Delaware-Barco SE Study Trip Amsterdam 2019

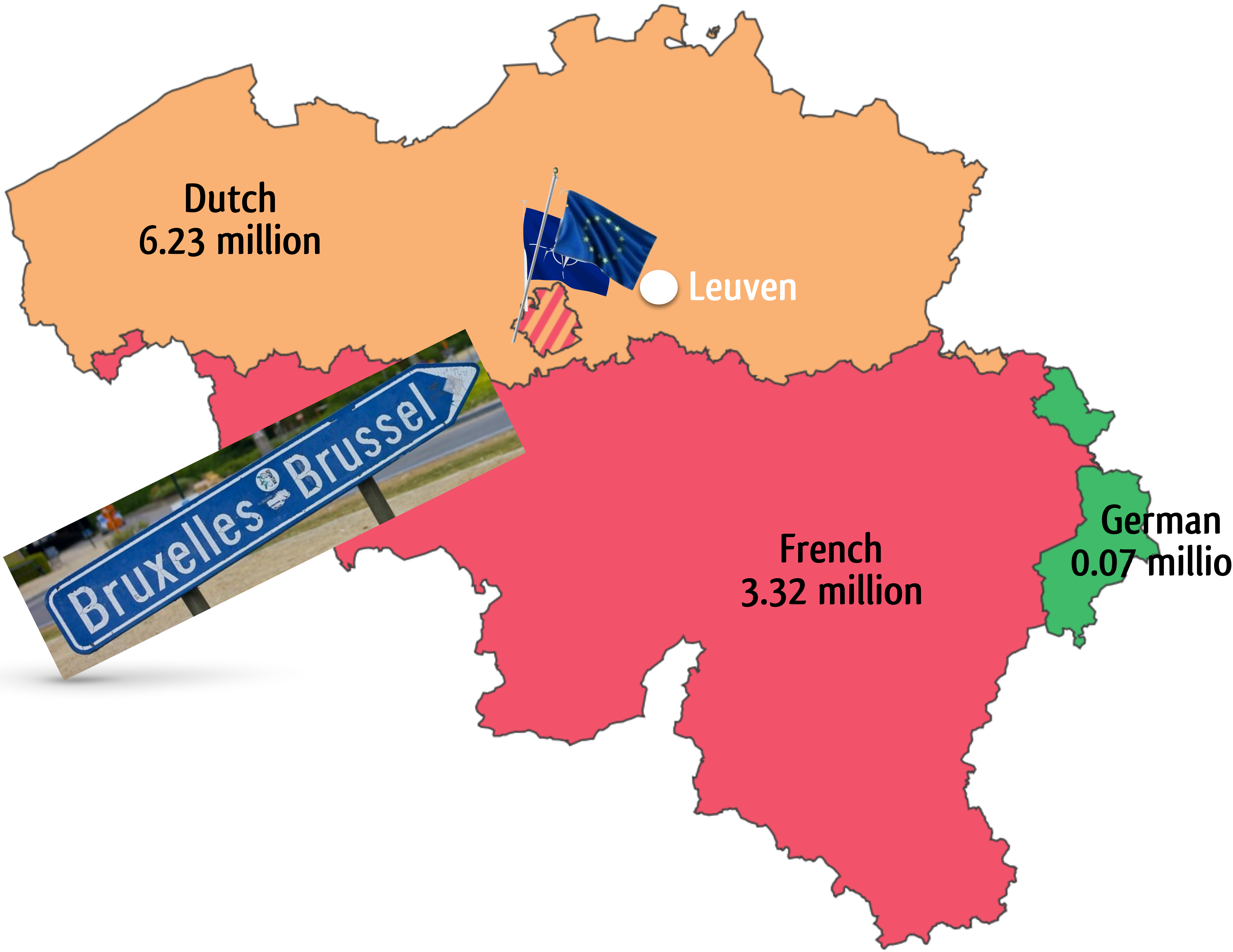
Scaling micro-service architectures up and out: an application perspective

infrastructure perspective
in the afternoon!

Coen De Roover
cderoove@vub.ac.be
http://soft.vub.ac.be/~cderoove/

VRIJE
UNIVERSITEIT
BRUSSEL

Brussels

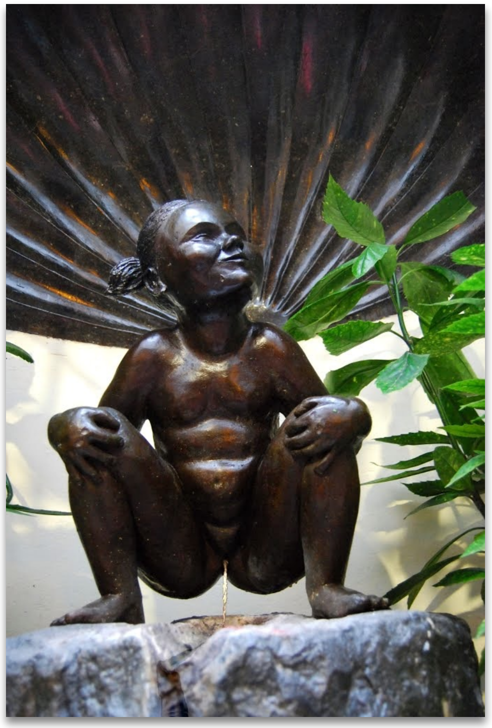

Dutch
6.23 million
Leuven
Bruxelles
Brussel
French
3.32 million
German
0.07 million


Netherlands
Germany
Luxemburg
France

Sights

Grote Markt - Grand Place

Manneken pis



Jeanneke pis

Atomium

Zinneke pis

Food

(

ch Fries

Waffles (from Brussels)

Waffles (from Liège)

Software Languages Lab @ Vrije Universiteit Brussel

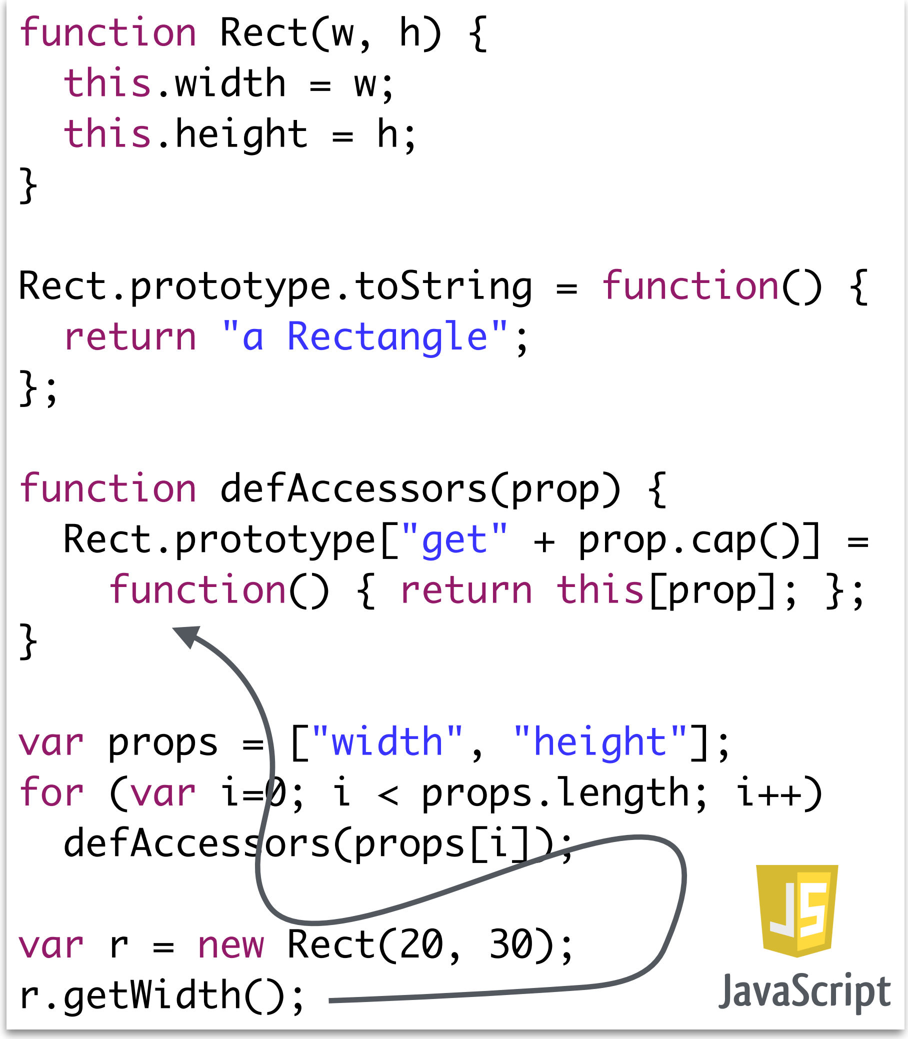
languages

tools

Design, implement, and formalize
novel programming technology that enables
constructing future software systems
in a more economic, more robust,
and more reusable manner.

Program analysis as tool enabler

Which functions are applied at a call site?
Which variables will have the same values?
Which procedures have no observable side effects?
Which expressions can be executed in parallel?



```
function Rect(w, h) {
  this.width = w;
  this.height = h;
}

Rect.prototype.toString = function() {
  return "a Rectangle";
};

function defAccessors(prop) {
  Rect.prototype["get" + prop.cap()] =
    function() { return this[prop]; };
}

var props = ["width", "height"];
for (var i=0; i < props.length; i++)
  defAccessors(props[i]);

var r = new Rect(20, 30);
r.getWidth();
```

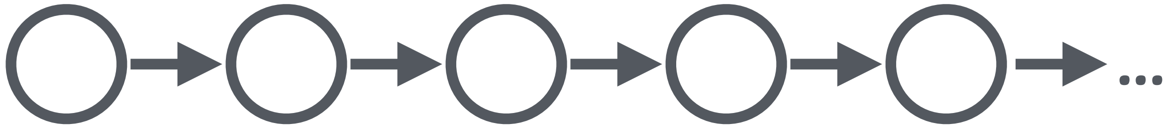
operational semantics encoded as abstract machine

$$
\begin{aligned}
\hat{\varsigma}_{\mathrm{CESK}} \in \hat{\Sigma}_{\mathrm{CESK}} &= \widehat{Control} \times \widehat{Env} \times \widehat{Store} \times \widehat{Addr} \\
\hat{c} \in \widehat{Control} &= Exp + \widehat{Val} \\
\hat{\rho} \in \widehat{Env} &= Var \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Val}) \\
\widehat{val} \in \widehat{Val} &= \widehat{Clo} + \widehat{Kont} \\
\hat{\kappa} \in \widehat{Kont} &::= \mathbf{halt} \mid \mathbf{ar}(e, \hat{\rho}, \hat{a}) \mid \mathbf{fn}(\widehat{clo}, \hat{a}) \\
\widehat{val} \in \widehat{Val} &::= (\lambda v.e) \times \widehat{Env} \\
\hat{a}, \hat{b} \in \widehat{Addr} & \qquad \text{a finite set of addresses}
\end{aligned}
$$

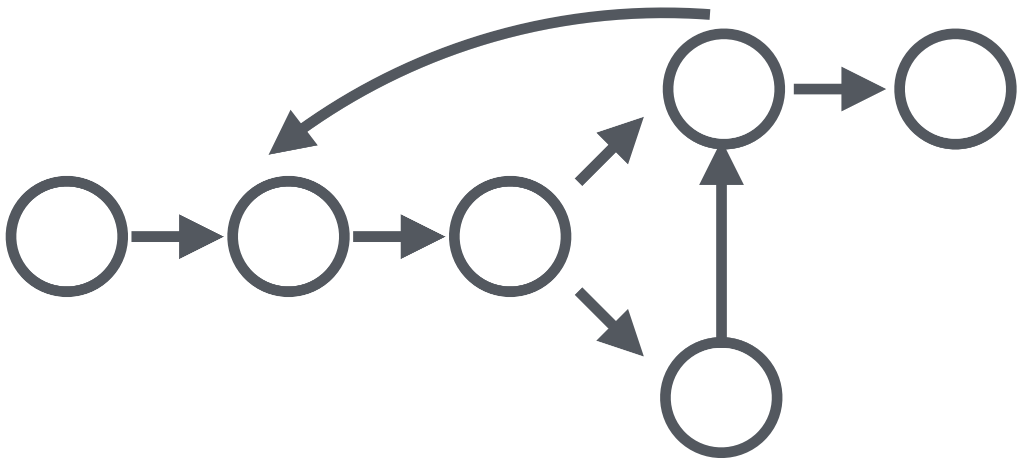
+

$$
\begin{aligned}
\langle v, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle &\mathrel{\widehat{\rightarrow}} \langle \widehat{clo}, \hat{\rho}', \hat{\sigma}, \hat{a}, \hat{u} \rangle \\
&\text{where } \widehat{clo} \in \hat{\sigma}(\hat{\rho}(v)). \\
\langle (\lambda v.e), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle &\rightarrow \langle ((\lambda v.e), \hat{\rho}), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{u} \rangle \\
\langle (e_0\ e_1), \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle &\mathrel{\widehat{\rightarrow}} \langle e_0, \hat{\rho}, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \hat{\rho}, \hat{a})], \hat{b}, \hat{u} \rangle. \\
\langle \widehat{clo}, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle &\mathrel{\widehat{\rightarrow}} \langle e, \hat{\rho}', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(\widehat{clo}, \hat{a}')], \hat{b}, \hat{u} \rangle \text{ if } \hat{\kappa} = \mathbf{ar}(e, \hat{\rho}', \hat{a}'), \\
\langle val, \hat{\rho}, \hat{\sigma}, \hat{a}, \hat{t} \rangle &\mathrel{\widehat{\rightarrow}} \langle e, \hat{\rho}'[v \mapsto \hat{b}], \hat{\sigma} \sqcup [\hat{b} \mapsto val], \hat{a}', \hat{u} \rangle \text{ if } \hat{\kappa} = \mathbf{fn}(((\lambda v.e), \hat{\rho}'), \hat{a}').
\end{aligned}
$$

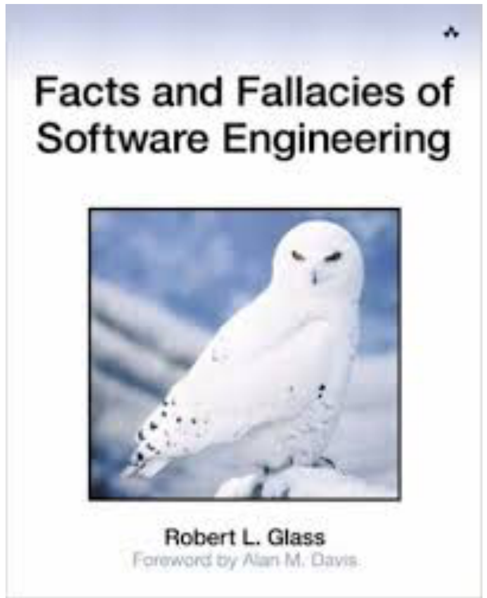
when configured for concrete interpretation



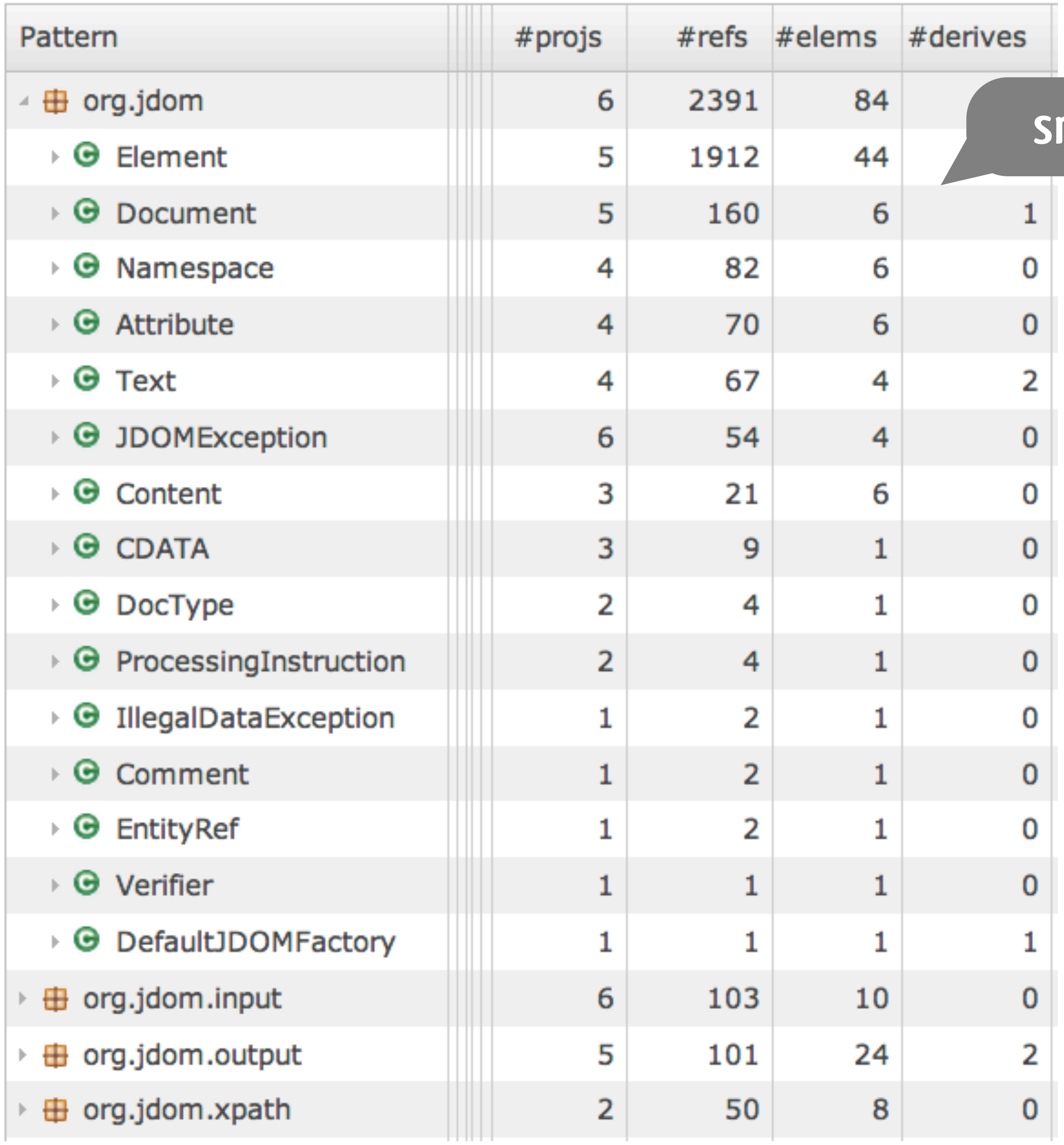
when configured for abstract interpretation



Repository analysis for evidence-based SE



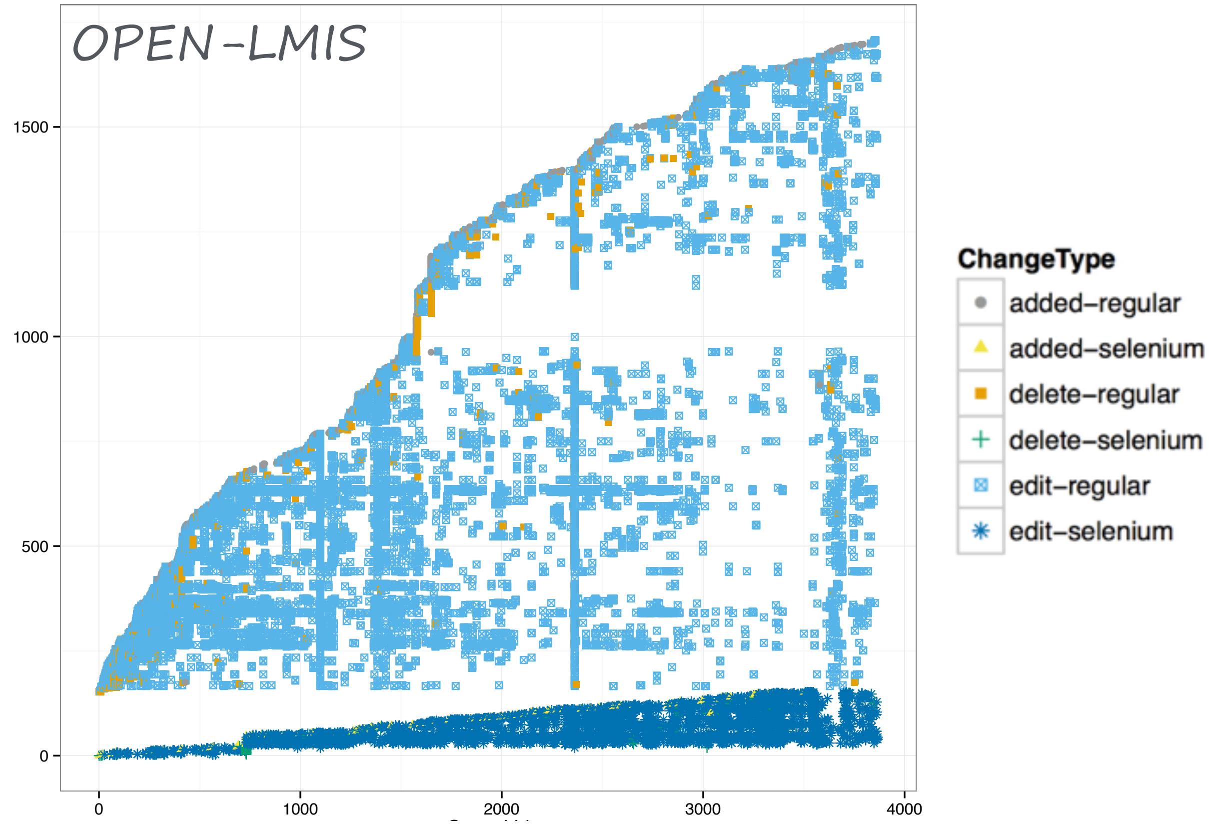
How to make informed decisions about a project?
What can we learn from existing project repositories?



Pattern	#projs	#refs	#elems	#derives
org.jdom	6	2391	84	
Element	5	1912	44	
Document	5	160	6	1
Namespace	4	82	6	0
Attribute	4	70	6	0
Text	4	67	4	2
JDOMException	6	54	4	0
Content	3	21	6	0
CDATA	3	9	1	0
DocType	2	4	1	0
ProcessingInstruction	2	4	1	0
IllegalDataException	1	2	1	0
Comment	1	2	1	0
EntityRef	1	2	1	0
Verifier	1	1	1	0
DefaultJDOMFactory	1	1	1	1
org.jdom.input	6	103	10	0
org.jdom.output	5	101	24	2
org.jdom.xpath	2	50	8	0

snapshot mining

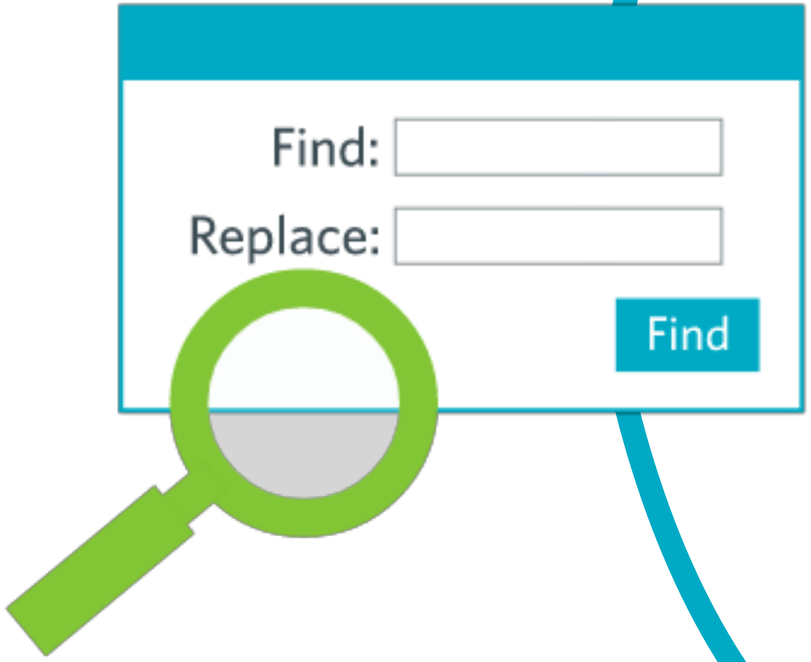
history mining



How much of a library is used in the wild?
How often are libraries subclassed?

