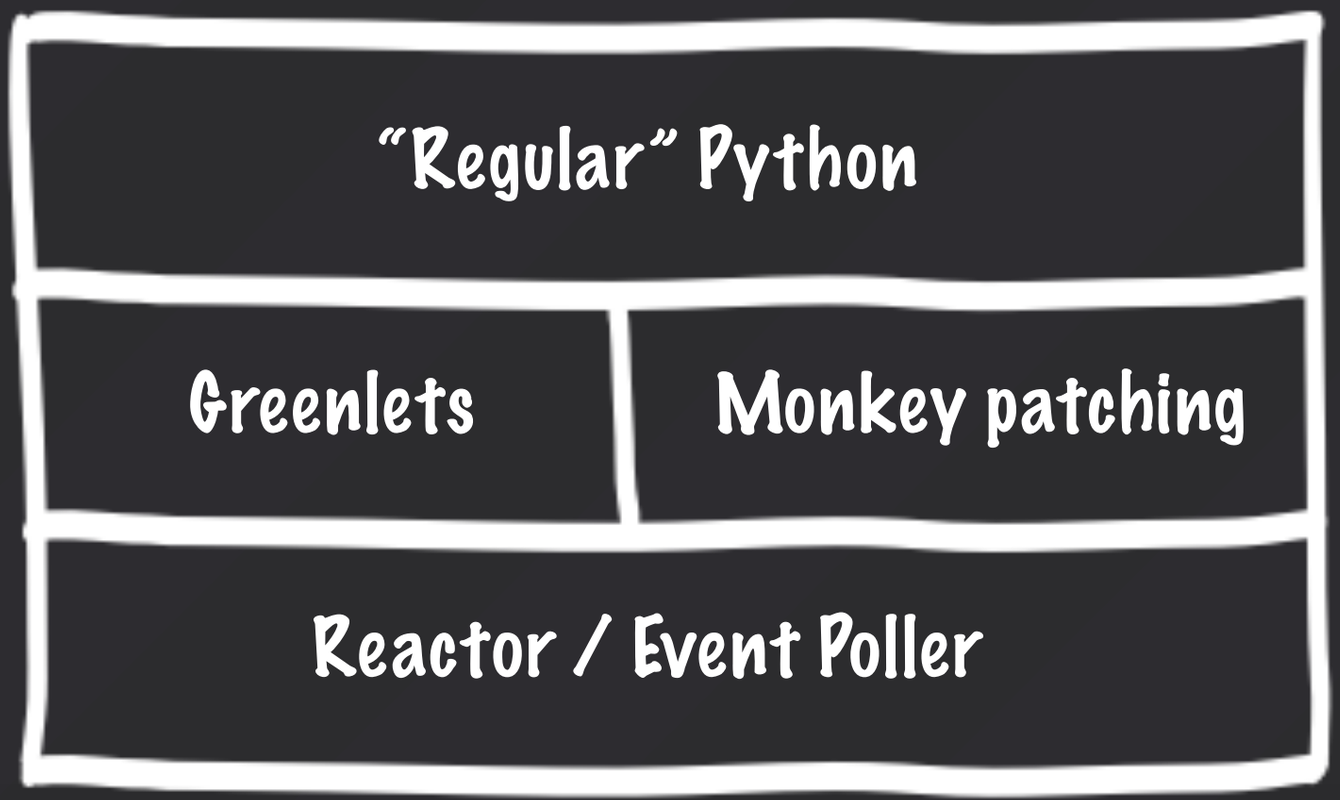
Evented seems to be preferred for scalable I/O applications

Evented Stack



```
def lookup(country, search_term):
    main_d = defer.Deferred()

    def first_step():
        query = "http://www.google.%s/search?q=%s" % (country,search_term)
        d = getPage(query)
        d.addCallback(second_step, country)
        d.addErrback(failure, country)

    def second_step(content, country):
        m = re.search('<div id="?res.*?href="(?P<url>http://[^"]+)"',
                      content, re.DOTALL)
        if not m:
            main_d.callback(None)
            return
        url = m.group('url')
        d = getPage(url)
        d.addCallback(third_step, country, url)
        d.addErrback(failure, country)

    def third_step(content, country, url):
        m = re.search("<title>(.*?)</title>", content)
        if m:
            title = m.group(1)
            main_d.callback(dict(url = url, title = title))
        else:
            main_d.callback(dict(url=url, title="{not-specified}"))

    def failure(e, country):
        print ".%s FAILED: %s" % (country, str(e))
        main_d.callback(None)

    first_step()
    return main_d
```

gevent


"Regular" Python
Greenlets
Monkey patching
Reactor / Event Poller

Green threads

“Threads” implemented in user space (VM, library)