Are test scripts abandoned over time or are they maintained as the application evolves?

Program transformation for automating changes

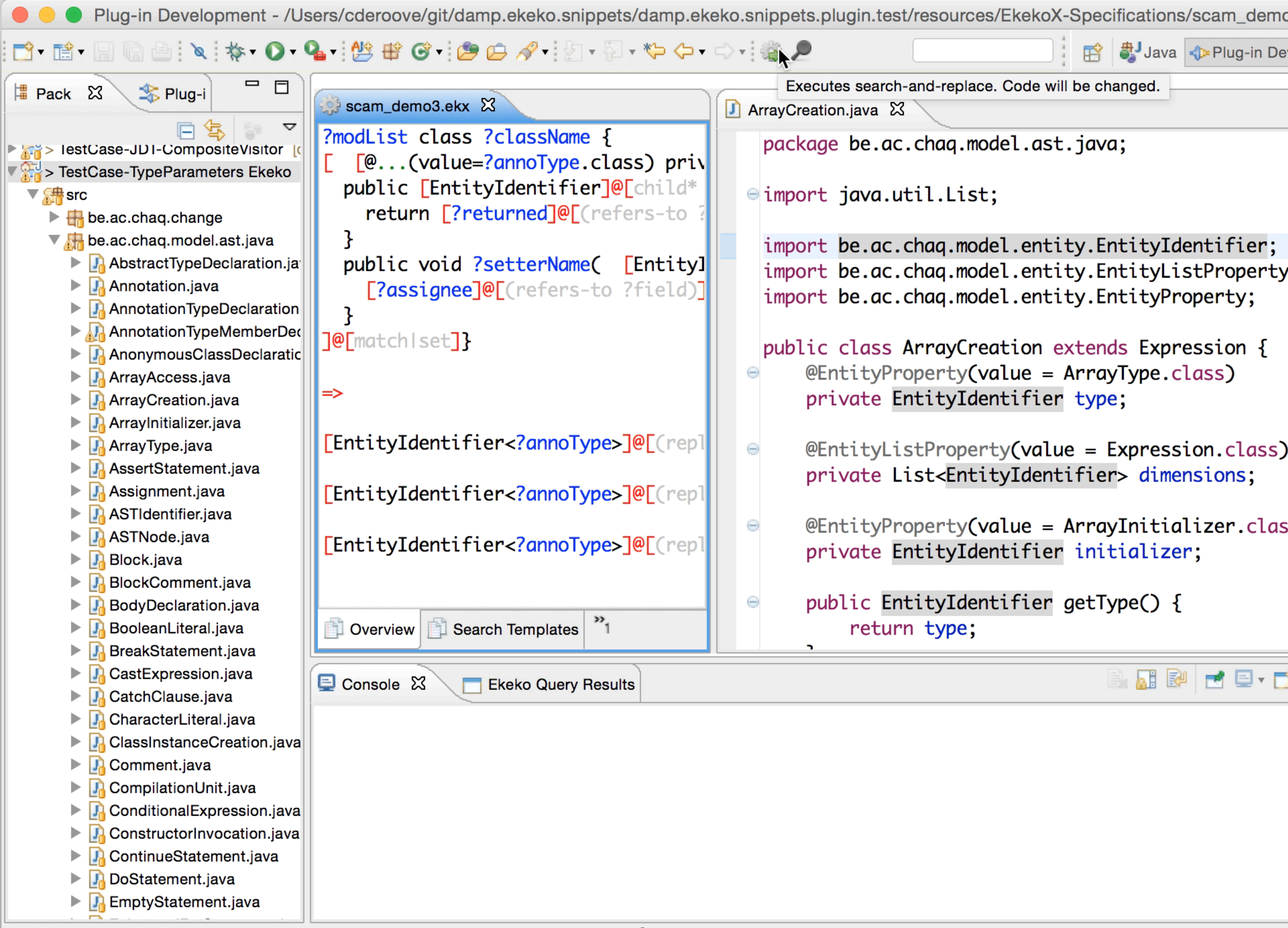


```
public class BreakStatement extends Statement {
    @EntityProperty(value = SimpleName.class)
    private EntityIdentifier label;

    public EntityIdentifier getLabel() {
        return label;
    }

    public void setLabel(EntityIdentifier label) {
        this.label = label;
    }
}
```

value of annotation

```
public class BreakStatement extends Statement {
    @EntityProperty(value = SimpleName.class)
    private EntityIdentifier<SimpleName> label;

    public EntityIdentifier<SimpleName> getLabel() {
        return label;
    }

    public void setLabel(EntityIdentifier<SimpleName> label) {
        this.label = label;
    }
}
```

to be used as type parameter



Executes search-and-replace. Code will be changed.

```
?modList class ?className {
[   [@...(value=?annoType.class) priv
  public [EntityIdentifier]@[child*
    return [?returned]@[(refers-to ?
  }
  public void ?setterName(  [Entity]
    [?assignee]@[(refers-to ?field)]
  }
]@[match|set]}


=>


[EntityIdentifier<?annoType>]@[(repl

[EntityIdentifier<?annoType>]@[(repl

[EntityIdentifier<?annoType>]@[(repl
```

```
package be.ac.chaq.model.ast.java;

import java.util.List;

import be.ac.chaq.model.entity.EntityIdentifier;
import be.ac.chaq.model.entity.EntityListProperty
import be.ac.chaq.model.entity.EntityProperty;

public class ArrayCreation extends Expression {
    @EntityProperty(value = ArrayType.class)
    private EntityIdentifier type;

    @EntityListProperty(value = Expression.class)
    private List<EntityIdentifier> dimensions;

    @EntityProperty(value = ArrayInitializer.clas
    private EntityIdentifier initializer;

    public EntityIdentifier getType() {
        return type;
```

Service-oriented architecture

introduction and motivation

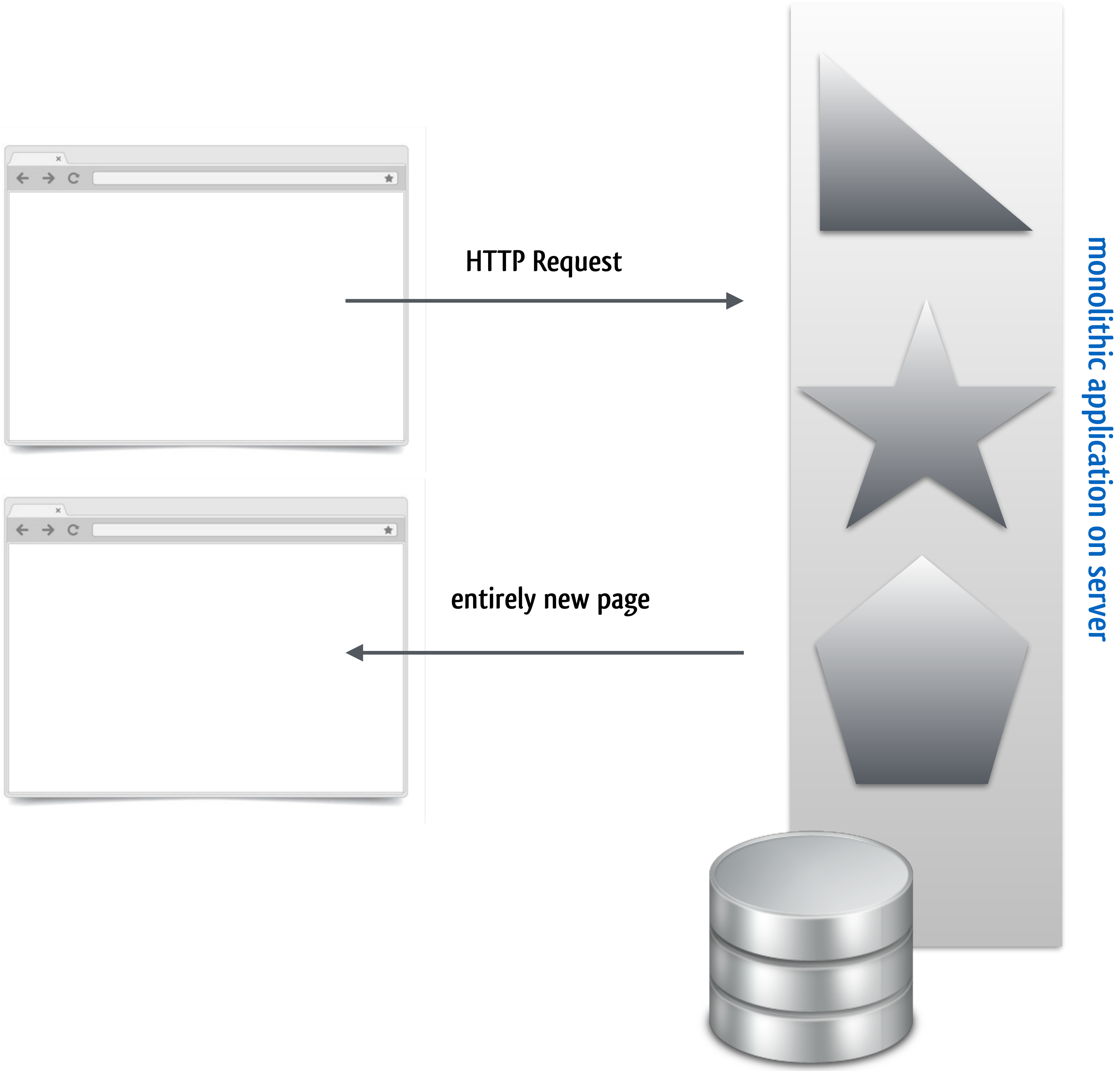
Scaling up

using concurrent actors

Scaling out

using distributed actors

Evolution of web application architectures



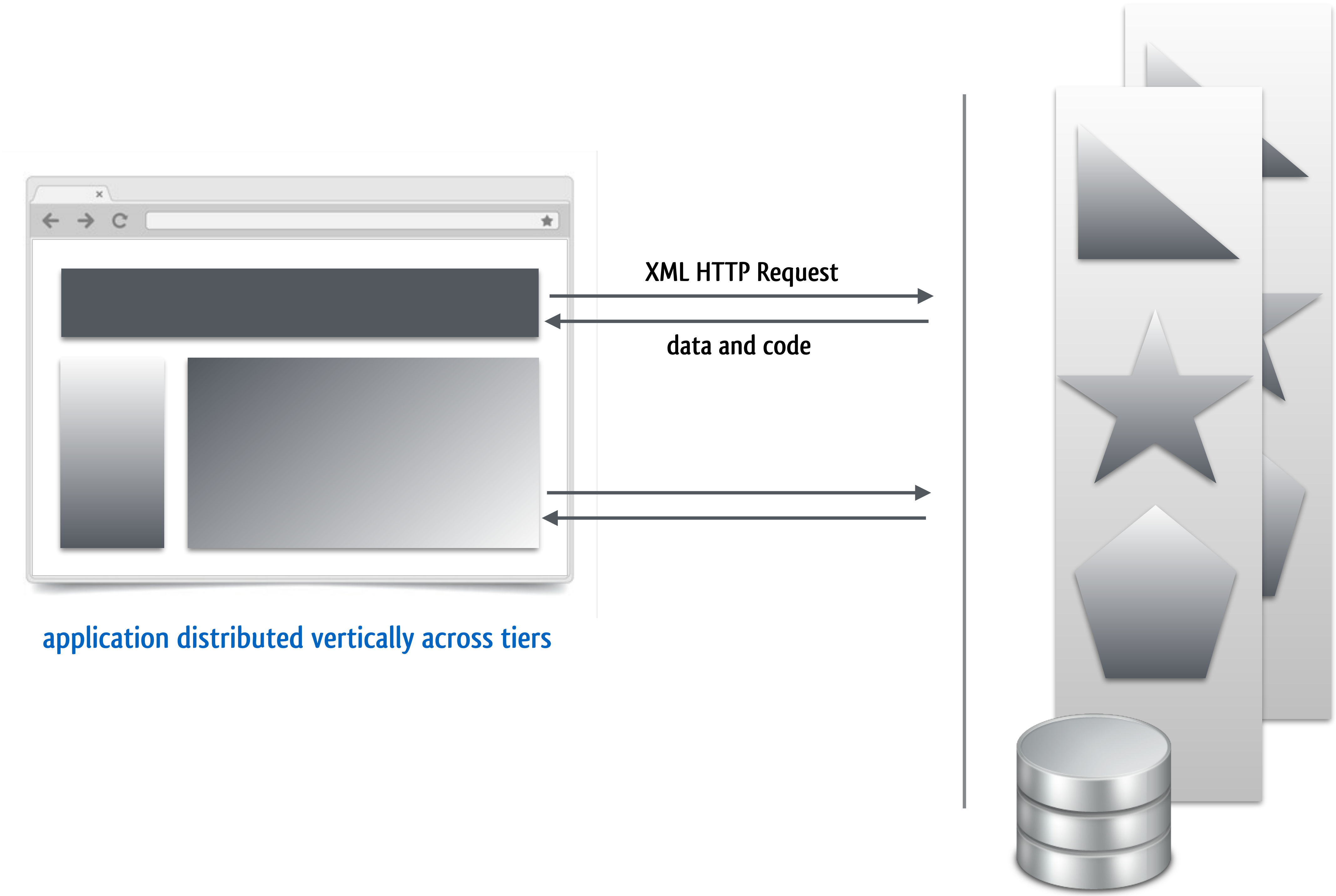
HTTP Request

entirely new page

monolithic application on server

multi-page application

Evolution of web application architectures



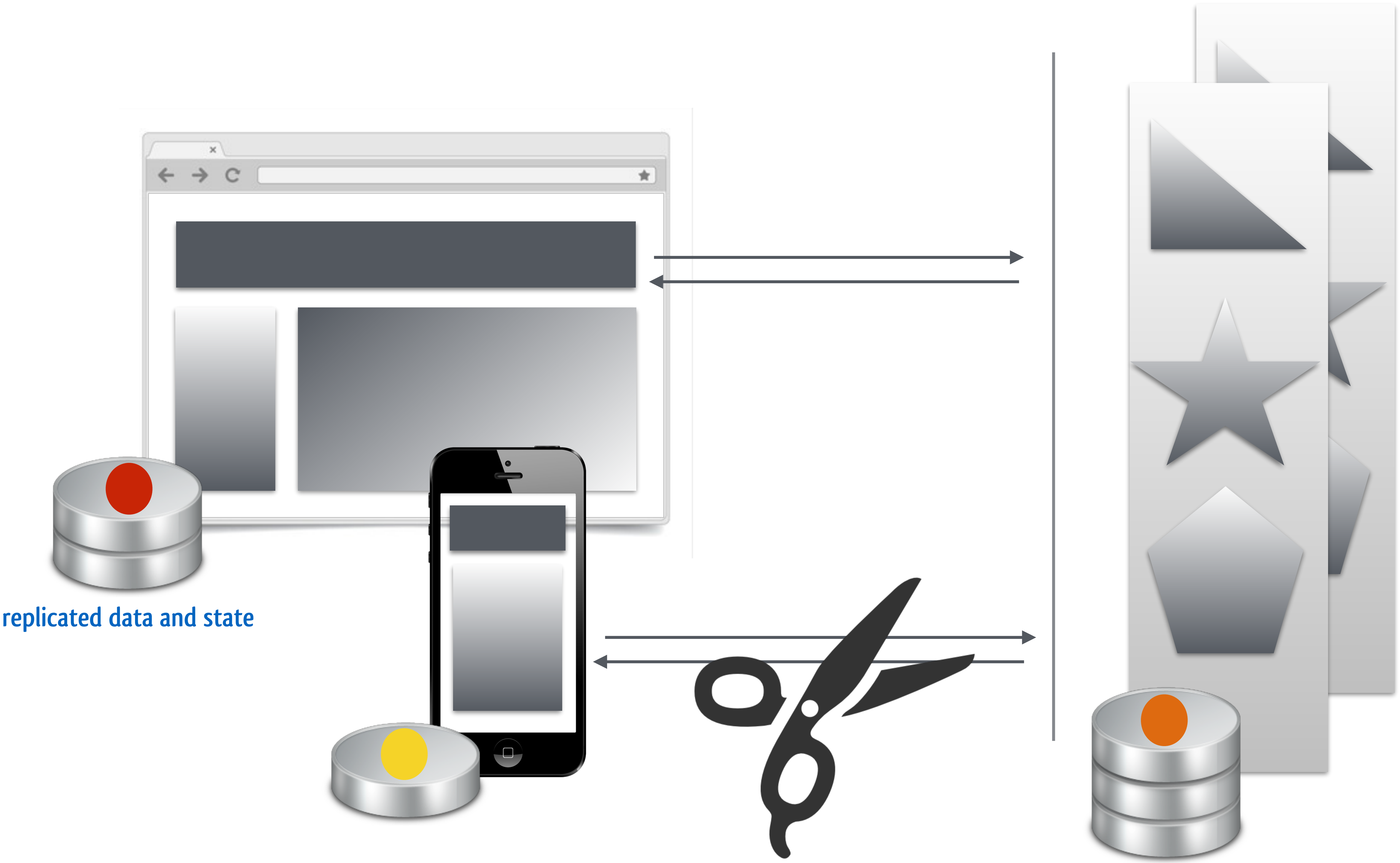
XML HTTP Request

data and code

application distributed vertically across tiers

single-page application

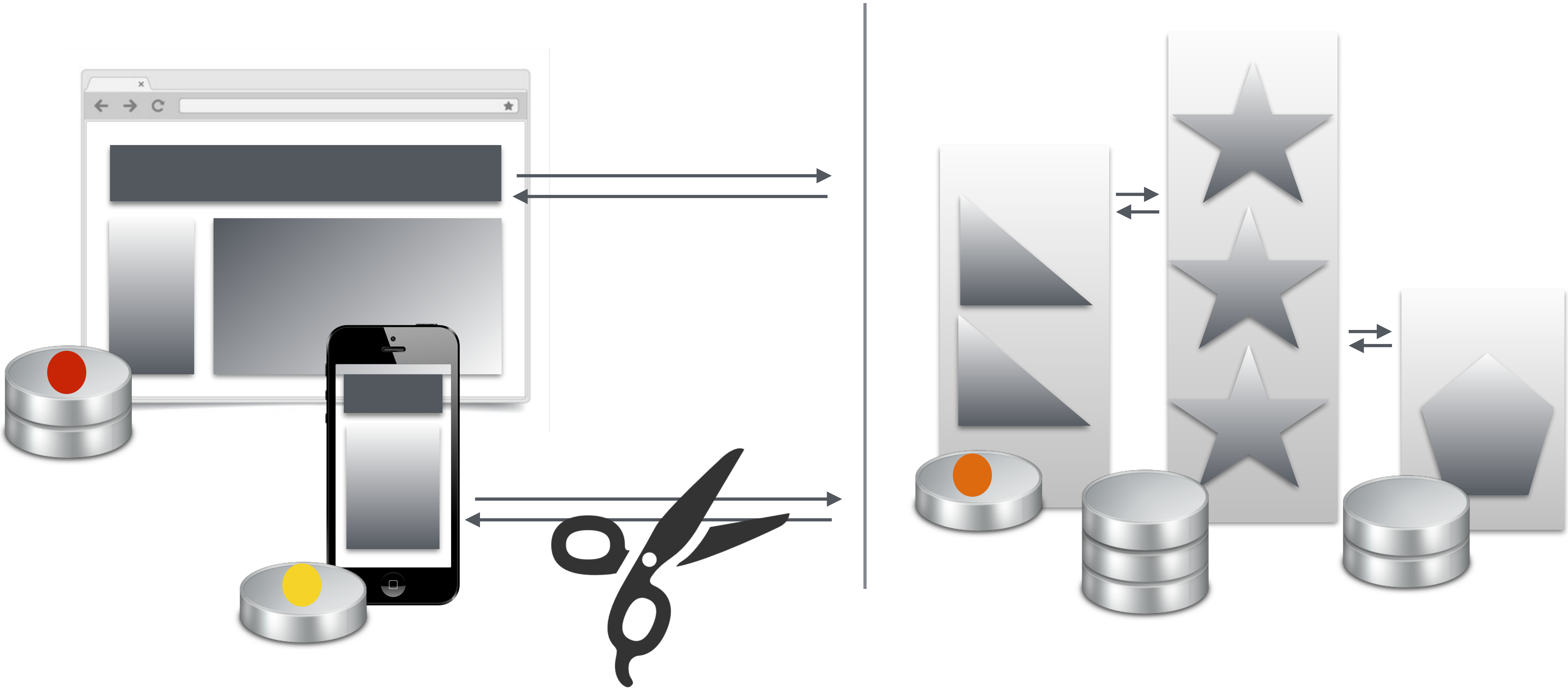
Evolution of web application architectures



replicated data and state

rich internet application

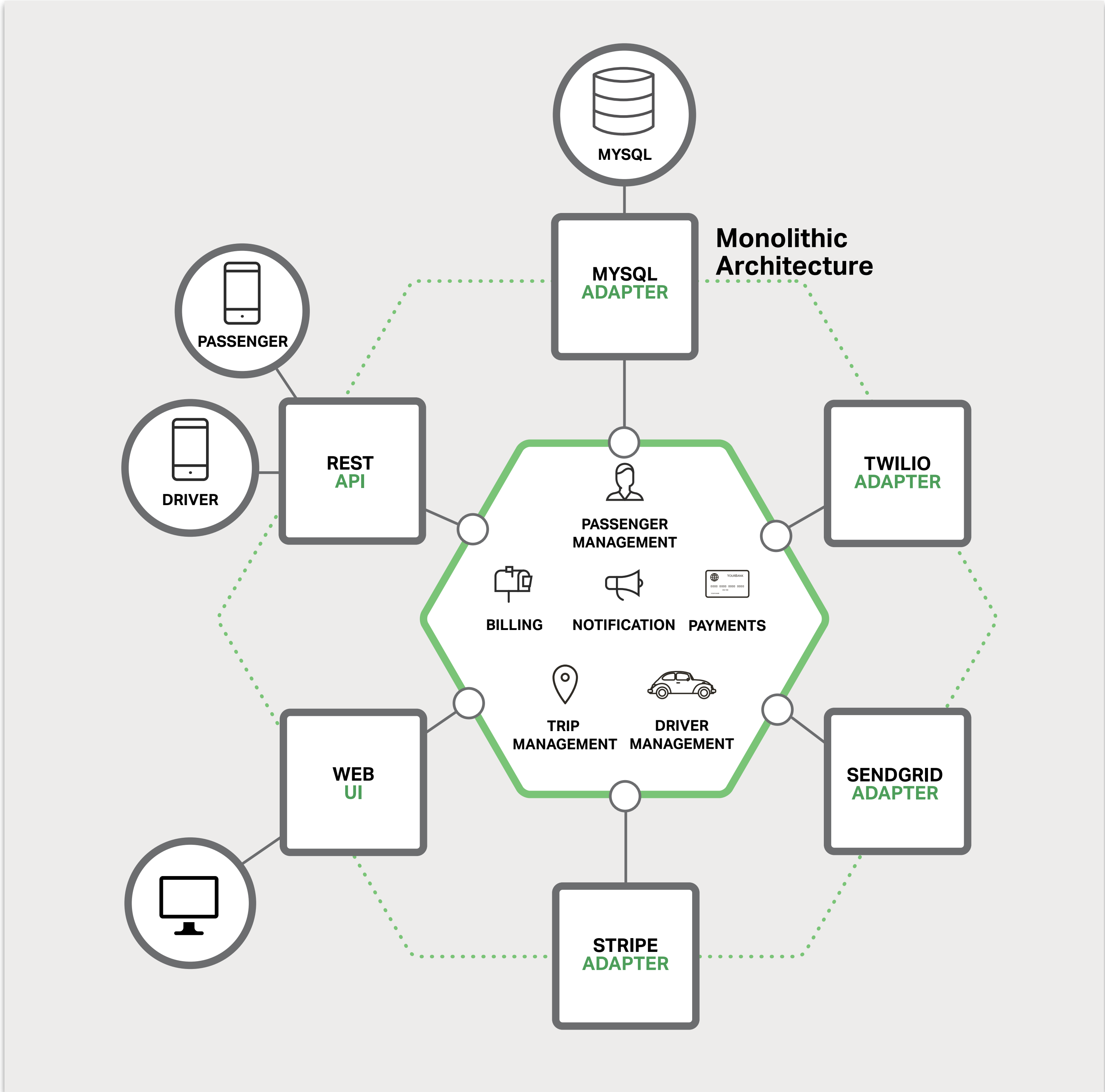
Evolution of web application architectures

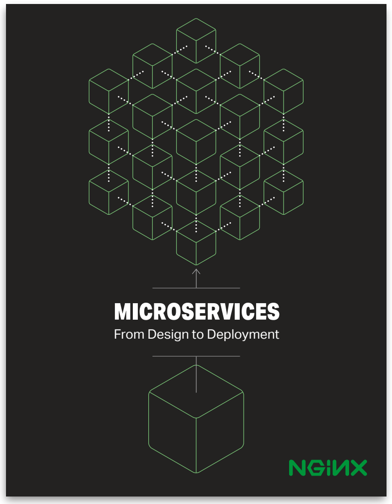


application distributed horizontally
between instances of the same tier

µ-services on server tier

Beyond web applications: Taxi platform

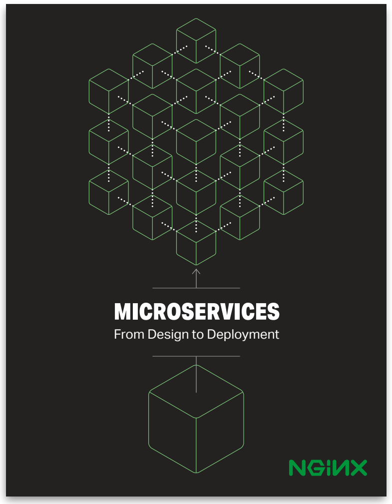




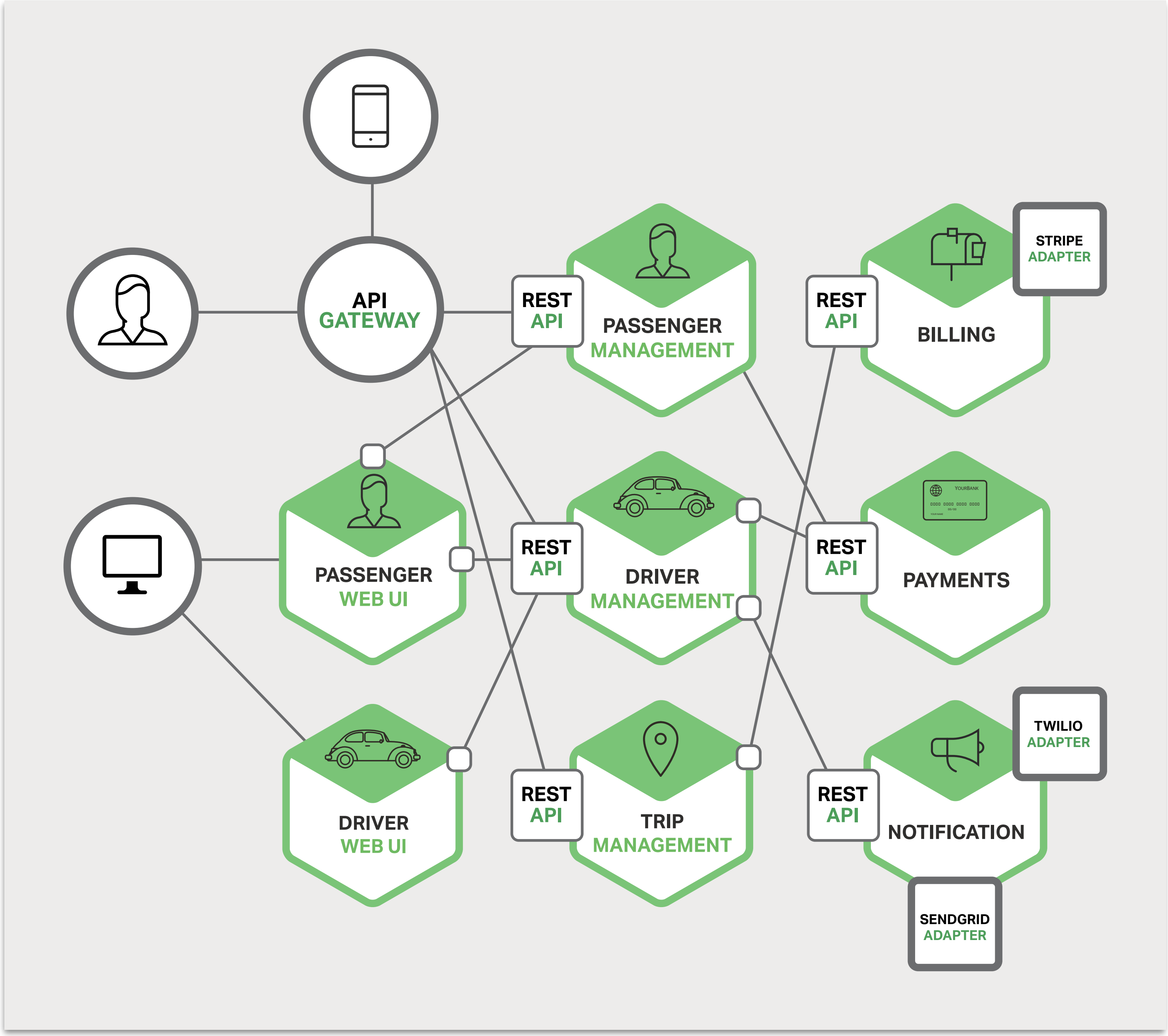
[Richardson 2016]

- one large, but modular application
- needs to be redeployed entirely upon smallest change
- difficult to accommodate components with different resource requirements

Taxi platform: decomposition in services



[Richardson 2016]


API GATEWAY

REST API

PASSENGER MANAGEMENT

REST API

BILLING

STRIPE ADAPTER

PASSENGER WEB UI

REST API

DRIVER MANAGEMENT

REST API

PAYMENTS

DRIVER WEB UI

REST API

TRIP MANAGEMENT

REST API

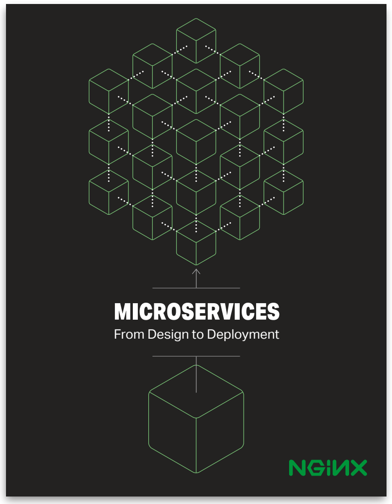
NOTIFICATION

TWILIO ADAPTER

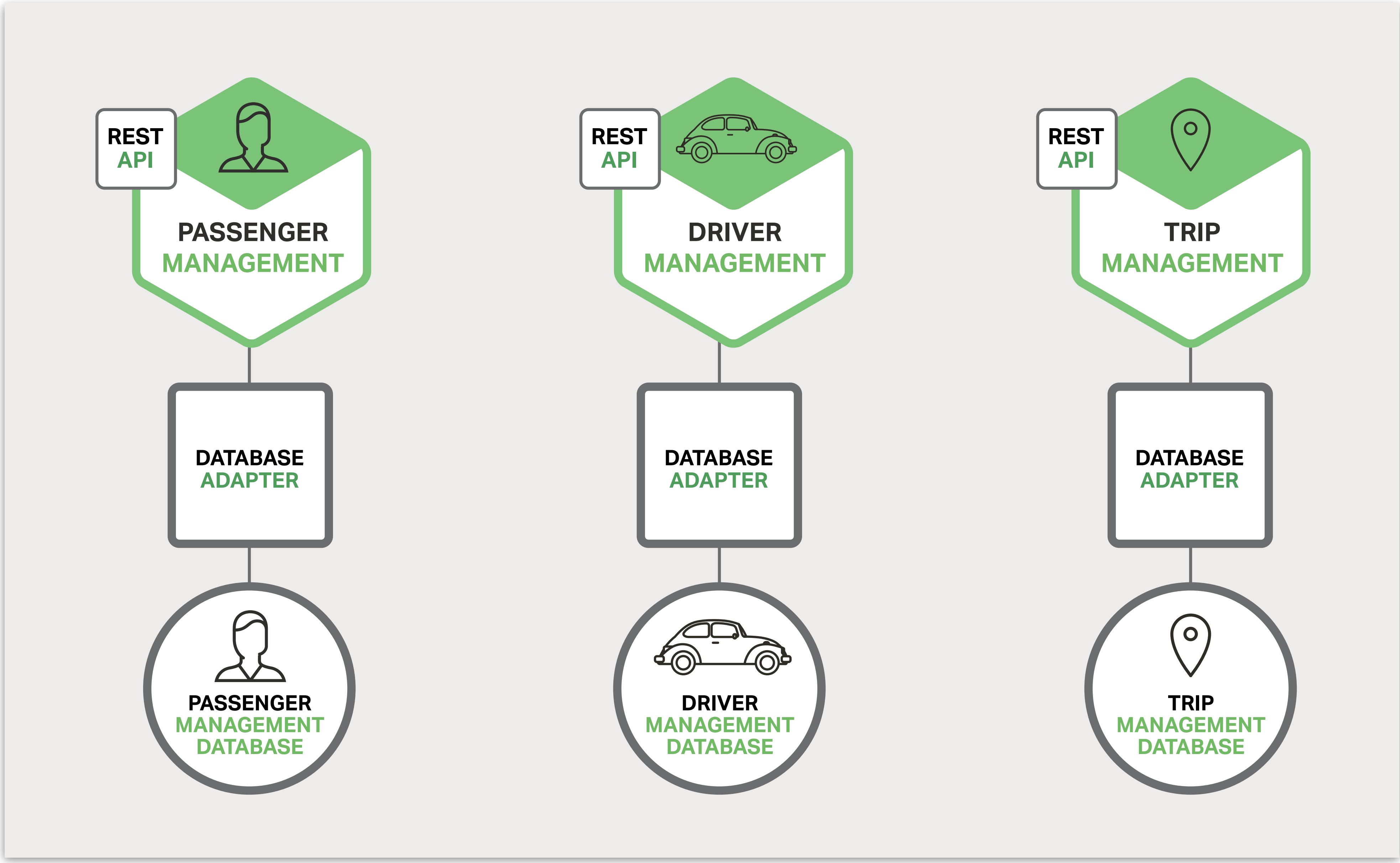
SENDGRID ADAPTER

- monolith distributed vertically into services that are deployed independently (scaling up)
- each service provides and consumes functionality as a mini-application on its own

Taxi platform: decomposition in services



[Richardson 2016]



REST API

PASSENGER MANAGEMENT

DATABASE ADAPTER

PASSENGER MANAGEMENT DATABASE

REST API

DRIVER MANAGEMENT

DATABASE ADAPTER

DRIVER MANAGEMENT DATABASE

REST API

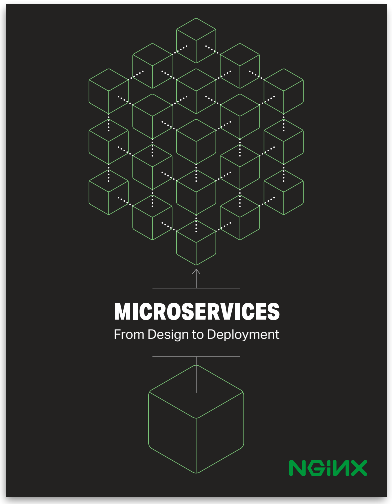
TRIP MANAGEMENT

DATABASE ADAPTER

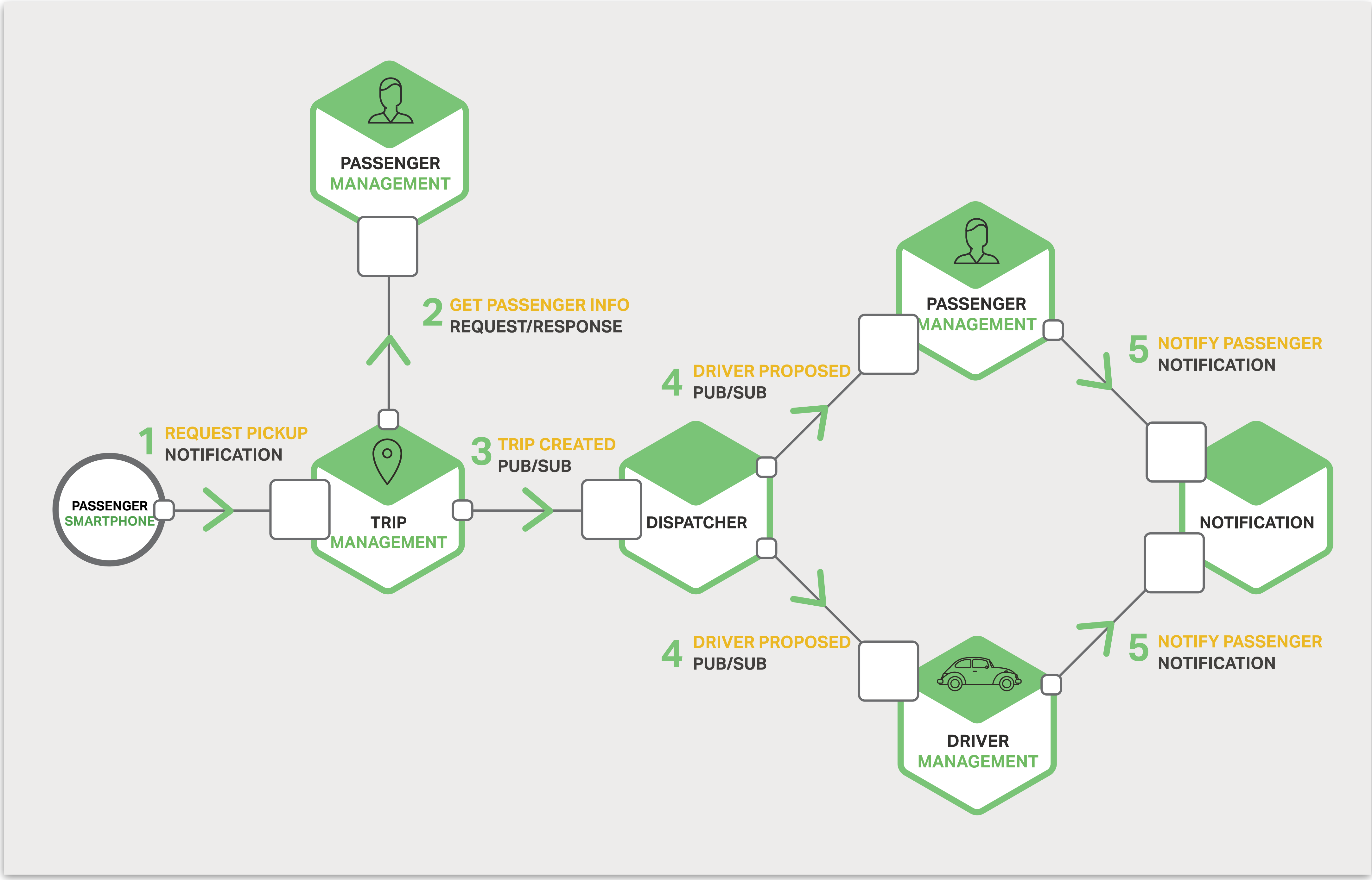
TRIP MANAGEMENT DATABASE

- every service owns its own data, ensuring loose coupling
- freedom to choose database that best suits its needs (e.g., geo-queries)
- but challenge of distributed data management:
  ensuring consistency of updates that span databases

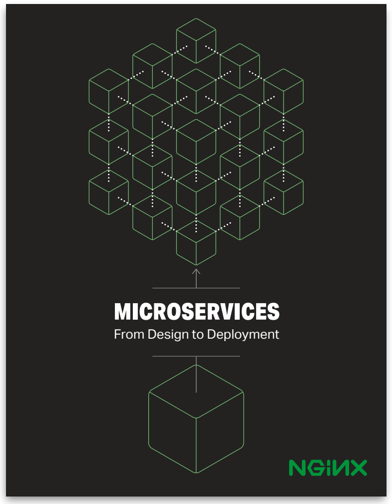
Taxi platform: inter-process communication



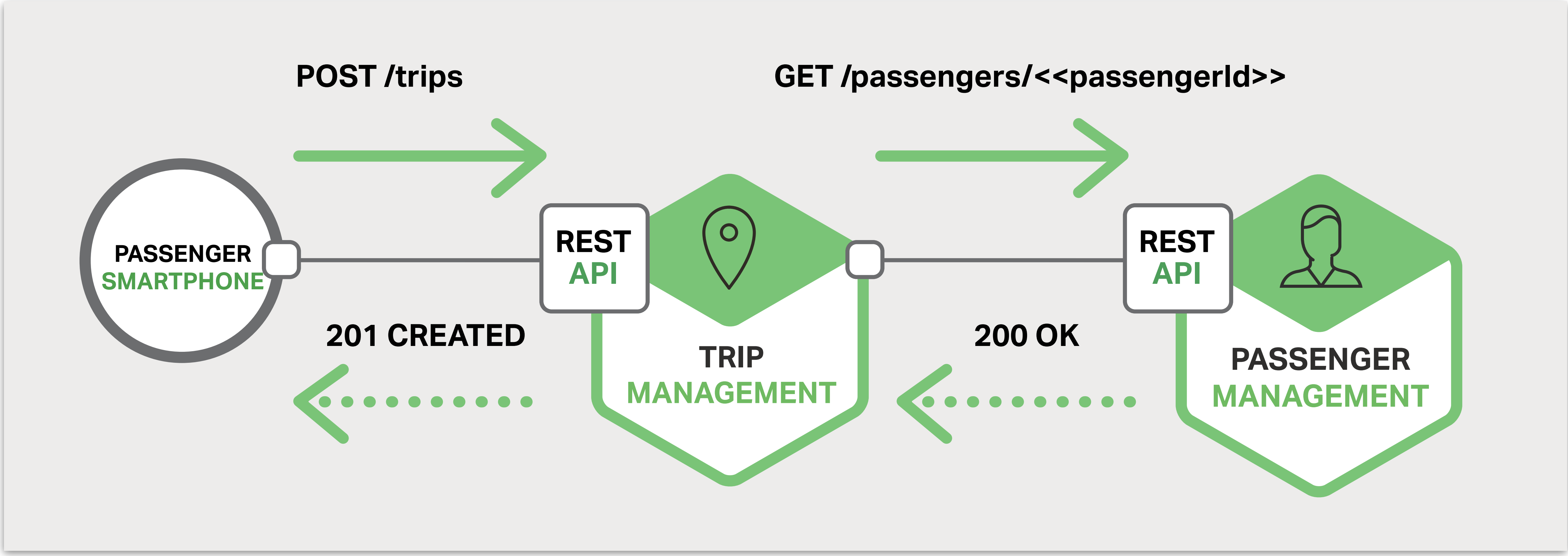
[Richardson 2016]



Inter-process communication: REST

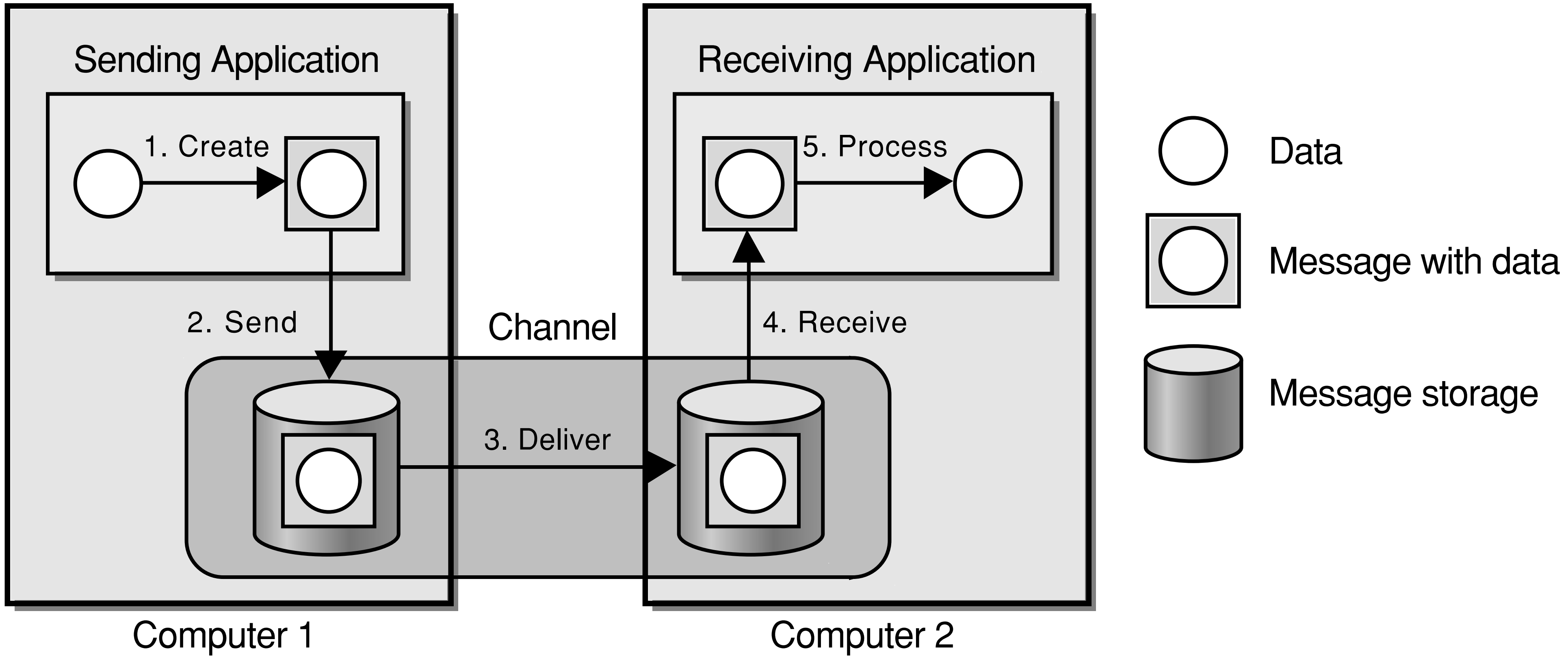


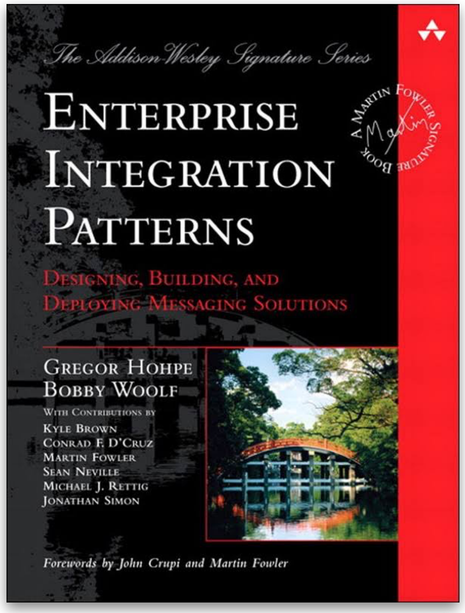
[Richardson 2016]



- simple and familiar, synchronous request/response cycle of HTTP
- not prone to fallacy of transparent distribution
- exposes business objects as resources at a URI
- four primary HTTP operations on those resources: POST, GET, PUT, DELET

Inter-process communication: async messaging




The Addison-Wesley Signature Series

ENTERPRISE INTEGRATION PATTERNS

DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS

GREGOR HOHPE
BOBBY WOOLF
WITH CONTRIBUTIONS BY
KYLE BROWN
CONRAD F. D'CRUZ
MARTIN FOWLER
SEAN NEVILLE
MICHAEL J. RETTIG
JONATHAN SIMON

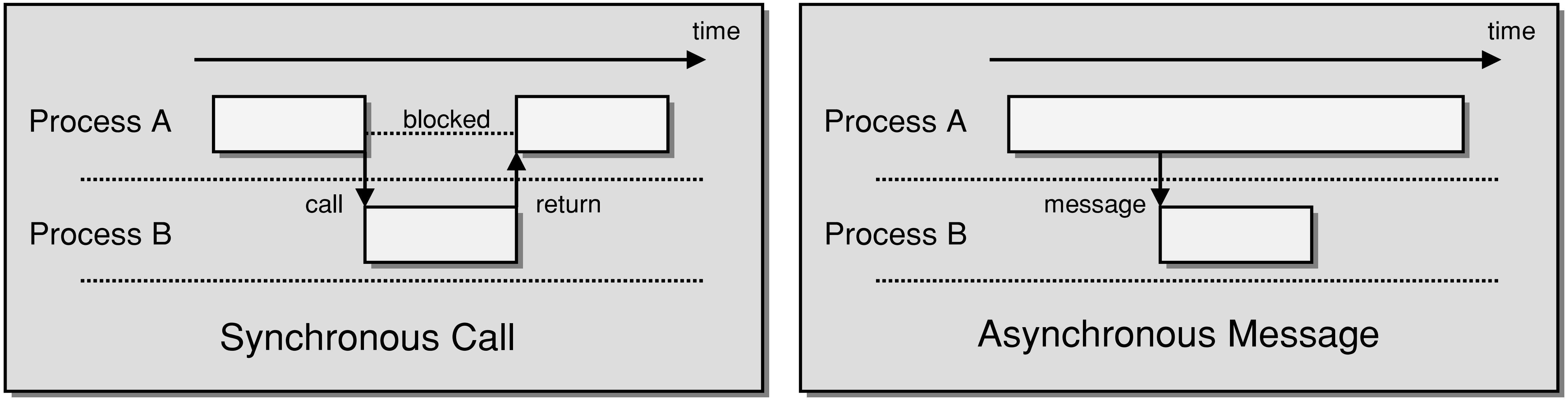
Forewords by John Crupi and Martin Fowler

A MARTIN FOWLER SIGNATURE BOOK

[Hophe et al., 2003]

- Create
  - The sender creates the message and populates it with data.
- Send
  - The sender adds the message to a channel.
- Deliver
  - The messaging system moves the message from the sender’s process to the receiver’s process, making it available to the receiver.
- Receive
  - The receiver reads the message from the channel.
- Process
  - The receiver extracts the data from the message.