Monkey patching

socket, ssl, threading, time

Twisted

Twisted

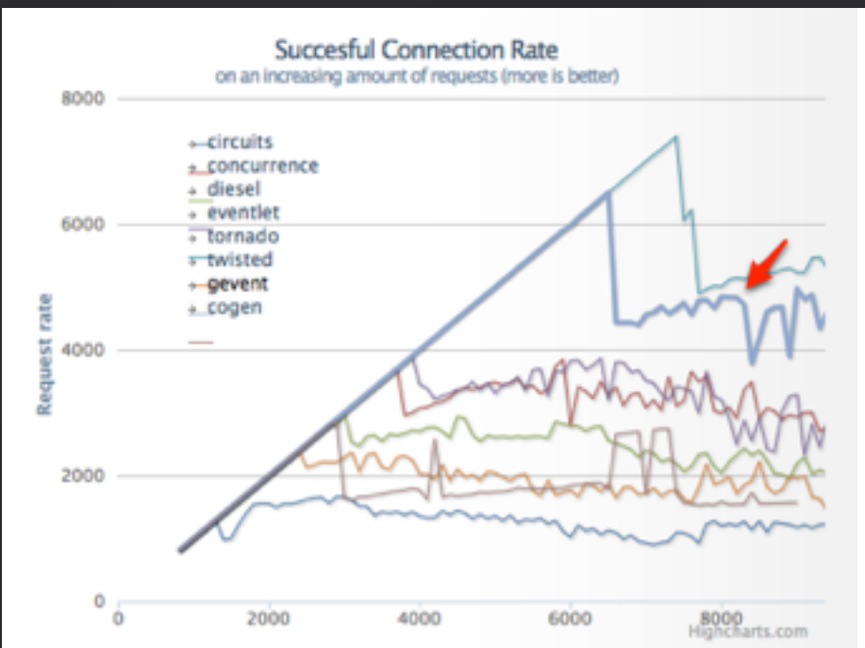
~400 modules

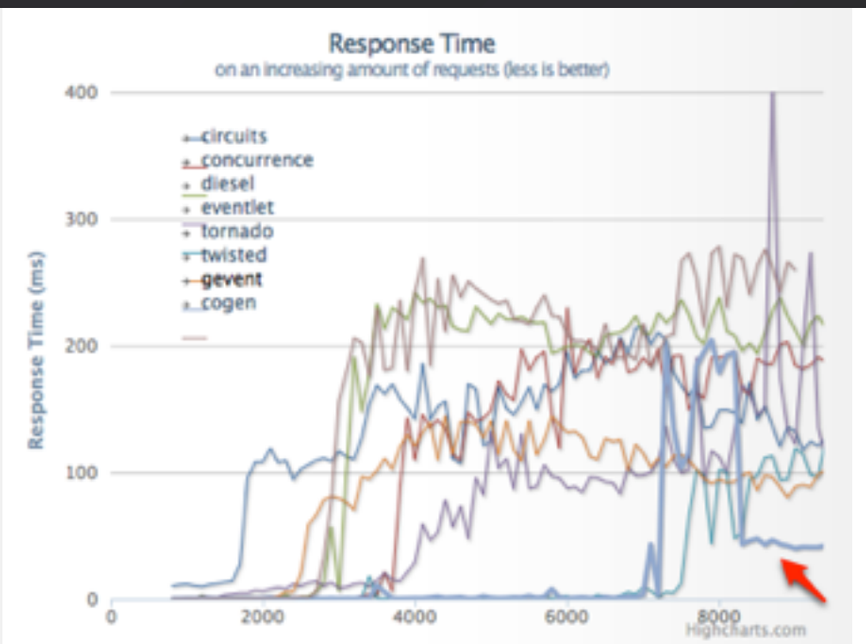
gevent

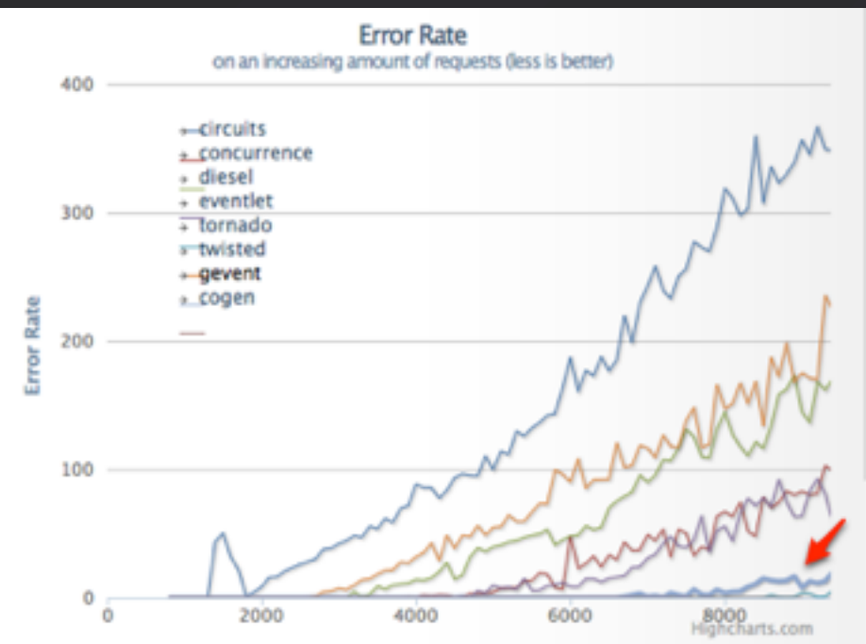
25 modules

Performance

Performance

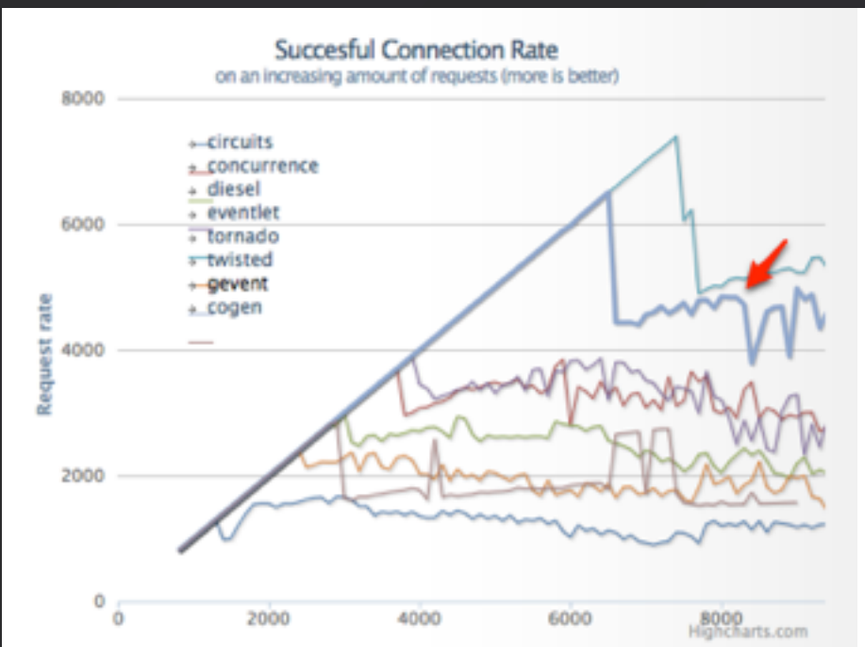


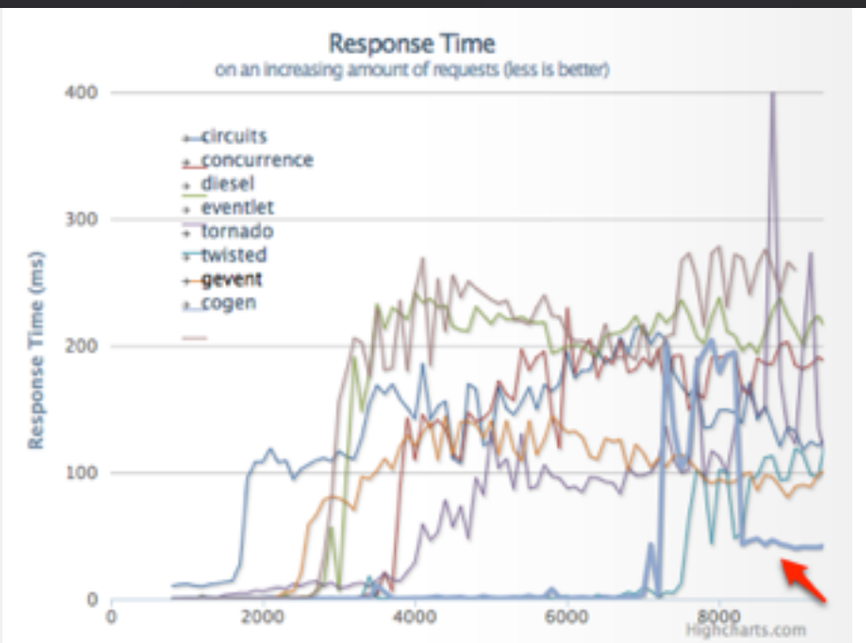


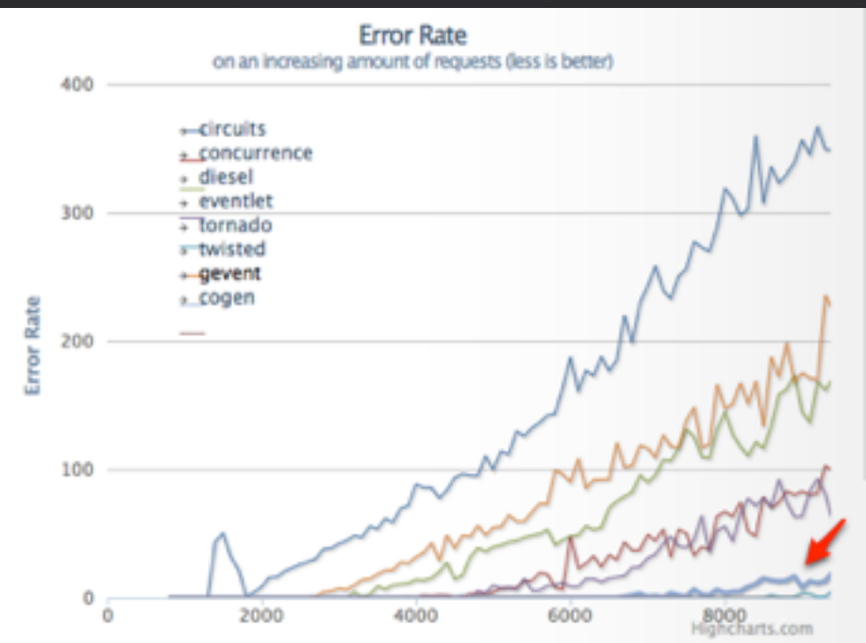


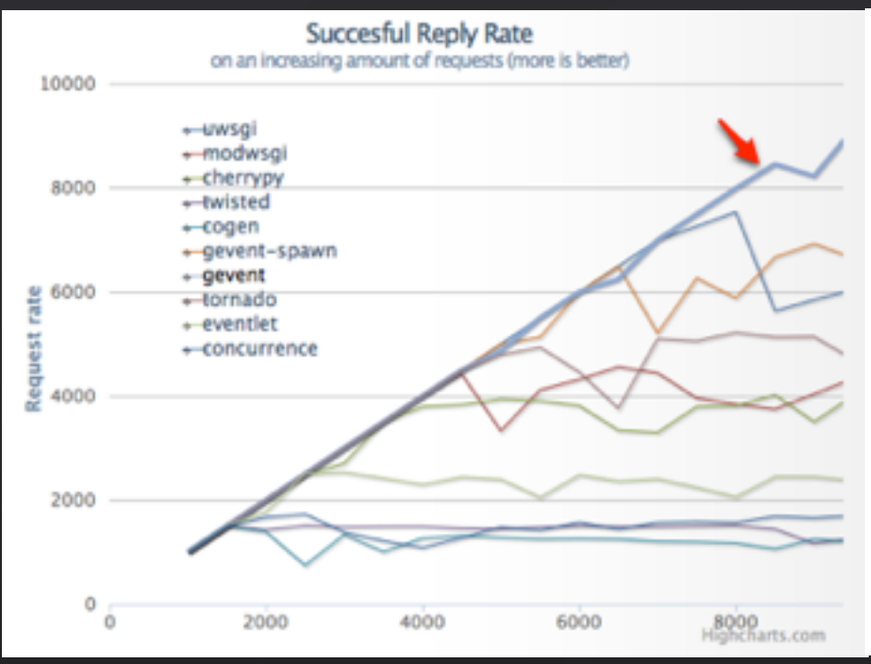
http://nichol.as

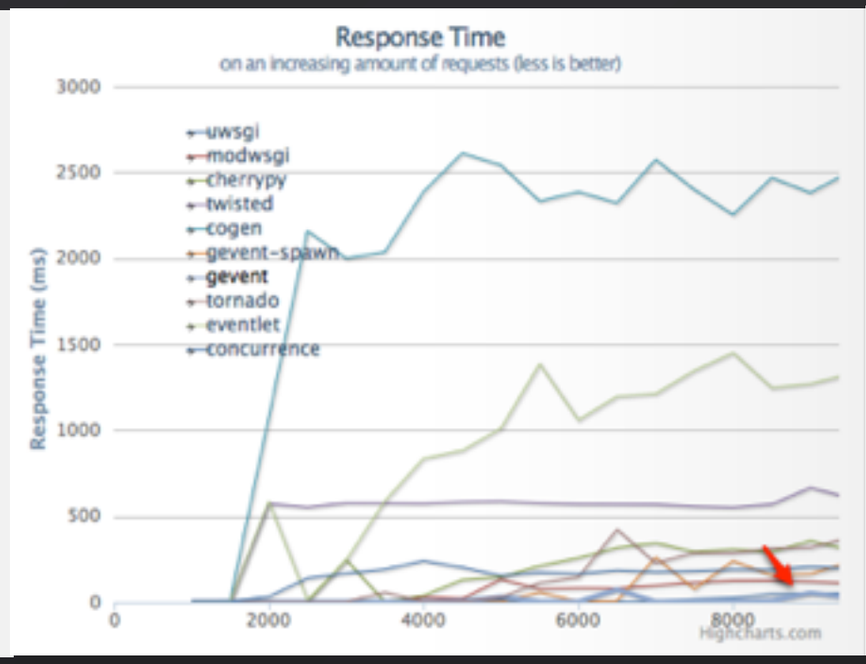
Performance

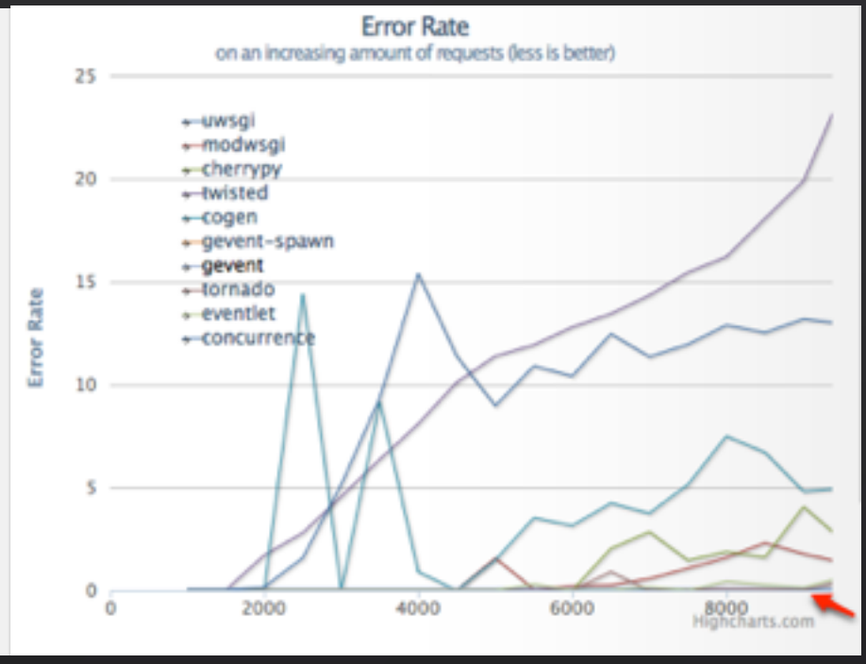












Building a Networking App

```
#===
# 1. Basic gevent TCP server

from gevent.server import StreamServer

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

tcp_server = StreamServer(('127.0.0.1', 1234), handle_tcp)
tcp_server.serve_forever()
```

```python
#===
# 2. Basic gevent TCP server and WSGI server

from gevent.pywsgi import WSGIServer
from gevent.server import StreamServer