Asynchronous messaging advantages



- Asynchronicity
  - Messaging enables a send-and-forget approach to communication.
  - The sender does not have to wait for the receiver to receive and process the message.
  - Once a message has been stored in the communication channel, the sender is free to perform other work while the message is transmitted and eventually processed in the background.
- Variable Timing
  - The messaging system queues up requests until the receiver is ready to process them.
  - Asynchronous messaging allows the sender to submit requests to the receiver at its own pace and the receiver to consume the requests at its own different pace.

Asynchronous messaging disadvantages

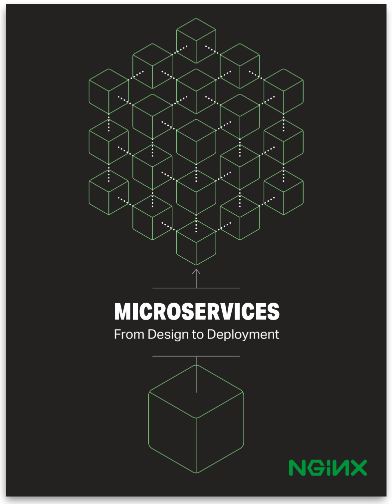
unfortunately, (distributed) communication is inherently unreliable
delivery of a message requires eventual availability of channel and recipient

rendering messages first-class entities enables implementing delivery guarantees:

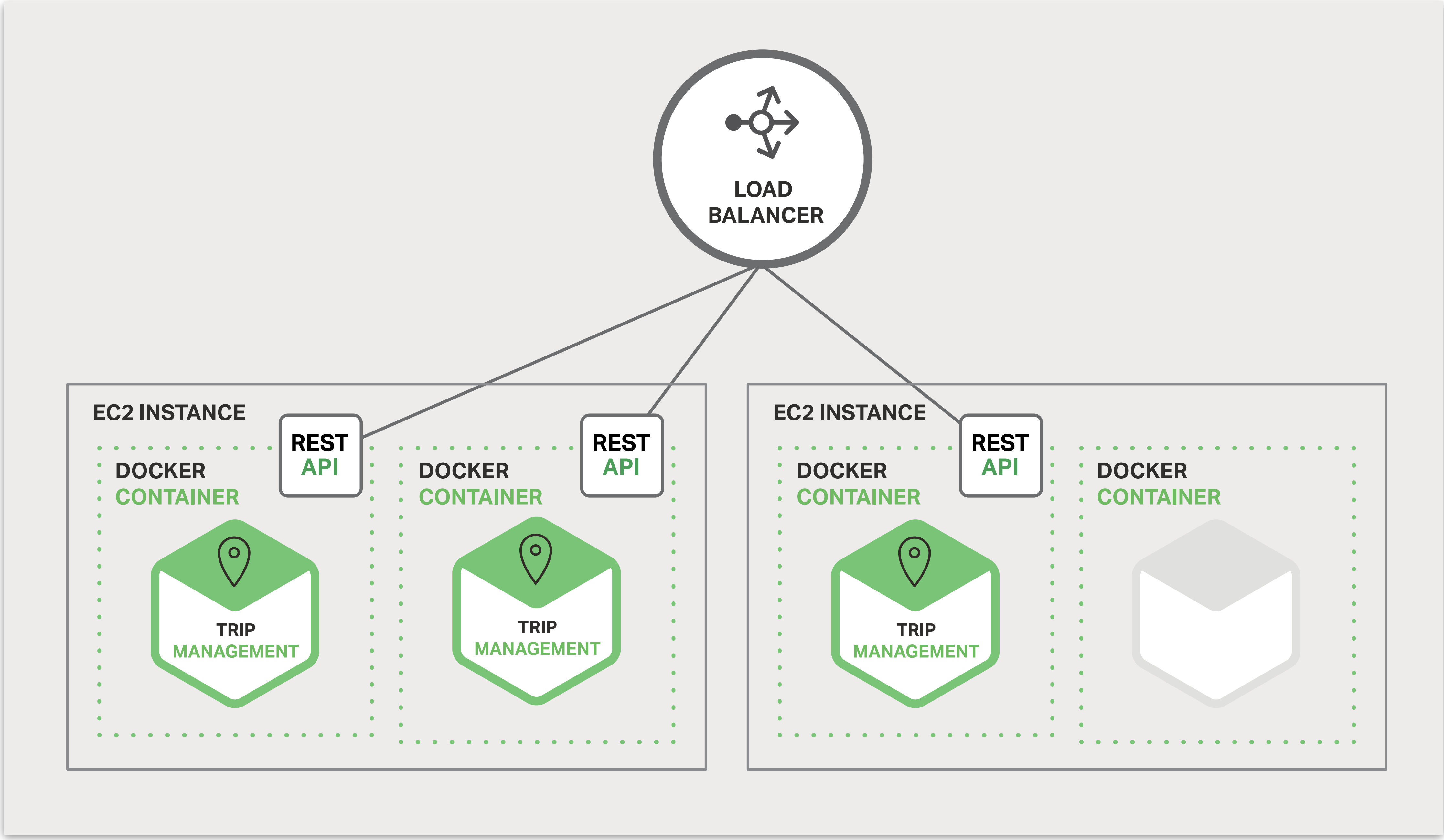
- at-most-once delivery:
  - no state required at sender nor receiver, a message sent once will either arrive or not
  - message will be delivered [0,1] times
- at-least-once:
  - keep state at the sender to ensure that a message will be resent until it has been acknowledged by the recipient
  - message will be delivered [1,∞] times as the acknowledgement message might be lost
- exactly-once:
  - as above, with additional state at the receiver to make sure only the first of the same messages will be processed
  - message will be delivered exactly 1 time
    (under the assumption of eventual availability of channel and recipient)

NOTE: as a recipient might fail while processing a message, reliability can only be guaranteed by application-level acknowledgements of message processing, it does not suffice for the messaging system to acknowledge putting the message in the recipients' mailbox

Taxi platform: containerisation

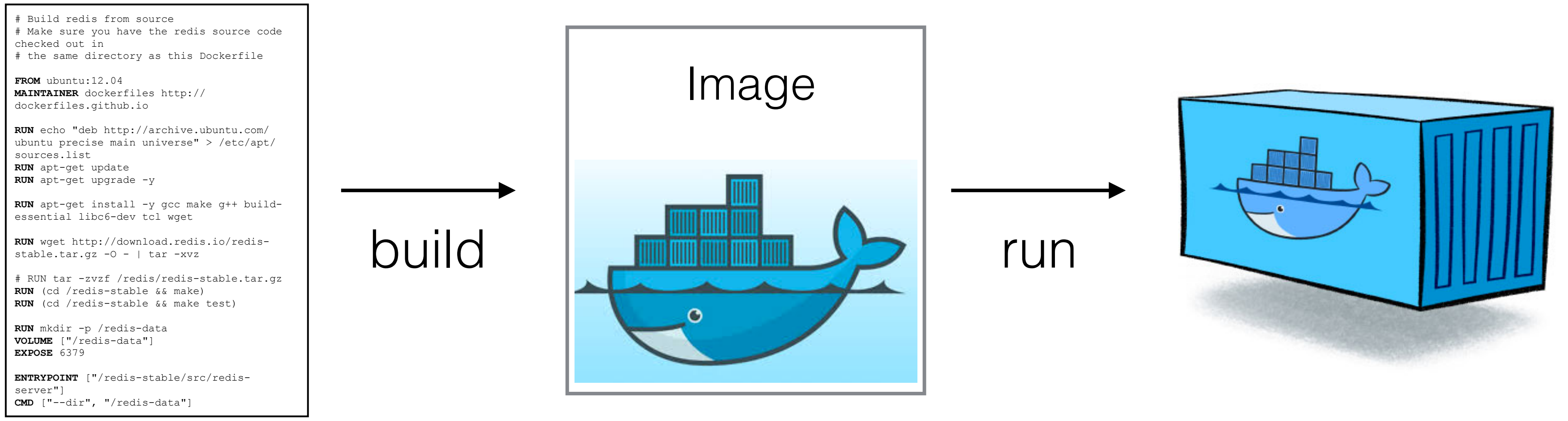


[Richardson 2016]



- individual service can be replicated horizontally (scaling out), often behind load balancer
- services run in containers (e.g., Docker) that can be provisioned and spun up fast
- containers can be orchestrated (e.g., Kubernetes)

Infrastructure-as-code



```
# Build redis from source
# Make sure you have the redis source code
checked out in
# the same directory as this Dockerfile

FROM ubuntu:12.04
MAINTAINER dockerfiles http://
dockerfiles.github.io

RUN echo "deb http://archive.ubuntu.com/
ubuntu precise main universe" > /etc/apt/
sources.list
RUN apt-get update
RUN apt-get upgrade -y

RUN apt-get install -y gcc make g++ build-
essential libc6-dev tcl wget

RUN wget http://download.redis.io/redis-
stable.tar.gz -O - | tar -xvz

# RUN tar -zvzf /redis/redis-stable.tar.gz
RUN (cd /redis-stable && make)
RUN (cd /redis-stable && make test)

RUN mkdir -p /redis-data
VOLUME ["/redis-data"]
EXPOSE 6379

ENTRYPOINT ["/redis-stable/src/redis-
server"]
CMD ["--dir", "/redis-data"]
```

Dockerfile

Docker Image

Docker Container

[Cito et al., MSR17]

Base Image

Dependencies

Install

Volume
Open Port

Start Server

```
FROM ubuntu:12.04
MAINTAINER John Doe

RUN echo "deb http://archive.ubuntu.com/ubuntu precise main universe" > /etc/apt/
sources.list
RUN apt-get update
RUN apt-get upgrade -y

RUN apt-get install -y gcc make g++ build-essential libc6-dev tcl wget
RUN sudo -E pip install scipy:0.18.1

# RUN tar -zvzf /redis/redis-stable.tar.gz
RUN (cd /redis-stable && make)
RUN (cd /redis-stable && make test)

ADD redis.conf /var/www/redis.conf

RUN mkdir -p /redis-data
VOLUME ["/redis-data"]
EXPOSE 6379

ENTRYPOINT ["/redis-stable/src/redis-server"]
CMD ["--dir", "/redis-data"]
```

Base Image can be an OS (Ubuntu)
or a different, existing image

Runs commands as if you were typing
them in the command line

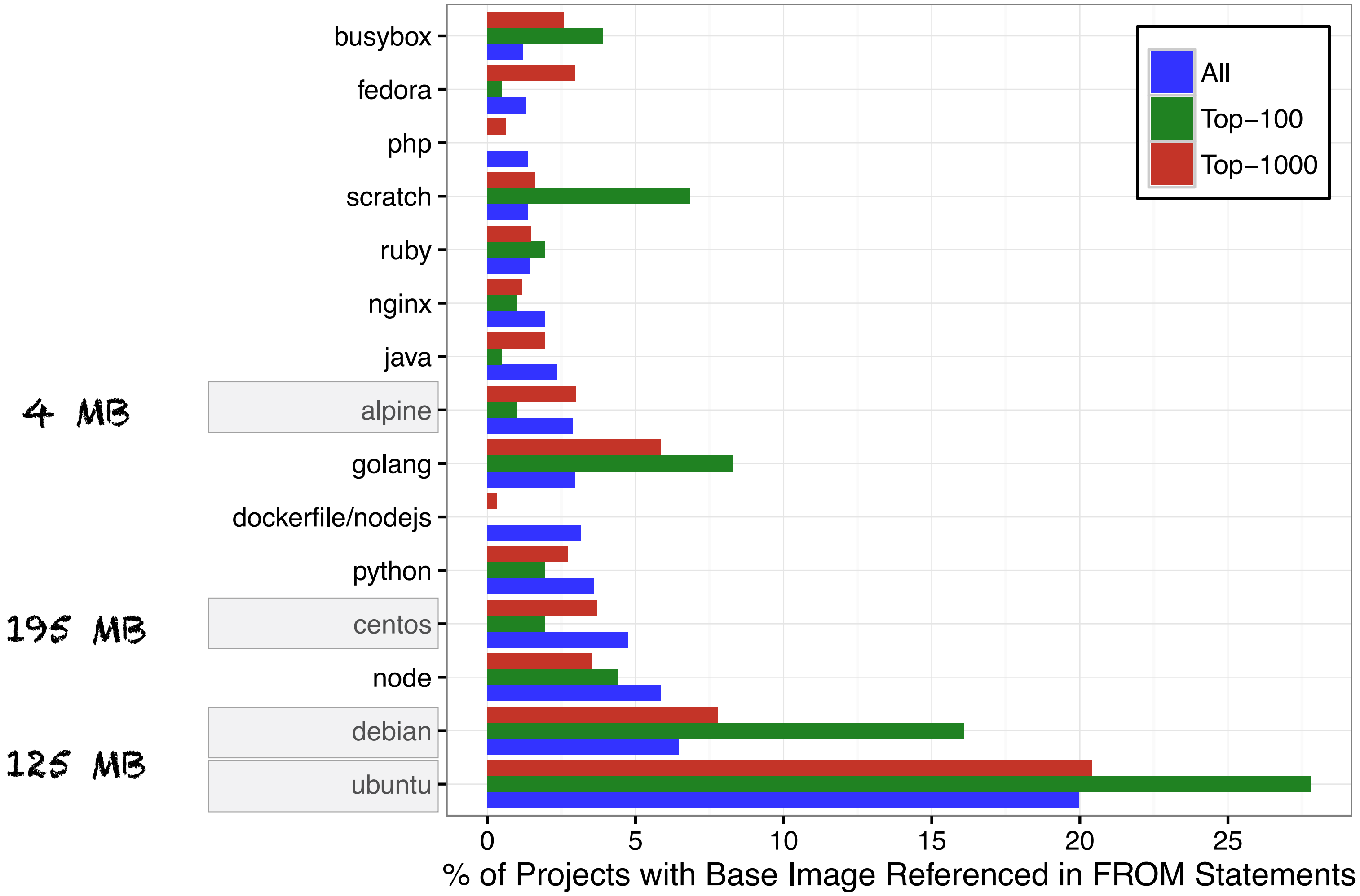
Copies local files from
build context into container

Infrastructure-as-code

Research finding

[Cito et al., MSR17]

Base Images & Sizes



- ⬇ Reduce Image Size
- 👍 Base Image Recommendation

Infrastructure-as-code

[Cito et al., MSR17]

Distribution of Instructions

Instruction	All	Top-1000	Top-100
RUN	40%	41%	48%
COMMENT	16%	14%	15%
ENV	6%	7%	9%
FROM	7%	8%	7%
ADD	6%	5%	2%
CMD	4%	4%	3%
COPY	3%	4%	3%
EXPOSE	4%	4%	3%
MAINTAINER	4%	4%	3%
WORKDIR	3%	3%	3%
ENTRYPOINT	2%	2%	1%
VOLUME	2%	2%	1%
USER	1%	1%	1%

Infrastructure-as-code

Research finding

[Cito et al., MSR17]

Distribution of RUN Instructions

Category	Examples	All	Top-1000	Top-100
Dependencies	apt-get, yum, npm	45.2%	44.7%	45.2%
File System	mkdir, cd, cp, rm	30.4%	29.3%	29.4%
Permissions	chmod, chown	7.3%	5.2%	2.3%
Build / Execute	make, install	5.3%	8.3%	13.5%
Environment	set, export, source	0.6%	1.0%	0.2%
Other		11.3%	11.5%	9.4%

Abstraction for Dependencies

Service-oriented architecture

introduction and motivation

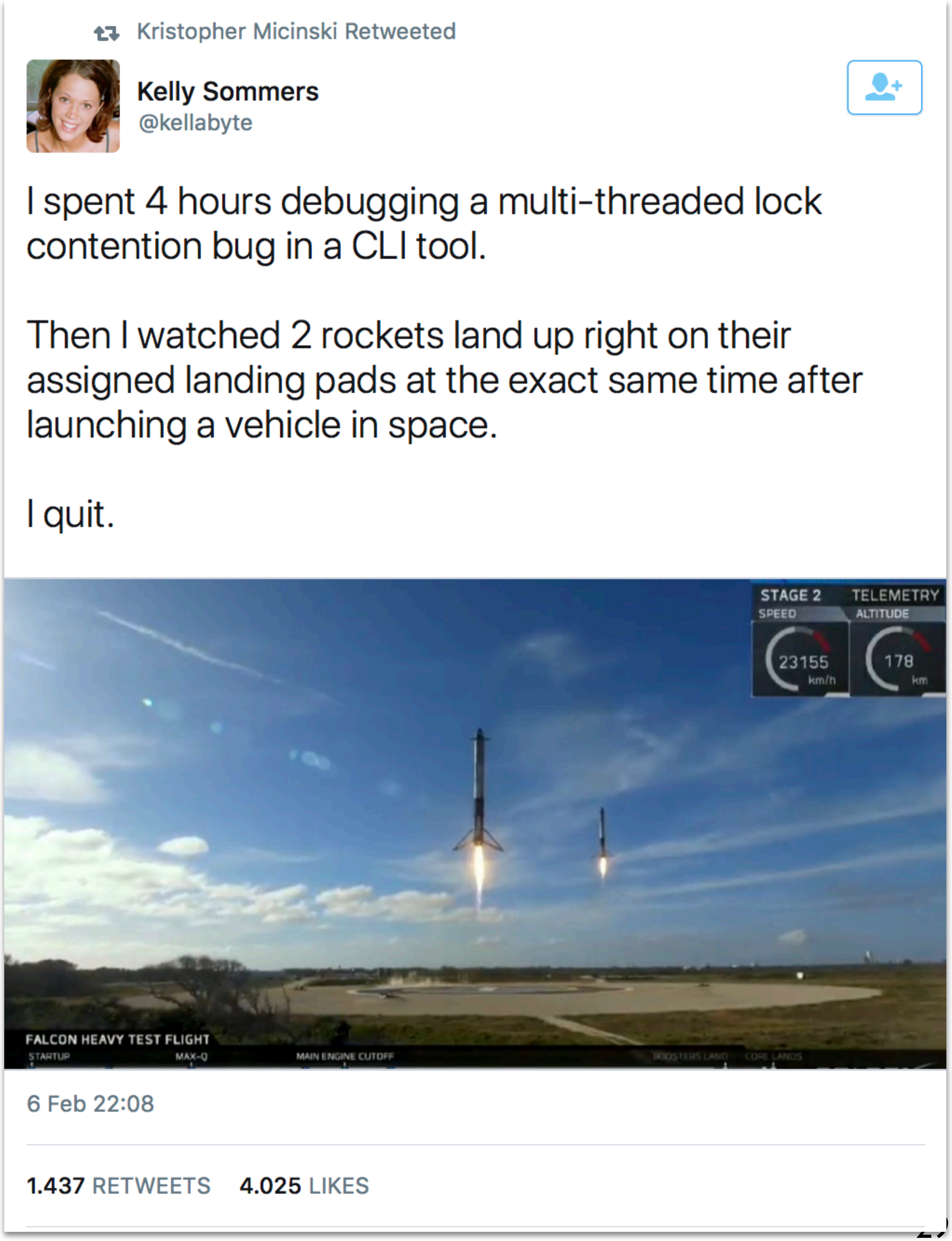
Scaling up

using concurrent actors

Scaling out

using distributed actors

Scaling up through concurrent programming



Kelly Sommers
@kellabyte

I spent 4 hours debugging a multi-threaded lock contention bug in a CLI tool.

Then I watched 2 rockets land up right on their assigned landing pads at the exact same time after launching a vehicle in space.

I quit.