def handle_http(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    print 'new http request!'
    return ["hello world"]

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

tcp_server = StreamServer(('127.0.0.1', 1234), handle_tcp)
tcp_server.start()

http_server = WSGIServer(('127.0.0.1', 8080), handle_http)
http_server.serve_forever()
```

```python
from gevent.pywsgi import WSGIServer
from gevent.server import StreamServer
from gevent.socket import create_connection

def handle_http(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    print 'new http request!'
    return ["hello world"]

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

def client_connect(address):
    sockfile = create_connection(address).makefile()
    while True:
        line = sockfile.readline() # returns None on EOF
        if line is not None:
            print "<<<", line,
        else:
            break

tcp_server = StreamServer(('127.0.0.1', 1234), handle_tcp)
tcp_server.start()

gevent.spawn(client_connect, ('127.0.0.1', 1234))

http_server = WSGIServer(('127.0.0.1', 8080), handle_http)
http_server.serve_forever()
```

```python
from gevent.pywsgi import WSGIServer
from gevent.server import StreamServer
from gevent.socket import create_connection

def handle_http(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    print 'new http request!'
    return ["hello world"]

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

def client_connect(address):
    sockfile = create_connection(address).makefile()
    while True:
        line = sockfile.readline() # returns None on EOF
        if line is not None:
            print "<<<", line,
        else:
            break

tcp_server = StreamServer(('127.0.0.1', 1234), handle_tcp)
http_server = WSGIServer(('127.0.0.1', 8080), handle_http)
greenlets = [
    gevent.spawn(tcp_server.serve_forever),
    gevent.spawn(http_server.serve_forever),
    gevent.spawn(client_connect, ('127.0.0.1', 1234)),
]
gevent.joinall(greenlets)
```

ZeroMQ in gevent?

traviscline / gevent-zeromq

pyzeromq gevent compat. lib — Read more

bumped version number to represent _state_event refactor

traviscline authored June 29, 2011 commit ead6ce80f4

gevent-zeromq /

name	age	message	history
examples/	January 25, 2011	changed to respect edge-triggered nature of zmq.FD... [traviscline]	
gevent_zeromq/	June 29, 2011	_state_event refactor [traviscline]	
.gitignore	April 19, 2011	updated setup.py in prep for pypi [traviscline]	
AUTHORS	March 02, 2011	added AUTHORS [traviscline]	
LICENSE	January 25, 2011	added LICENSE, added to README(.rst) [traviscline]	
MANIFEST.in	April 20, 2011	Changed to include core.c in MANIFEST.in so buildi... [traviscline]	
		Slight comment wording updates, version bump. [traviscline]	

```python
from gevent import spawn
from gevent_zeromq import zmq

context = zmq.Context()

def serve():
    socket = context.socket(zmq.REP)
    socket.bind("tcp://localhost:5559")
    while True:
        message = socket.recv()
        print "Received request: ", message
        socket.send("World")

server = spawn(serve)

def client():
    socket = context.socket(zmq.REQ)
    socket.connect("tcp://localhost:5559")
    for request in range(10):
        socket.send("Hello")
        message = socket.recv()
        print "Received reply ", request, "[", message, "]"

spawn(client).join()
```

Actor model?

Easy to implement, in whole or in part,
optionally with ZeroMQ

1 Answer

active oldest **votes**

2

To make actors with gevent, use a Greenlet subclass with embedded gevent.queue.Queue instance used as an inbox. To read a message from the inbox, simply get() from the queue. To send a message to an actor, put it into that actor's queue.