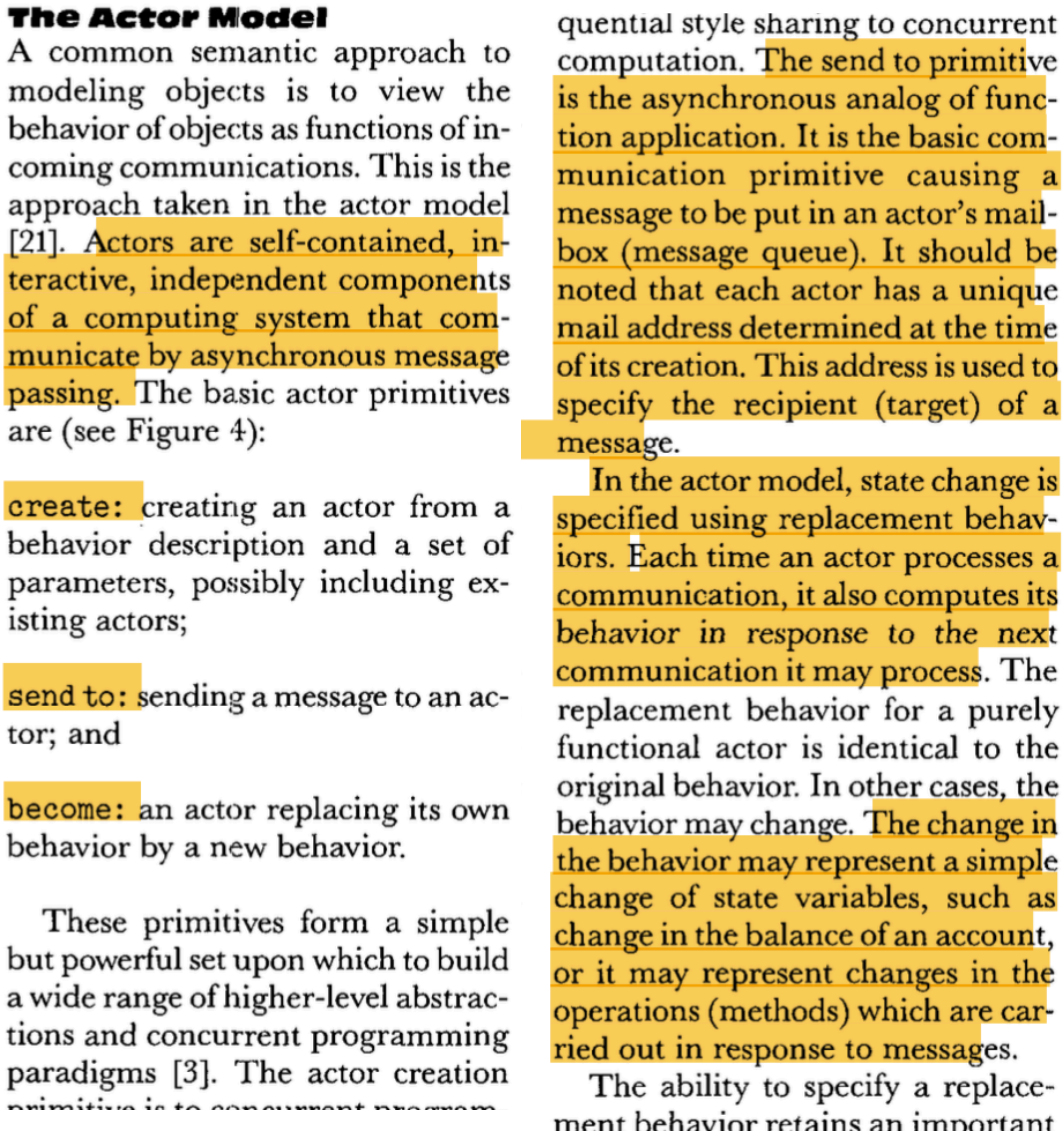
6 Feb 22:08

1.437 RETWEETS 4.025 LIKES

"Scalability is the measure to which a system can adapt to a change in demand for resources, without negatively impacting performance."

Concurrency is a means to achieve scalability:
add more threads to server when needed, which the application automatically starts using

Introduction to concurrent actor programming



The Actor Model

A common semantic approach to modeling objects is to view the behavior of objects as functions of incoming communications. This is the approach taken in the actor model [21]. Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. The basic actor primitives are (see Figure 4):

`create:` creating an actor from a behavior description and a set of parameters, possibly including existing actors;

`send to:` sending a message to an actor; and

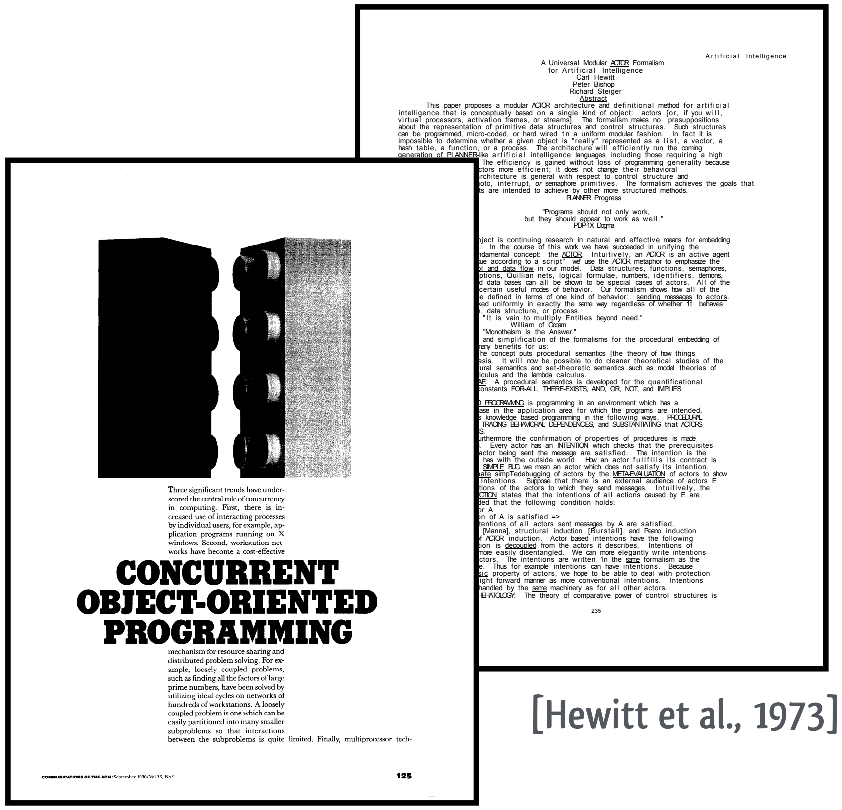
`become:` an actor replacing its own behavior by a new behavior.

These primitives form a simple but powerful set upon which to build a wide range of higher-level abstractions and concurrent programming paradigms [3]. The actor creation primitive is to concurrent program-

quential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

In the actor model, state change is specified using replacement behaviors. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. The replacement behavior for a purely functional actor is identical to the original behavior. In other cases, the behavior may change. The change in the behavior may represent a simple change of state variables, such as change in the balance of an account, or it may represent changes in the operations (methods) which are carried out in response to messages.

The ability to specify a replacement behavior retains an important



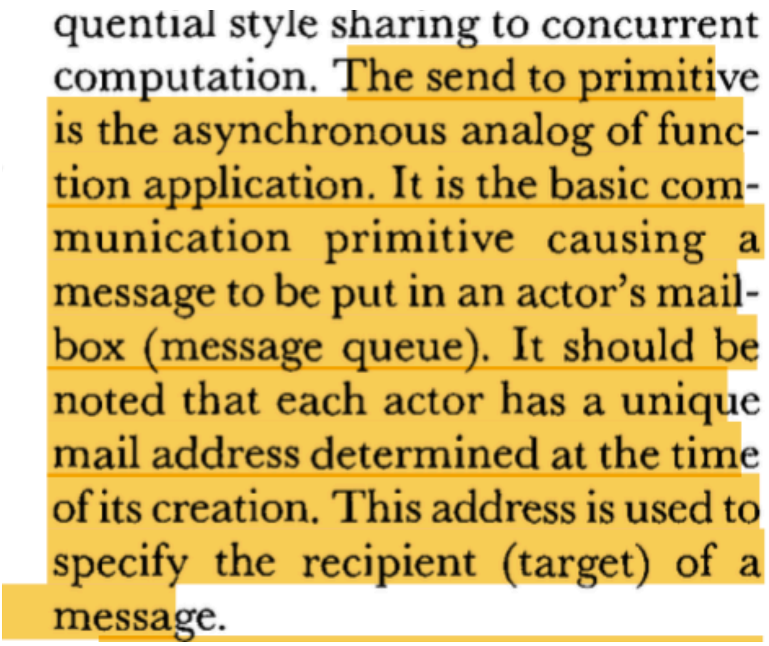
A Universal Modular ACTOR Formalism for Artificial Intelligence

CONCURRENT OBJECT-ORIENTED PROGRAMMING

[Hewitt et al., 1973]

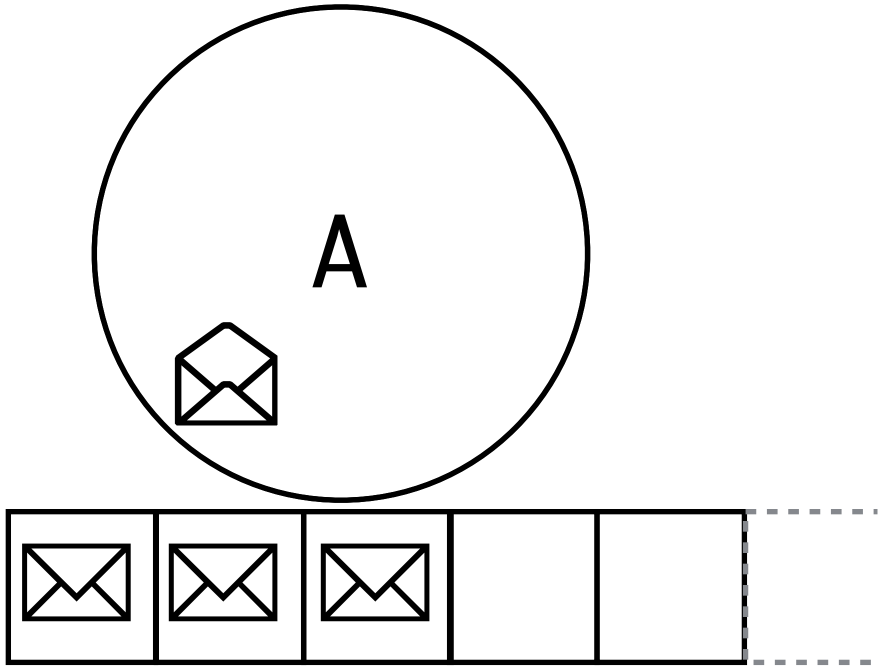
[Agha 1990]

Introduction to concurrent actor programming

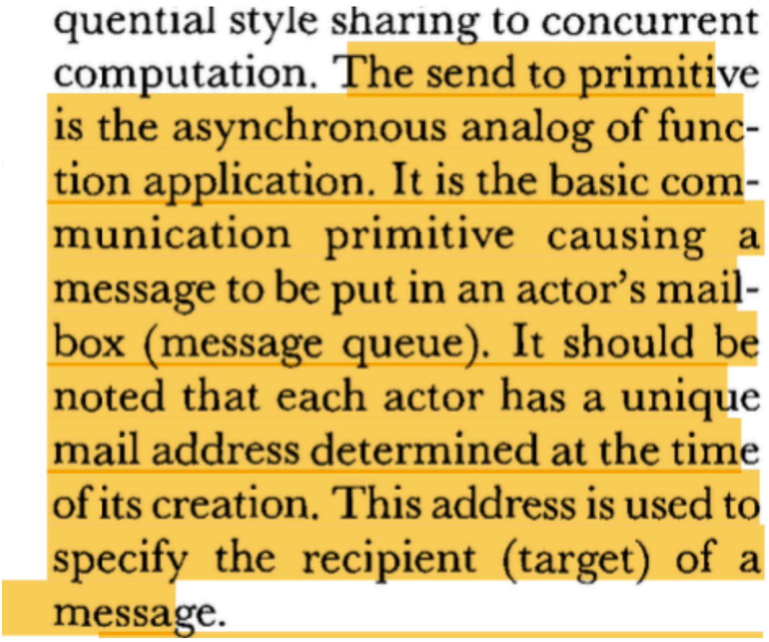


quential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

- An actor can only:
  - process messages one-by-one from a mailbox



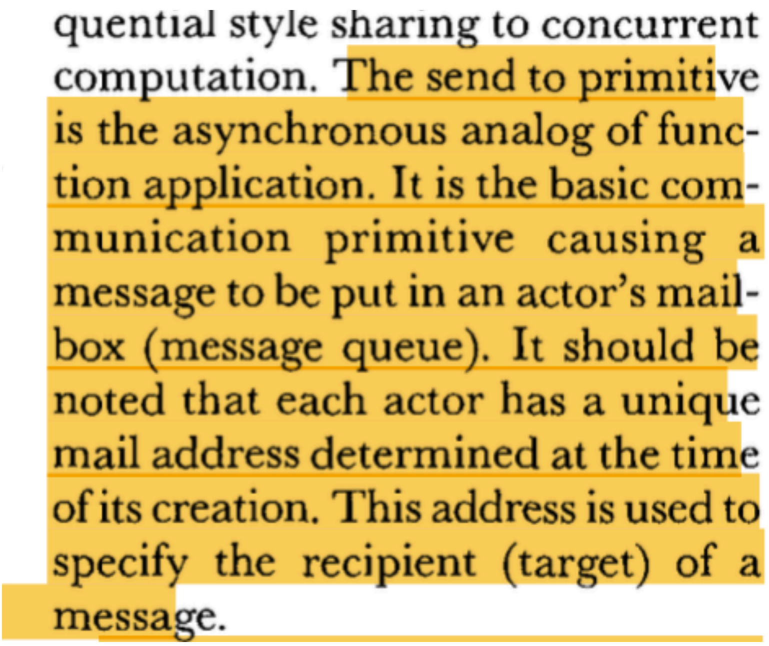
Introduction to concurrent actor programming



quential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

- An actor can only:
  - process messages one-by-one from a mailbox
  - create other actors

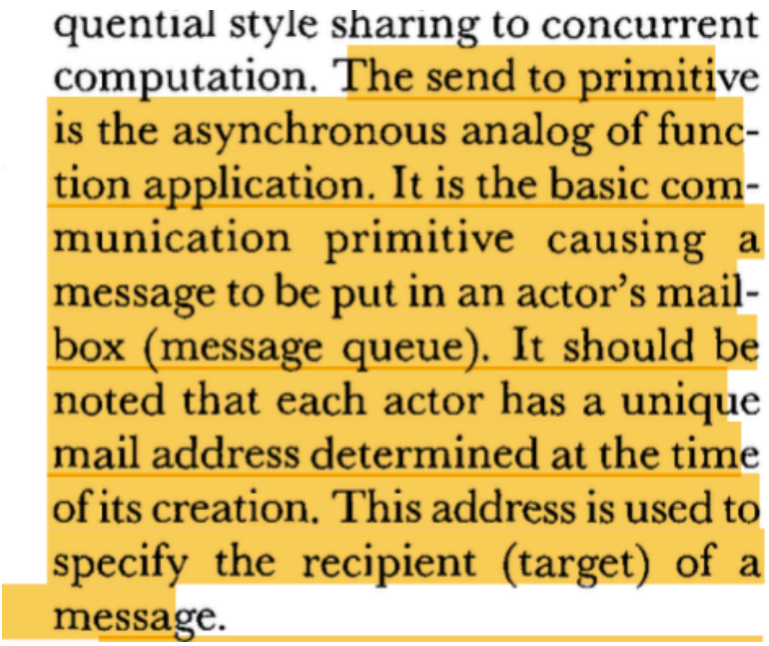
Introduction to concurrent actor programming



quential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

- An actor can only:
  - process messages one-by-one from a mailbox
  - create other actors
  - send messages to other actors asynchronously

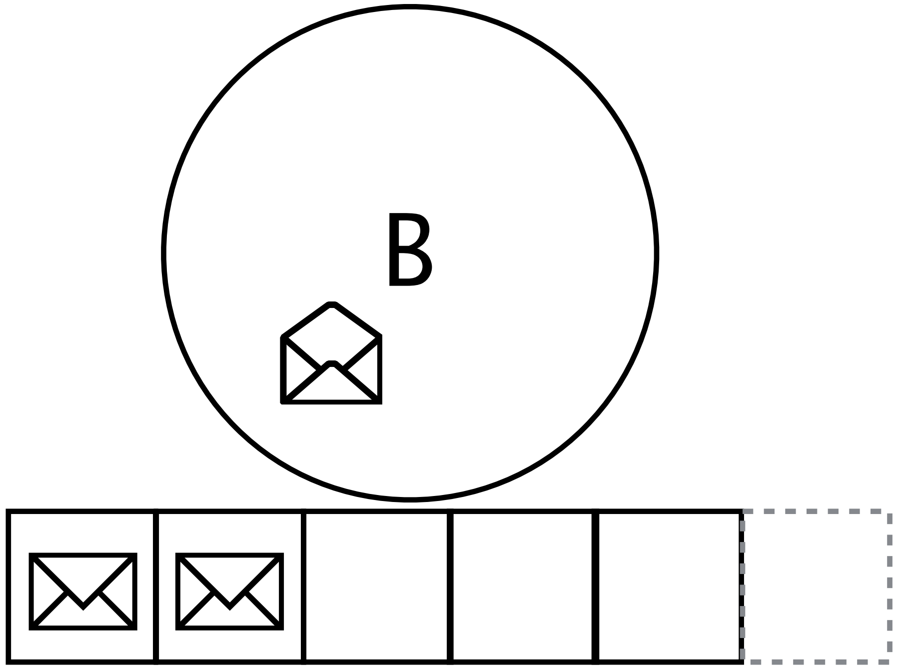
Introduction to concurrent actor programming



quential style sharing to concurrent computation. The send to primitive is the asynchronous analog of function application. It is the basic communication primitive causing a message to be put in an actor's mailbox (message queue). It should be noted that each actor has a unique mail address determined at the time of its creation. This address is used to specify the recipient (target) of a message.

- An actor can only:
  - process messages one-by-one from a mailbox
  - create other actors
  - send messages to other actors asynchronously
  - change its message processing behavior

Introduction to concurrent actor programming



- An actor is effectively single-threaded
  - messages are received and processed sequentially, the actor invokes its behaviour one-by-one on every message that is received
  - processing one message is the atomic unit of execution, it cannot be interleaved with the processing of another message
  - changes in behaviour (i.e., become) are in effect for the processing of the next message

- But message processors of separate actors can be executed concurrently!

Introduction to akka : defining actor types

Messages exchanged in our example

```
object CounterMessages {
  case object Incr
  case object Get
}
```

Counter actor with assignment

```
class Counter extends Actor {
  import CounterMessages._

  var count = 0

  def receive = {
    case Incr => count = count + 1
    case Get => sender ! count
  }

}
```

Actor trait describes behavior

```
type Receive = PartialFunction[Any, Unit]
trait Actor {
  //...
  def receive : Receive

  implicit val self : ActorRef
  def sender : ActorRef
}
```

parameter-less method returns a partial function from messages to unit

address of the sender of the current message being processed

an actor's address is accessible through variable "self"

Messages are sent to actor addresses

```
abstract class ActorRef {
    def !(msg : Any)(implicit sender : ActorRef = Actor.noSender) : Unit
    def tell(msg : Any, sender : ActorRef) = this.!(msg)(sender)
}
```

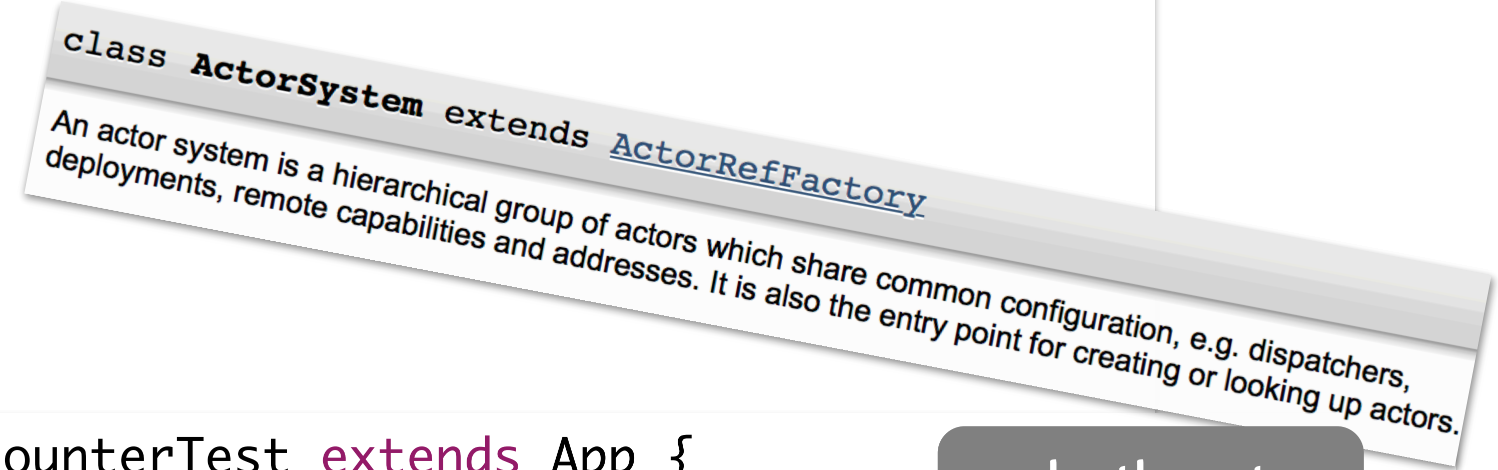
implicit val self defined in Actor trait
+ implicit parameter in method
=> the sender's address is picked up implicitly

Introduction to akka : creating actors

actors can be created by other actors

```
trait ActorContext {
  //...
  def actorOf(p: Props, name: String) : ActorRef
  def stop(a : ActorRef) : Unit
}
```

```
class CounterClient extends Actor {
  import CounterMessages._

  val counter : ActorRef = context.actorOf(Props[Counter], "counter")
  counter ! Incr
  counter ! Incr
  counter ! Get

  def receive = {
    case count: Int => {
      println(s"count was $count")
      context.stop(counter)
      context.stop(self)
    }
  }
}
}
```



class **ActorSystem** extends ActorRefFactory

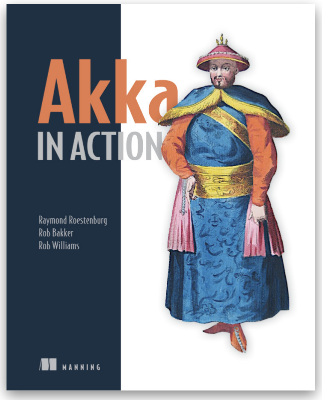
An actor system is a hierarchical group of actors which share common configuration, e.g. dispatchers, deployments, remote capabilities and addresses. It is also the entry point for creating or looking up actors.

```
object CounterTest extends App {
  val actorSystem : ActorSystem =
      ActorSystem("counterActorSystem")
  val client : ActorRef =
      actorSystem.actorOf(Props[CounterClient], "client")
  actorSystem.terminate()
}
```

or by the actor system

```
> count was 2
```

GoTicks.com: REST API



[Roestenburg et al. 2016]

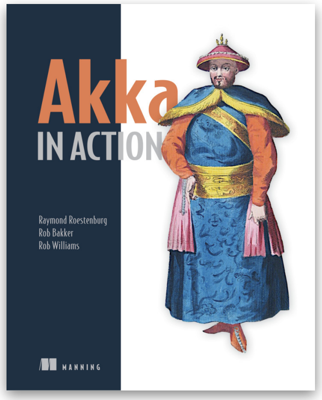
CRUD operations on resources as HTTP request-response cycles

Description	HTTP method	URL	Request body	Status code	Response example

Create an event	POST	/events/RHCP	`{ "tickets" : 250}`	201 Created	`{`
					`"name": "RHCP",`
					`"tickets": 250`
					`}`

Get all events	GET	/events	N/A	200 OK	`[ { event : "RHCP", tickets : 249 }, { event : "Radiohead", tickets : 130 } ]`
Buy tickets	POST	/events/RHCP/tickets	`{ "tickets" : 2 }`	201 Created	`{ "event" : "RHCP", "entries" : [ { "id" : 1 }, { "id" : 2 } ] }`
Cancel an event	DELETE	/events/RHCP	N/A	200 OK	`{ event : "RHCP", tickets : 249 }`