Read about subclassing Greenlet here.

If you need help with writing the Actor class, feel free to ask the mailing list.

link | edit | flag

answered **Aug 8 '10 at 16:36**

community wiki
Denis Bilenko

Thank you very much! I will dive into gevent, and try it extensively – daitangio Aug 9 '10 at 12:53

add comment

What is gevent missing?

What is gevent missing?

- Documentation

What is gevent missing?

- Documentation
- Application framework

gservice

Application framework for gevent

```python
from gevent.pywsgi import WSGIServer
from gevent.server import StreamServer
from gevent.socket import create_connection

def handle_http(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    print 'new http request!'
    return ["hello world"]

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

def client_connect(address):
    sockfile = create_connection(address).makefile()
    while True:
        line = sockfile.readline() # returns None on EOF
        if line is not None:
            print "<<<", line,
        else:
            break

tcp_server = StreamServer(('127.0.0.1', 1234), handle_tcp)
http_server = WSGIServer(('127.0.0.1', 8080), handle_http)
greenlets = [
    gevent.spawn(tcp_server.serve_forever),
    gevent.spawn(http_server.serve_forever),
    gevent.spawn(client_connect, ('127.0.0.1', 1234)),
]
gevent.joinall(greenlets)
```

```python
from gevent.pywsgi import WSGIServer
from gevent.server import StreamServer
from gevent.socket import create_connection

from gservice.core import Service

def handle_http(env, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    print 'new http request!'
    return ["hello world"]

def handle_tcp(socket, address):
    print 'new tcp connection!'
    while True:
        socket.send('hello\n')
        gevent.sleep(1)

def client_connect(address):
    sockfile = create_connection(address).makefile()
    while True:
        line = sockfile.readline() # returns None on EOF
        if line is not None:
            print "<<<", line,
        else:
            break

app = Service()
app.add_service(StreamServer(('127.0.0.1', 1234), handle_tcp))
app.add_service(WSGIServer(('127.0.0.1', 8080), handle_http))
app.add_service(TcpClient(('127.0.0.1', 1234), client_connect))
app.serve_forever()
```

```
from gservice.core import Service
from gservice.config import Setting

class MyApplication(Service):
    http_port = Setting('http_port')
    tcp_port = Setting('tcp_port')
    connect_address = Setting('connect_address')

    def __init__(self):
        self.add_service(WSGIServer(('127.0.0.1', self.http_port), self.handle_http))
        self.add_service(StreamServer(('127.0.0.1', self.tcp_port), self.handle_tcp))
        self.add_service(TcpClient(self.connect_address, self.client_connect))

    def client_connect(self, address):
        sockfile = create_connection(address).makefile()
        while True:
            line = sockfile.readline() # returns None on EOF
            if line is not None:
                print "<<<", line,
            else:
                break

    def handle_tcp(self, socket, address):
        print 'new tcp connection!'
        while True:
            socket.send('hello\n')
            gevent.sleep(1)

    def handle_http(self, env, start_response):
        start_response('200 OK', [('Content-Type', 'text/html')])
        print 'new http request!'
        return ["hello world"]
```

```
# example.conf.py

pidfile = 'example.pid'
logfile = 'example.log'
http_port = 8080
tcp_port = 1234
connect_address = ('127.0.0.1', 1234)

def service():
    from example import MyApplication
    return MyApplication()
```

```
# Run in the foreground
gservice -C example.conf.py

# Start service as daemon
gservice -C example.conf.py start

# Control service
gservice -C example.conf.py restart
gservice -C example.conf.py reload
gservice -C example.conf.py stop

# Run with overriding configuration
gservice -C example.conf.py -X 'http_port = 7070'
```

Generalizing

gevent proves a model that can be implemented in almost any language that can implement an evented stack

gevent

- Easy ... just normal Python
- Small ... only 25 modules
- Fast ... top performing server
- Compatible ... works with most libraries

Futuristic evented platform for network applications.

Raiden

Lightning fast, scalable messaging

https://github.com/progrium/raiden

Concurrency models

- Traditional multithreading
- Async or Evented I/O
- Actor model

Conclusion

Two very simple, but very powerful tools for distributed / concurrent systems

Thanks

@progrium