GoTicks.com: REST API

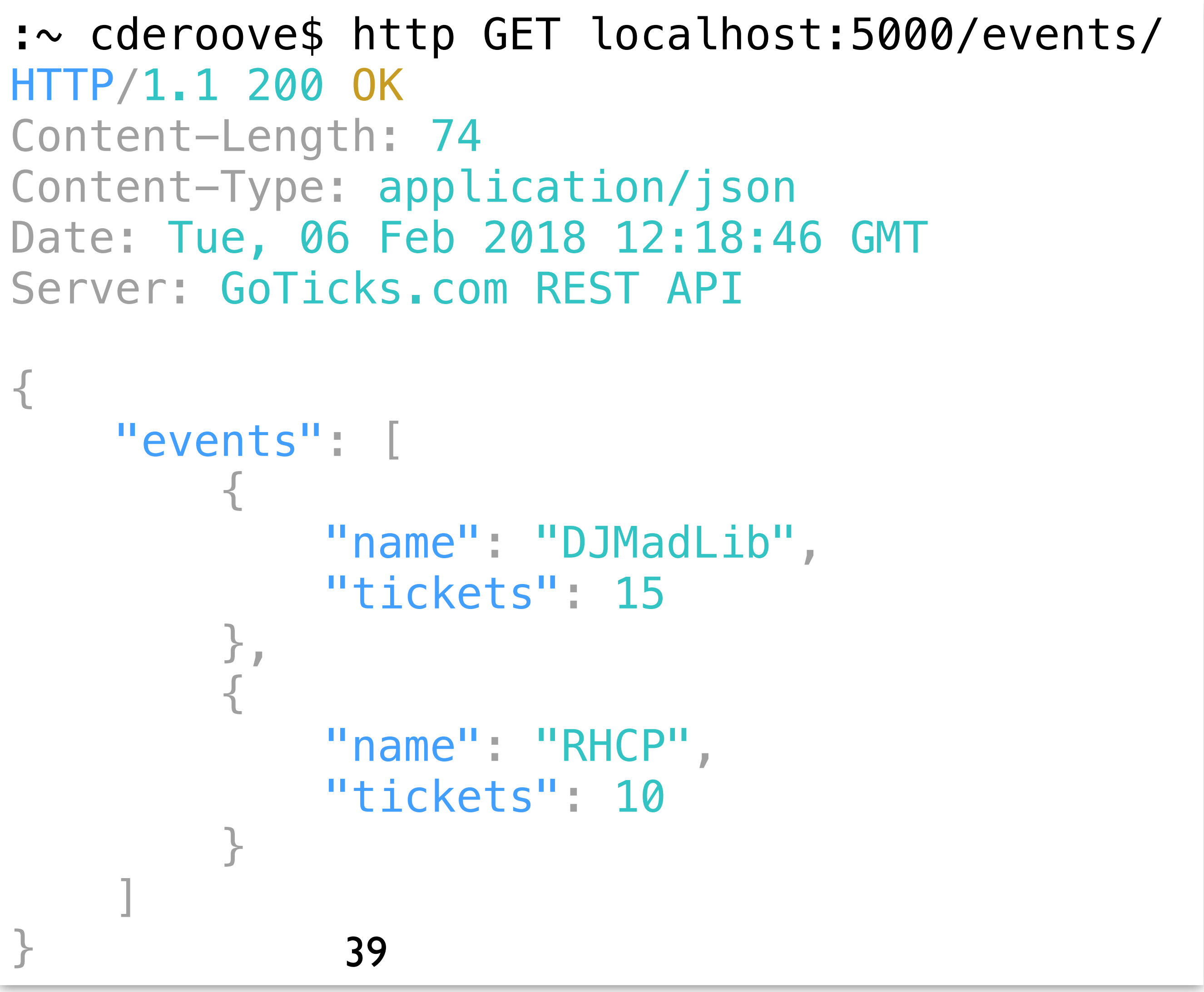


[Roestenburg et al. 2016]

create a Red Hot Chilli Peppers event with 10 tickets

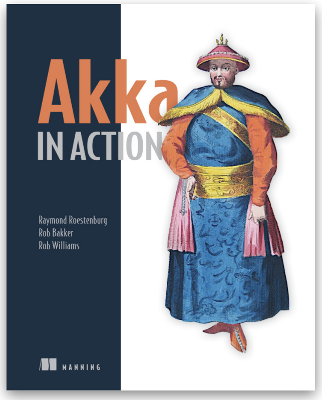
```
:~ cderoove$ http POST localhost:5000/events/RHCP tickets:=10
HTTP/1.1 201 Created
Content-Length: 28
Content-Type: application/json
Date: Tue, 06 Feb 2018 12:07:30 GMT
Server: GoTicks.com REST API

{
    "name": "RHCP",
    "tickets": 10
}
```

list available tickets for all events

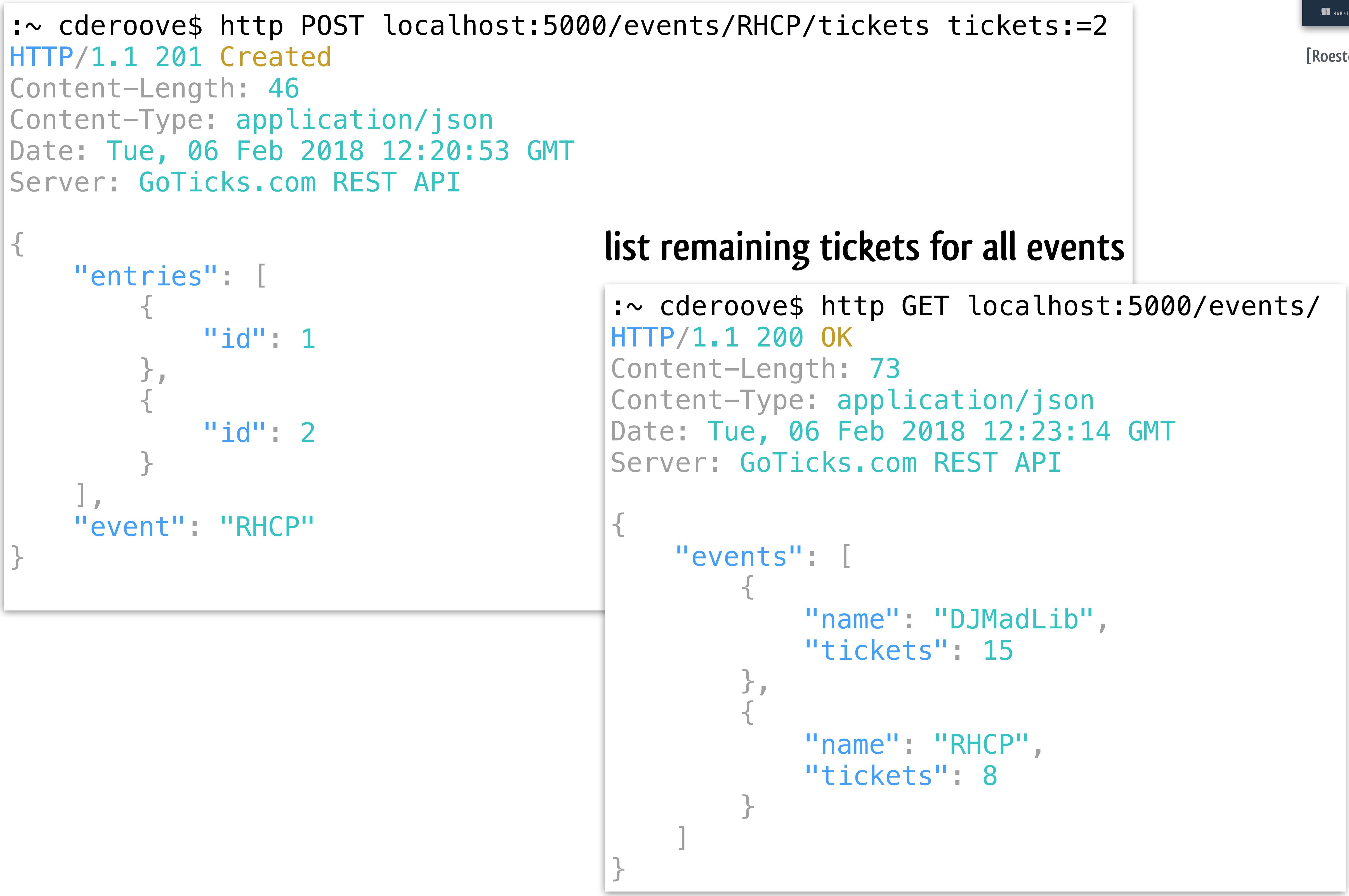


```
:~ cderoove$ http GET localhost:5000/events/
HTTP/1.1 200 OK
Content-Length: 74
Content-Type: application/json
Date: Tue, 06 Feb 2018 12:18:46 GMT
Server: GoTicks.com REST API

{
    "events": [
        {
            "name": "DJMadLib",
            "tickets": 15
        },
        {
            "name": "RHCP",
            "tickets": 10
        }
    ]
}
```

GoTicks.com: REST API


[Roestenburg et al. 2016]

purchase two tickets for Red Hot Chilli Peppers event

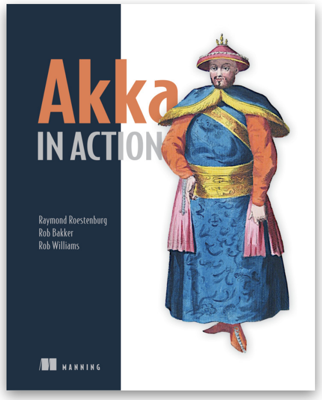


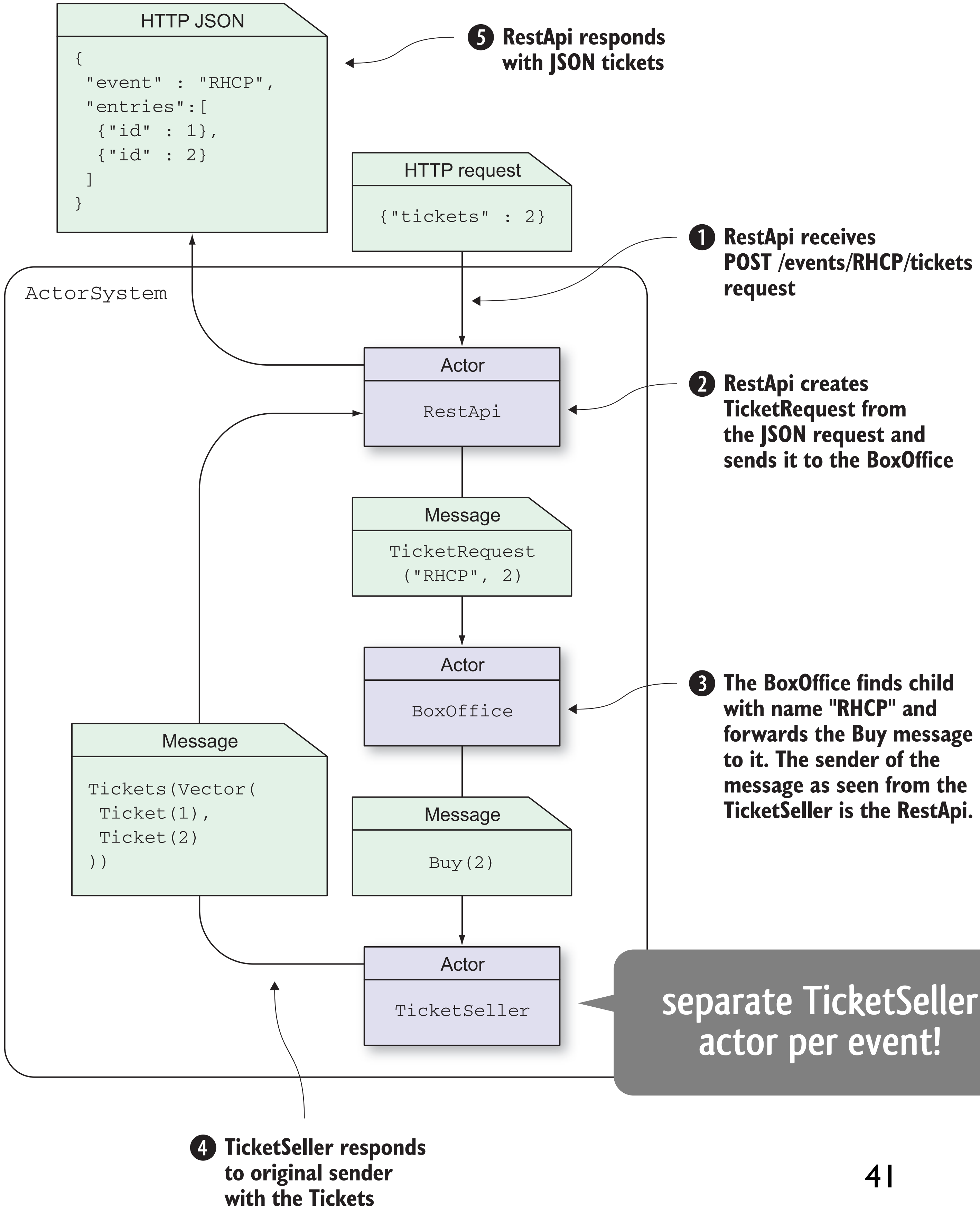
```
:~ cderoove$ http POST localhost:5000/events/RHCP/tickets tickets:=2
HTTP/1.1 201 Created
Content-Length: 46
Content-Type: application/json
Date: Tue, 06 Feb 2018 12:20:53 GMT
Server: GoTicks.com REST API

{
    "entries": [
        {
            "id": 1
        },
        {
            "id": 2
        }
    ],
    "event": "RHCP"
}
```

list remaining tickets for all events

```
:~ cderoove$ http GET localhost:5000/events/
HTTP/1.1 200 OK
Content-Length: 73
Content-Type: application/json
Date: Tue, 06 Feb 2018 12:23:14 GMT
Server: GoTicks.com REST API

{
    "events": [
        {
            "name": "DJMadLib",
            "tickets": 15
        },
        {
            "name": "RHCP",
            "tickets": 8
        }
    ]
}
```

GoTicks.com: REST API


[Roestenburg et al. 2016]

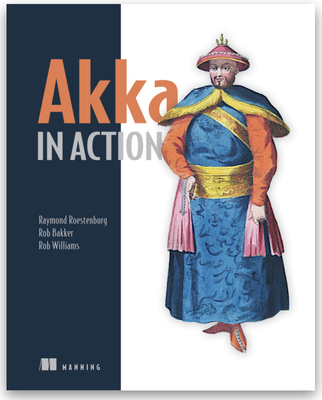


processing tickets and answering REST calls happens concurrently

separate TicketSeller actor per event!

Actors facilitate fine-grained upscaling within a container

GoTicks example: scaling upwards



[Roestenburg et al. 2016]

TicketSeller actor

```
class TicketSeller(event: String) extends Actor {
  import TicketSeller._

  var tickets = Vector.empty[Ticket]

  def receive = {
    case Add(newTickets) =>
      tickets = tickets ++ newTickets
    case Buy(nrOfTickets) =>
      val entries = tickets.take(nrOfTickets)
      if(entries.size >= nrOfTickets) {
        sender() ! Tickets(event, entries)
        tickets = tickets.drop(nrOfTickets)
      } else sender() ! Tickets(event)
    case GetEvent =>
      sender() ! Some(BoxOffice.Event(event, tickets.size))
    case Cancel =>
      sender() ! Some(BoxOffice.Event(event, tickets.size))
      self ! PoisonPill
  }
}
```

vector of numbered tickets

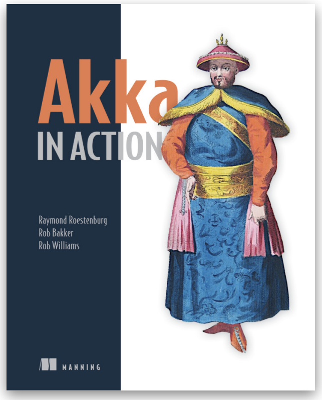
populated by the BoxOffice actor

answers with the requested number of tickets, or an empty message

answers with the remaining tickets for the event

terminates the TicketSeller actor when the event is canceled

GoTicks example: scaling upwards



[Roestenburg et al. 2016]

BoxOffice actor

```scala
class BoxOffice(implicit timeout: Timeout) extends Actor {

  def createTicketSeller(name: String) =
    context.actorOf(TicketSeller.props(name), name)

  def receive = {
    case CreateEvent(name, tickets) =>
      def create() = {
        val eventTickets = createTicketSeller(name)
        val newTickets = (1 to tickets).map { ticketId =>
          TicketSeller.Ticket(ticketId)
        }.toVector
        eventTickets ! TicketSeller.Add(newTickets)
        sender() ! EventCreated(Event(name, tickets))
      }
      context.child(name).fold(create())(_ => sender() ! EventExists)

  //...
```

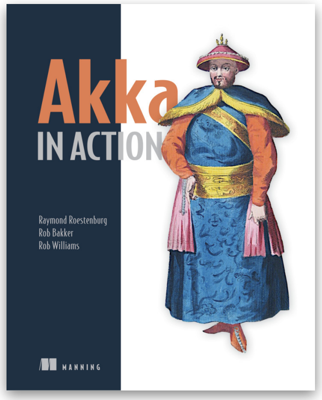
creates a TicketSeller for the given event as a child actor

adds a vector of numbered tickets to the seller's inventory

communicates success back to RestAPI actor

checks whether a TicketSeller for the given event already exists, and creates one otherwise

GoTicks example: scaling upwards



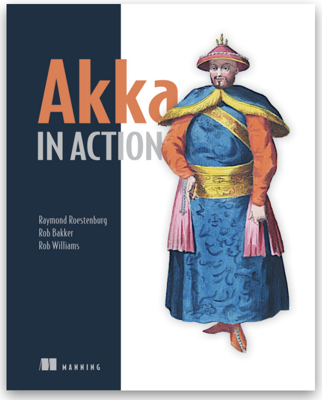
[Roestenburg et al. 2016]

```
//…
case GetTickets(event, tickets) =>
  def notFound() = sender() ! TicketSeller.Tickets(event)
  def buy(child: ActorRef) =
    child.forward(TicketSeller.Buy(tickets))
  context.child(event).fold(notFound())(buy)

case GetEvent(event) =>
  def notFound() = sender() ! None
  def getEvent(child: ActorRef) = child forward TicketSeller.GetEvent
  context.child(event).fold(notFound())(getEvent)

 case CancelEvent(event) =>
   def notFound() = sender() ! None
   def cancelEvent(child: ActorRef) = child forward TicketSeller.Cancel
   context.child(event).fold(notFound())(cancelEvent)
//…
```

forwards, rather than sends, a Buy message to the appropriate child actor

this ensures responses will go to the RESTApi Actor

GoTicks example: scaling upwards



[Roestenburg et al. 2016]

```
  //…
  case GetEvents =>
    import akka.pattern.ask
    import akka.pattern.pipe

    def getEvents = context.children.map { child =>
      self.ask(GetEvent(child.path.name)).mapTo[Option[Event]]
    }

    def convertToEvents(f: Future[Iterable[Option[Event]]]) =
      f.map(_.flatten).map(l=> Events(l.toVector))

    pipe(convertToEvents(Future.sequence(getEvents))) to sender()
}
```

asks sends a message and returns a future for the response

pipe forwards the value the future resolves to, as soon as it becomes available

but of course, asynchronous programming needs some getting used to!

Service-oriented architecture

introduction and motivation

Scaling up

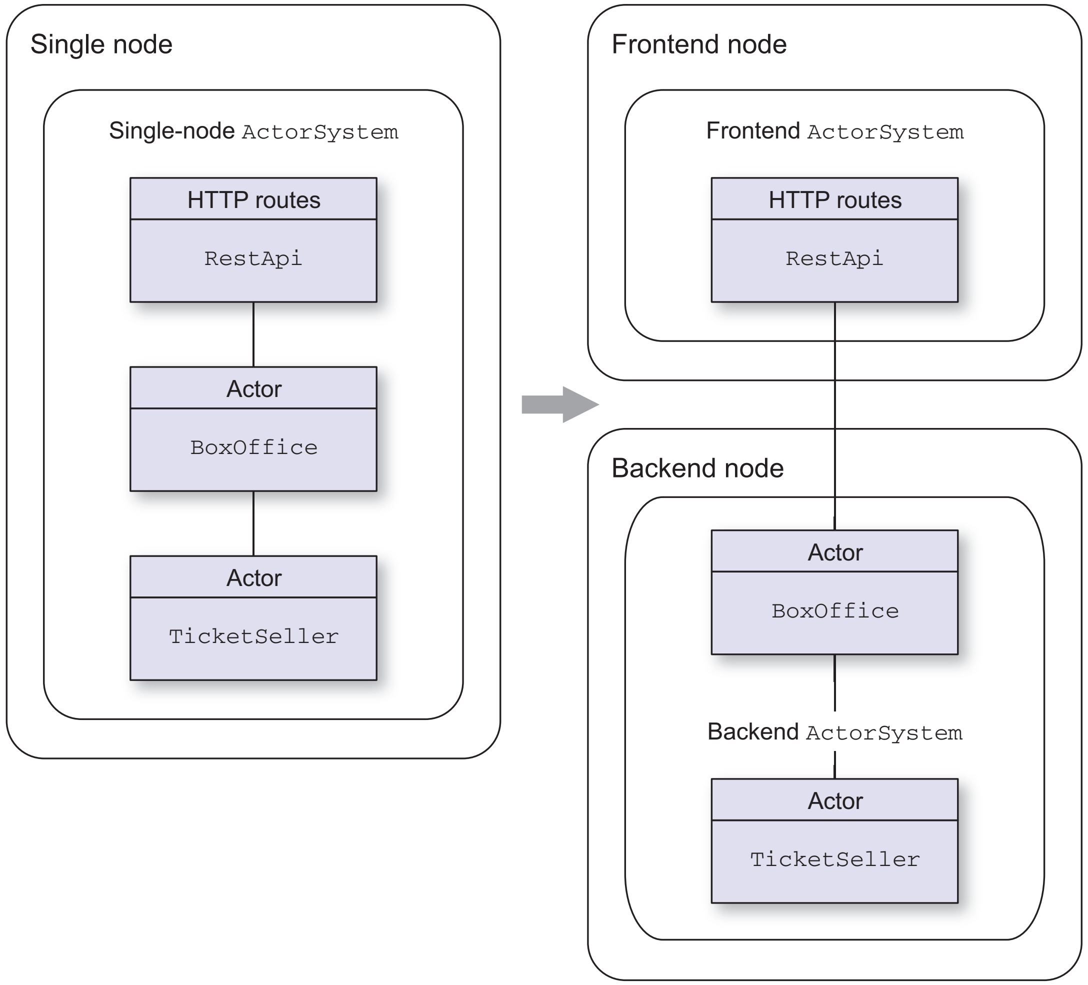
using concurrent actors

Scaling out

using distributed actors

Scaling out through distributed programming

Distribution is another means to achieve scalability:
add threads from different network nodes to the application



Single node

Single-node `ActorSystem`

HTTP routes
`RestApi`

Actor
`BoxOffice`

Actor
`TicketSeller`

Frontend node

Frontend `ActorSystem`

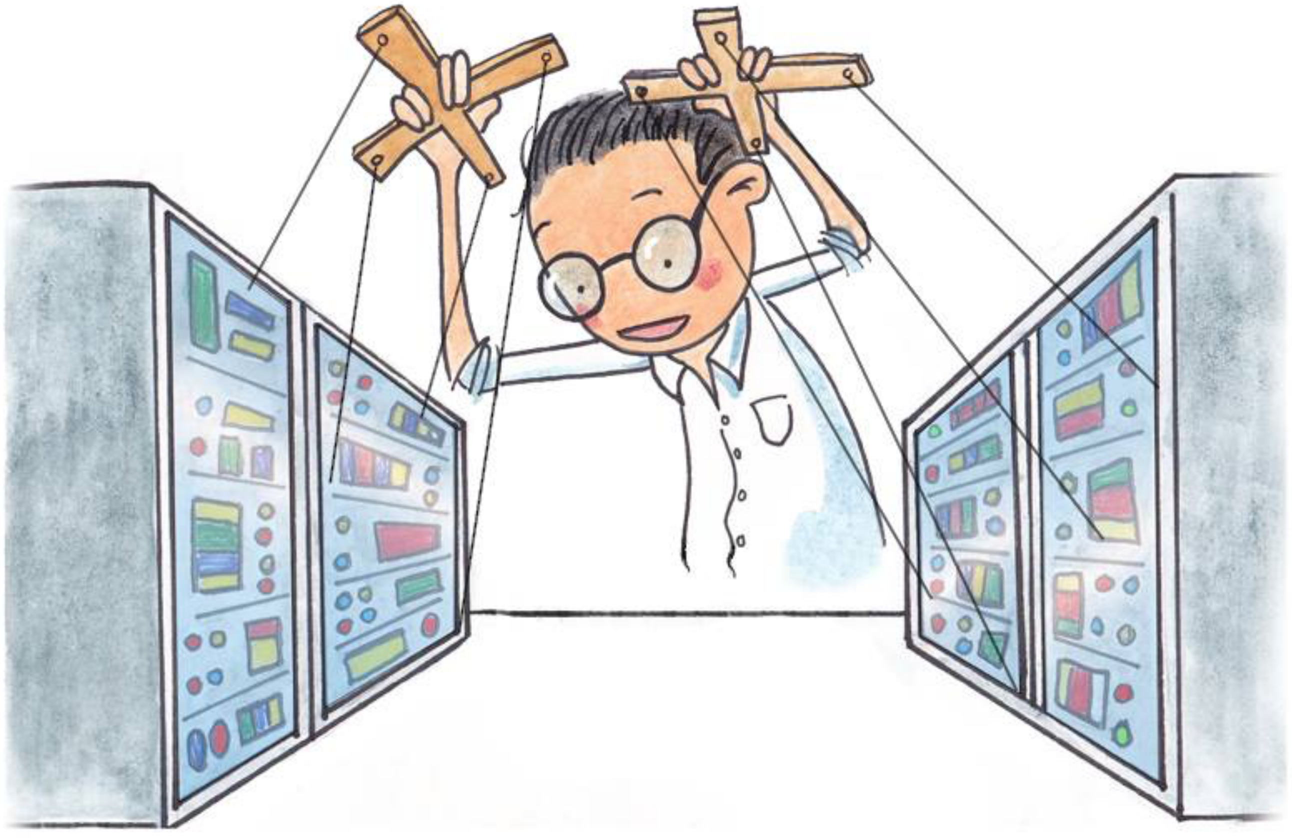
HTTP routes
`RestApi`

Backend node

Actor
`BoxOffice`

Backend `ActorSystem`

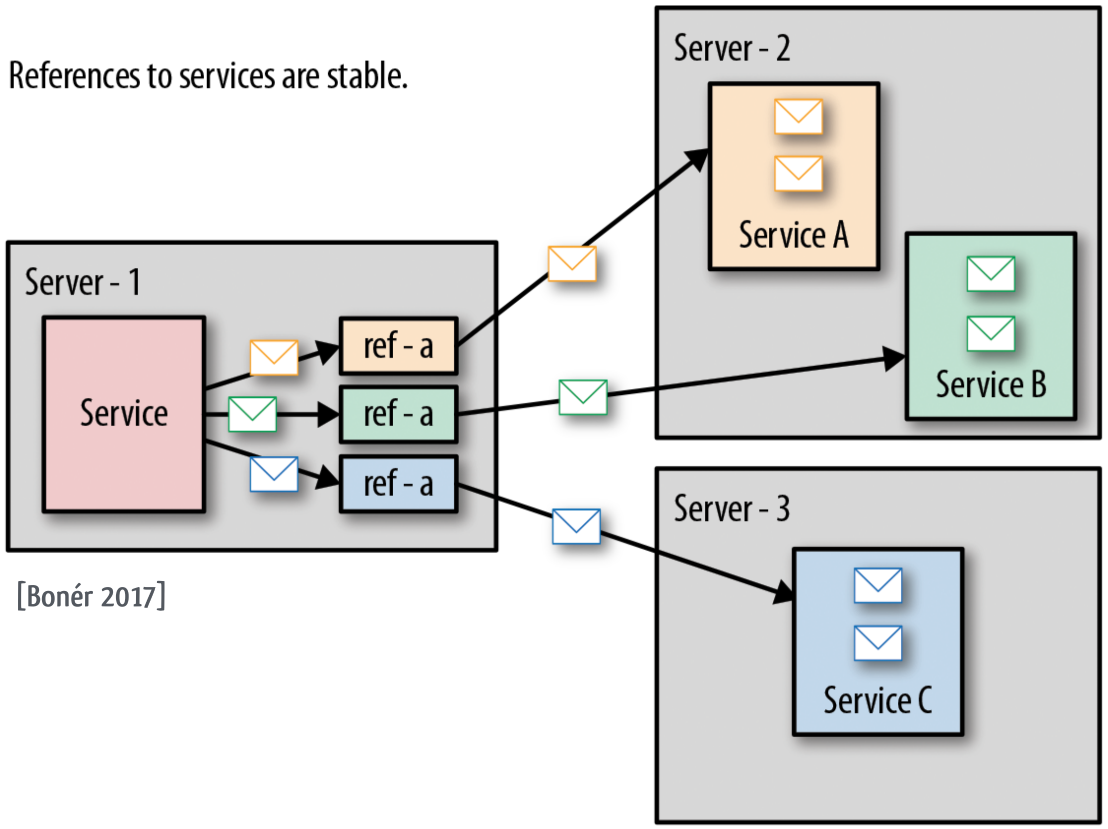
Actor
`TicketSeller`



Introduction to distributed actor programming

- actor systems are distributable by design
  - strong encapsulation: no shared data
  - location-transparent communication through addresses (ActorRefs): same ! for sending asynchronous message to local and to remote ActorRef

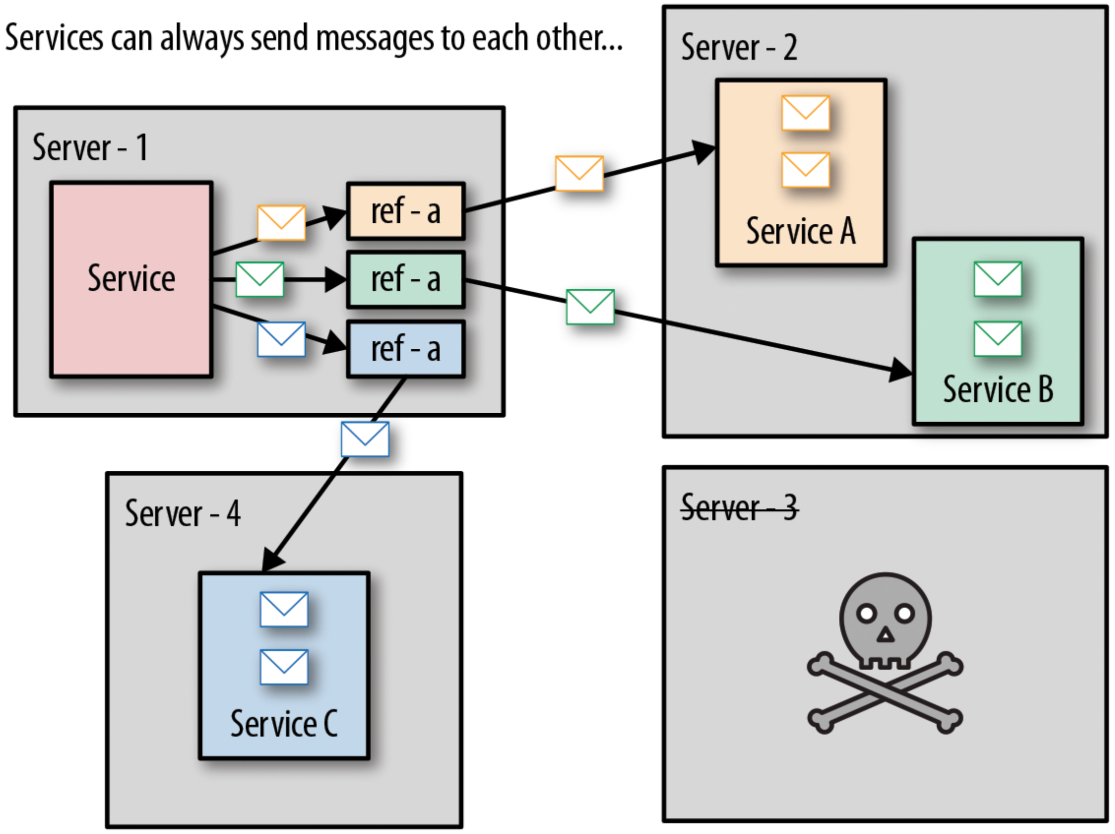
- actor systems are resilient by design
  - strong encapsulation: failures don't cascade to other parts
  - actors are created by a supervisor, to whom failure handling is delegated: enables decoupling business logic from failure handling
  - flexible supervision strategies: stop, escalate, restart...



References to services are stable.

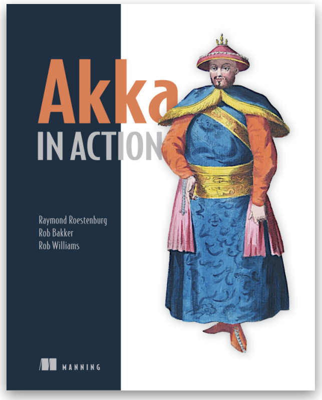
[Bonér 2017]

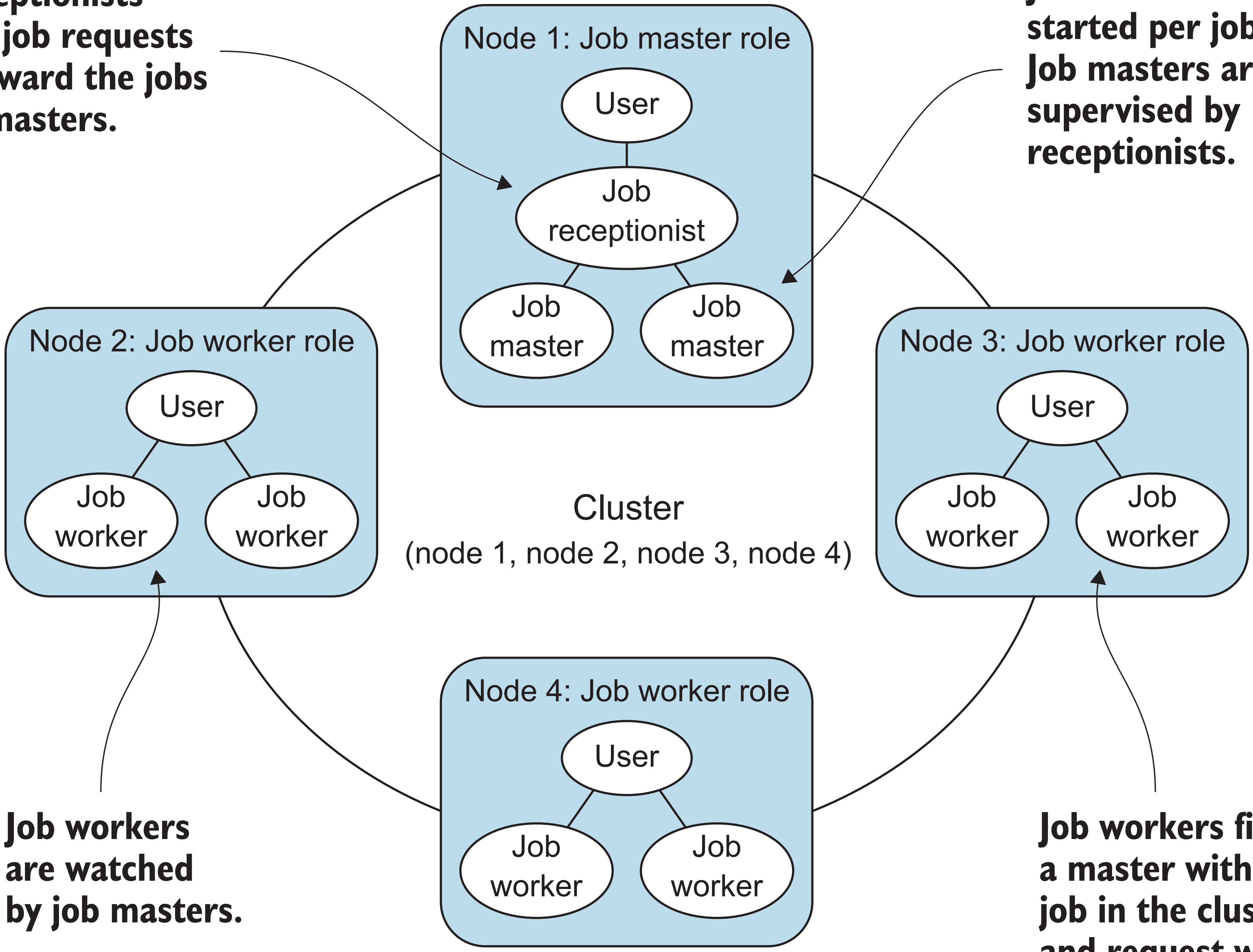
Services can always send messages to each other...



...even as the recipient is on the move!

Scaling out: word counting cluster


[Roestenburg et al. 2016]



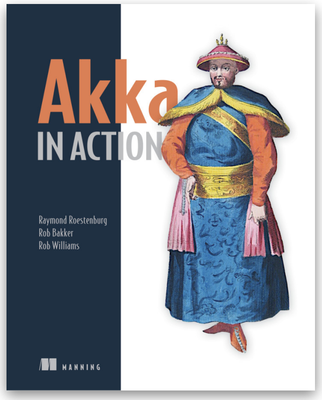
Job receptionists receive job requests and forward the jobs to job masters.

Job masters are started per job. Job masters are supervised by receptionists.

Job workers are watched by job masters.

Job workers find a master with a job in the cluster and request work.

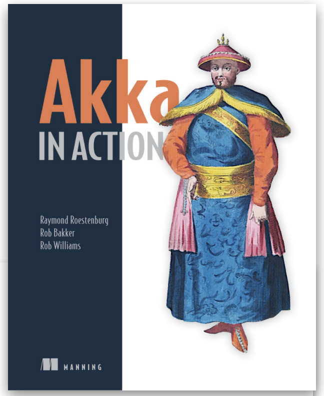
Words cluster: starting JVM nodes



[Roestenburg et al. 2016]

```
java -DPORT=2551
     -Dconfig.resource=/seed.conf
     -jar target/words-node.jar

java -DPORT=2554
     -Dconfig.resource=/master.conf
     -jar target/words-node.jar

java -DPORT=2555
     -Dconfig.resource=/worker.conf
     -jar target/words-node.jar

java -DPORT=2556
     -Dconfig.resource=/worker.conf
     -jar target/words-node.jar
```

Words cluster: entry point for each JVM



[Roestenburg et al. 2016]

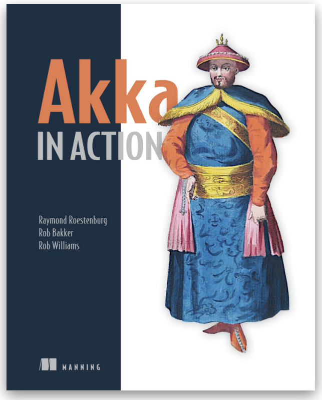
join the “words” cluster, using the given role configuration

```
object Main extends App {
  val config = ConfigFactory.load()
  val system = ActorSystem("words", config)

  println(s"Starting node with roles: ${Cluster(system).selfRoles}”)

  if(system.settings.config.getStringList("akka.cluster.roles")
     .contains("master")) {
    Cluster(system).registerOnMemberUp {
      val receptionist = system.actorOf(Props[JobReceptionist], "receptionist")
      println("Master node is ready.")

      val text = List("this is a test", "of some very naive word counting",
                      "but what can you say", "it is what it is")

      receptionist ! JobRequest("the first job",
                                (1 to 100000).flatMap(i => text ++ text).toList)
   }
  }
}
```

start up JobReceptionist actor if this node has been assigned the master role

send the receptionist a very large text to count words in

Words cluster: router for work distribution



[Roestenburg et al. 2016]

```
trait CreateWorkerRouter {
  this: Actor =>

  def createWorkerRouter: ActorRef = {
    context.actorOf(
      ClusterRouterPool(BroadcastPool(10),
                        ClusterRouterPoolSettings(totalInstances = 100,
                                                  maxInstancesPerNode = 20,
                                                  allowLocalRoutees = false,
                                                  useRole = None))
                                                    .props(Props[JobWorker]),
      name = "worker-router")
  }
}
```

so-called self-type:
expresses that this trait can only be mixed in with Actor types

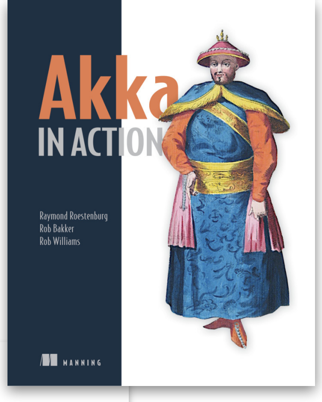
creates an actor of a built-in router type that will create at pool of 10 JobWorker children in the cluster

no workers on this node

and only on nodes with this role

type of the created actor children

Words cluster: master in idle state



[Roestenburg et al. 2016]

```
class JobMaster extends Actor
                   with ActorLogging
                   with CreateWorkerRouter {
  //…
  val router = createWorkerRouter

  override def supervisorStrategy: SupervisorStrategy =
    SupervisorStrategy.stoppingStrategy

  def receive = idle

  def idle: Receive = {
    case StartJob(jobName, text) =>
      textParts = text.grouped(10).toVector
      val cancellable =
        context.system.scheduler.schedule(0 millis, 1000 millis,
                                          router, Work(jobName, self))
      context.setReceiveTimeout(60 seconds)
      become(working(jobName, sender, cancellable))
  }
  //…
```

address of newly-created router actor

initial message-processing function

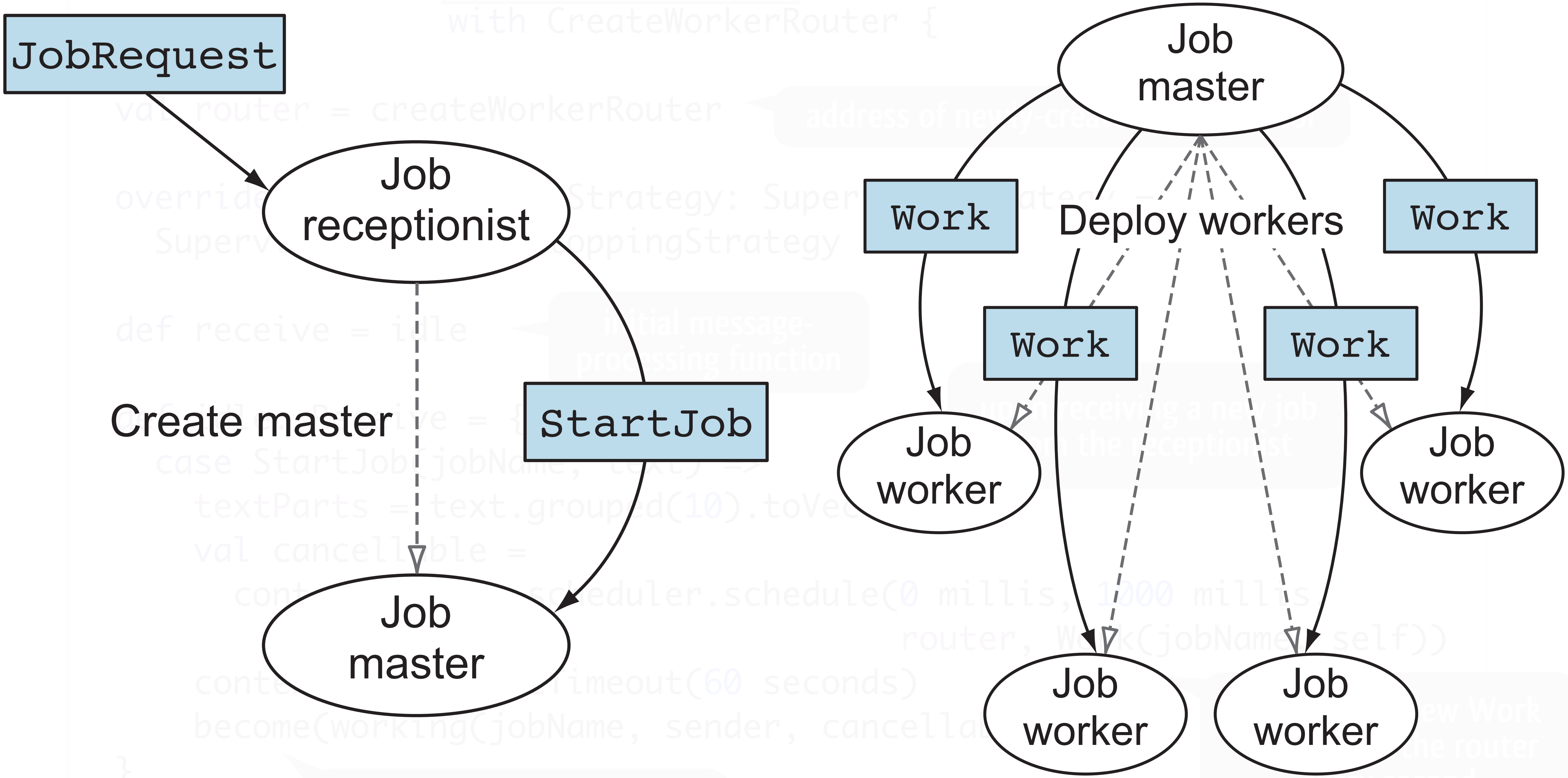
upon receiving a new job from the receptionist

expect worker to enlist / submit results before timeout

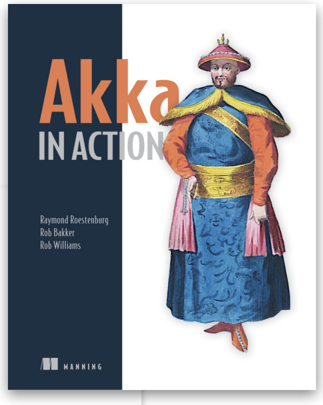
schedules a new Work message to the worker actors pool every second, in case new workers have joined the pool since the job was started

transitions to the working state, with a new message processing function

Words cluster: master in idle state



Words cluster: master in working state



[Roestenburg et al. 2016]

```
def working(jobName: String, receptionist: ActorRef,
            cancellable: Cancellable): Receive = {
  case Enlist(worker) =>
    watch(worker)
    workers  = workers + worker

  case NextTask =>
    if(textParts.isEmpty) {
      sender() ! WorkLoadDepleted
    } else {
      sender() ! Task(textParts.head, self)
      workGiven = workGiven + 1
      textParts = textParts.tail
    }

  case TaskResult(countMap) =>
    intermediateResult = intermediateResult :+ countMap
    workReceived = workReceived + 1

    if(textParts.isEmpty && workGiven == workReceived) {
      cancellable.cancel()
      become(finishing(jobName, receptionist, workers))
      setReceiveTimeout(Duration.Undefined)
      self ! MergeResults
    }
```

watch for termination of, and keep track of workers that enlist with this job master

give work to workers requesting it

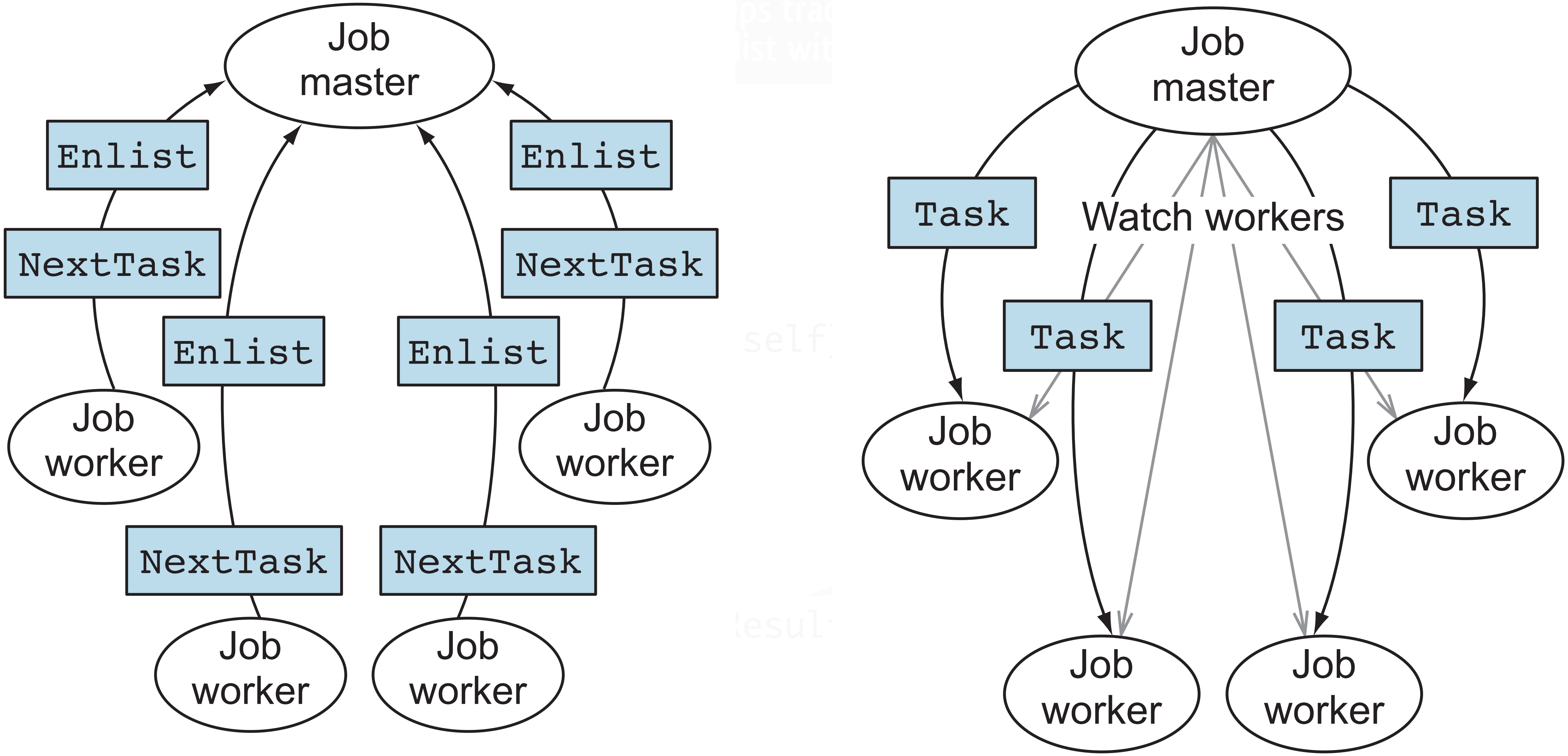
collect intermediate results in a vector of countMaps

transition to finishing state

start merging results

Words cluster: master in working state

```
def working(jobName: String, receptionist: ActorRef,
            cancellable: Cancellable): Receive = {
  case Enlist(worker) =>
```

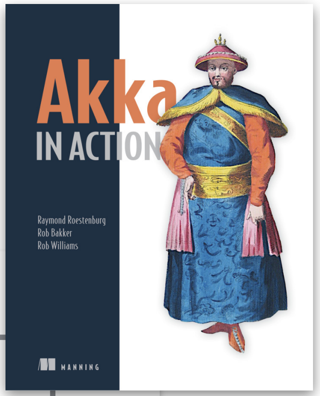


```
    if(textParts.isEmpty && workGiven == workReceived) {
      cancellable.cancel()
      become(finishing(jobName, receptionist, workers))
      setReceiveTimeout(Duration.Undefined)
      self ! MergeResults
    }
```

transition to finishing state

start merging results

Words cluster: managing worker termination



[Roestenburg et al. 2016]

```
 //…
 case ReceiveTimeout =>
    if(workers.isEmpty) {
      log.info(s"No workers responded in time. Cancelling job $jobName.")
      stop(self)
    } else setReceiveTimeout(Duration.Undefined)

  case Terminated(worker) =>
    log.info(s"Worker $worker got terminated. Cancelling job $jobName.")
    stop(self)
}

def finishing(jobName: String,
              receptionist: ActorRef,
              workers: Set[ActorRef]): Receive = {
  case MergeResults =>
    val mergedMap = merge()
    workers.foreach(stop(_))
    receptionist ! WordCount(jobName, mergedMap)

  case Terminated(worker) =>
    log.info(s"Job $jobName is finishing.
              Worker ${worker.path.name} is stopped.")
}
```

JobMaster stops itself in case no workers have enlisted before timeout

JobMaster stops as soon as one if its workers fails

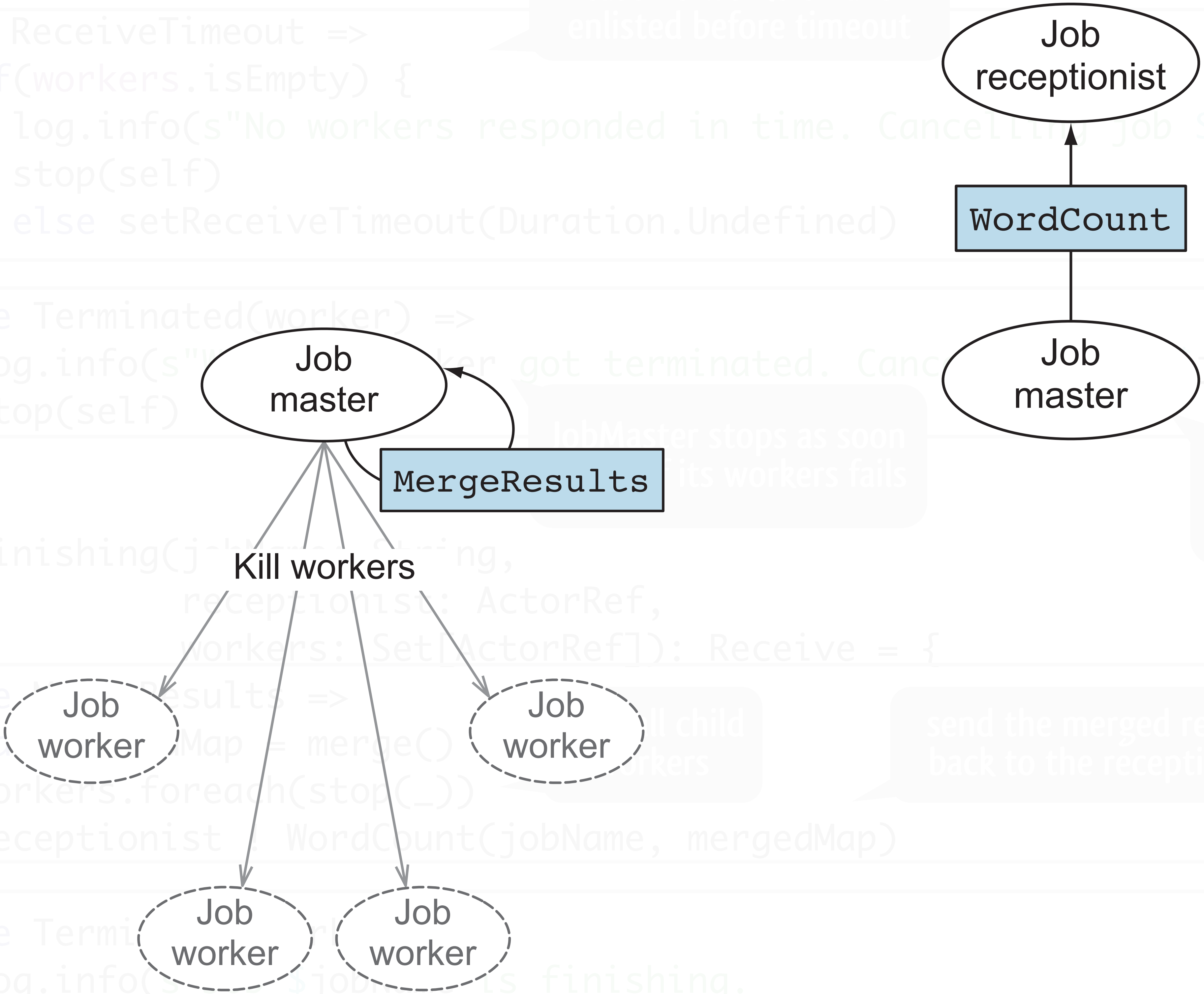
watching workers => notified of termination

stop all child workers

send the merged results back to the receptionist

Words cluster: managing worker termination

[Roestenburg et al. 2016]



```
//…
case ReceiveTimeout =>
  if(workers.isEmpty) {
    log.info(s"No workers responded in time. Cancelling job $jobName.")
    stop(self)
  } else setReceiveTimeout(Duration.Undefined)

 case Terminated(worker) =>
  log.info(s"W          ker got terminated. Canc          jobName.")
  stop(self)
}

def finishing(j          ing,
          receptionist: ActorRef,
          workers: Set[ActorRef]): Receive = {
 case          esults =>
  va          Map = merge()
  workers.foreach(stop(_))
  receptionist ! WordCount(jobName, mergedMap)

 case Termi
  log.info(s      $jobN      is finishing.
          Worker ${worker.path.name} is stopped.")
}
```

JobMaster stops itself in case no workers have enlisted before timeout

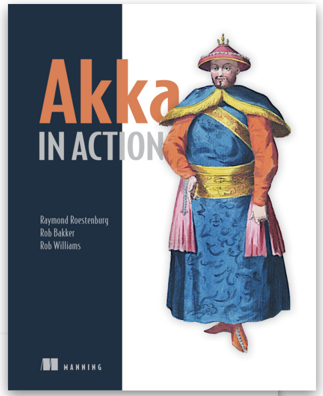
JobMaster stops as soon   its workers fails

watching workers => notified of termination

ll child orkers

send the merged results back to the receptionist

Words cluster: worker in idle state



[Roestenburg et al. 2016]

```
class JobWorker extends Actor
                   with ActorLogging {

  var processed = 0

  def receive = idle

  def idle: Receive = {
    case Work(jobName, master) =>
      become(enlisted(jobName, master))

      log.info(s"Enlisted, will start requesting work for job '${jobName}'.")
      master ! Enlist(self)
      master ! NextTask
      watch(master)

      setReceiveTimeout(30 seconds)
  }

  //…
```

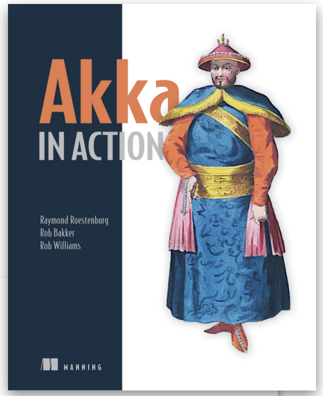
initial message-processing function

enlist for job, change state

request task and watch JobMaster for termination

expect a task from the JobMaster within 30 seconds

Words cluster: worker in enlisted state



[Roestenburg et al. 2016]

```
def enlisted(jobName: String, master: ActorRef): Receive = {
  case ReceiveTimeout =>
    master ! NextTask

  case Task(textPart, master) =>
    val countMap = processTask(textPart)
    processed = processed + 1
    master ! TaskResult(countMap)
    master ! NextTask

  case WorkLoadDepleted =>
    log.info(s"Work load ${jobName} is depleted, retiring...")
    setReceiveTimeout(Duration.Undefined)
    become(retired(jobName))

  case Terminated(master) =>
    setReceiveTimeout(Duration.Undefined)
    log.error(s"Master terminated that ran Job ${jobName}, stopping self.")
    stop(self)

}
```

request another task again upon timeout

process the received task, send the result to the JobMaster, and request another task

job finished: switches of ReceiveTimeout and transition to retired state

stop when master terminated

Words cluster: worker in enlisted state

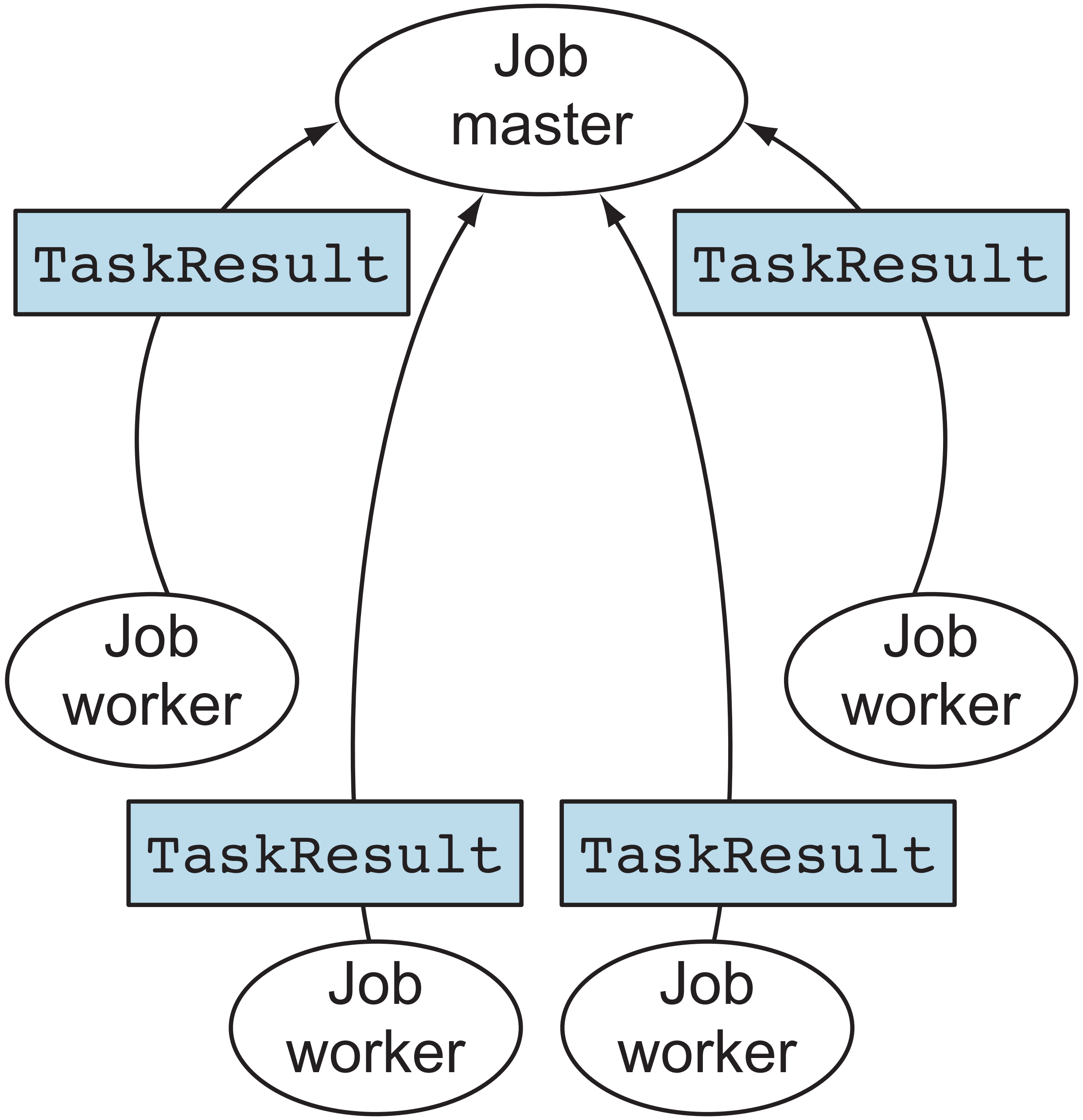
[Roestenburg et al. 2016]

```
def enlisted(jobName: String, master: ActorRef): Receive = {
  case ReceiveTi
    master ! Next

  case Task(textP
    val countMap
    processed = p
    master ! Task
    master ! Next

  case WorkLoadDe
    log.info(s"Wo                                         .")
    setReceiveTim
    become(retire

  case Terminated
    setReceiveTim
    log.error(s"M                                  stopping self.")
    stop(self)
}
```

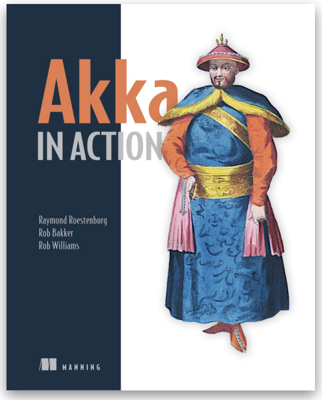
sk
ut

nd the result to
another task

itches of ReceiveTimeout
tion to retired state



Words cluster: worker in retired state



[Roestenburg et al. 2016]

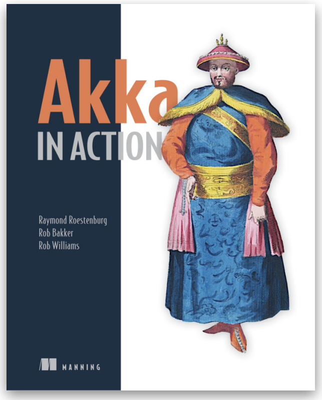
```
 //…
 def retired(jobName: String): Receive = {
   case Terminated(master) =>
     log.error(s"Master terminated that ran Job ${jobName}, stopping self.")
     stop(self)

   case _ => log.error("I'm retired.")
  }

  def processTask(textPart: List[String]): Map[String, Int] = {
    textPart.flatMap(_.split("\\W+"))
      .foldLeft(Map.empty[String, Int]) {
      (count, word) =>
        if (word == "FAIL") throw new RuntimeException("SIMULATED FAILURE!")
        count + (word -> (count.getOrElse(word, 0) + 1))
    }
  }
}
```

terminate when master has terminated

should no longer receive messages in retired state

crash when the text for the given task contains the word FAIL

Words cluster: receptionist



[Roestenburg et al. 2016]

```
class JobReceptionist extends Actor
                         with ActorLogging
                         with CreateMaster {
  def receive = {
    case jr @ JobRequest(name, text) =>
      log.info(s"Received job $name")
      val masterName = "master-"+URLEncoder.encode(name, "UTF8")
      val jobMaster = createMaster(masterName)
      val job = Job(name, text, sender, jobMaster)
      jobs = jobs + job
      jobMaster ! StartJob(name, text)
      watch(jobMaster)

    case WordCount(jobName, map) =>
      log.info(s"Job $jobName complete.")
      log.info(s"result:${map}")
      jobs.find(_.name == jobName).foreach { job =>
        job.respondTo ! JobSuccess(jobName, map)
        stop(job.jobMaster)
        jobs = jobs - job
      }
  //...
```

create a new JobMaster for the newly-received job request

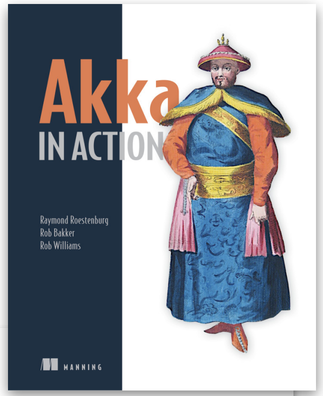
remember this job request (and its sender)

watch the JobMaster for termination

send back the result to the sender of the job request

stop the JobMaster

Words cluster: receptionist with resilient jobs


[Roestenburg et al. 2016]

upon termination of one of the watched JobMasters

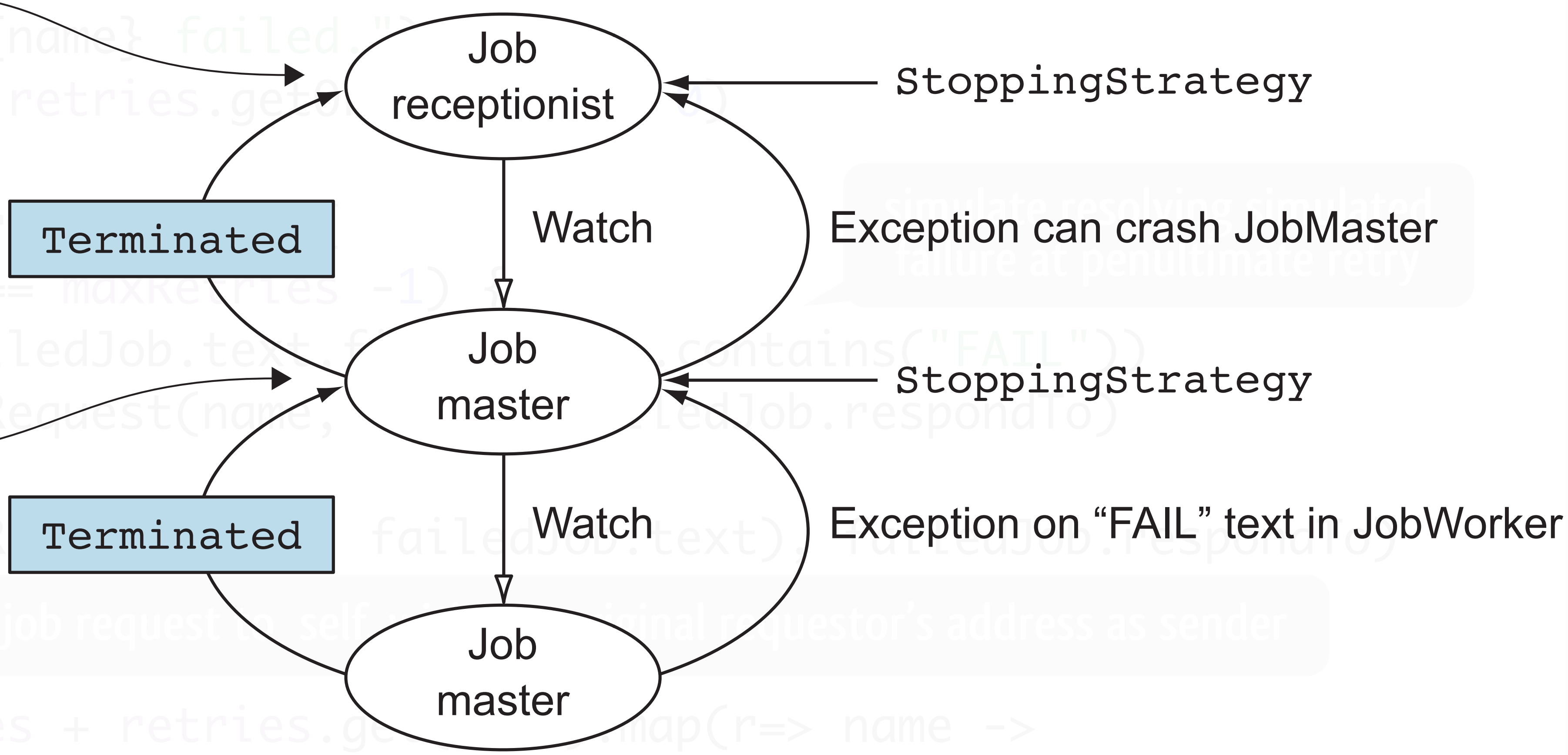
```
 case Terminated(jobMaster) =>
   jobs.find(_.jobMaster == jobMaster).foreach { failedJob =>
    log.error(s"Job Master $jobMaster terminated before finishing job.")

    val name = failedJob.name
    log.error(s"Job ${name} failed.")
    val nrOfRetries = retries.getOrElse(name, 0)

    if(maxRetries > nrOfRetries) {
     if(nrOfRetries == maxRetries -1) {
      val text = failedJob.text.filterNot(_.contains("FAIL"))
      self.tell(JobRequest(name, text), failedJob.respondTo)
     } else
      self.tell(JobRequest(name, failedJob.text), failedJob.respondTo)

     retries = retries + retries.get(name).map(r=> name ->
                          (r + 1)).getOrElse(name -> 1)
    }
   }
 }
}
```

simulate resolving simulated failure at penultimate retry

re-send job request to self, with the original requestor's address as sender

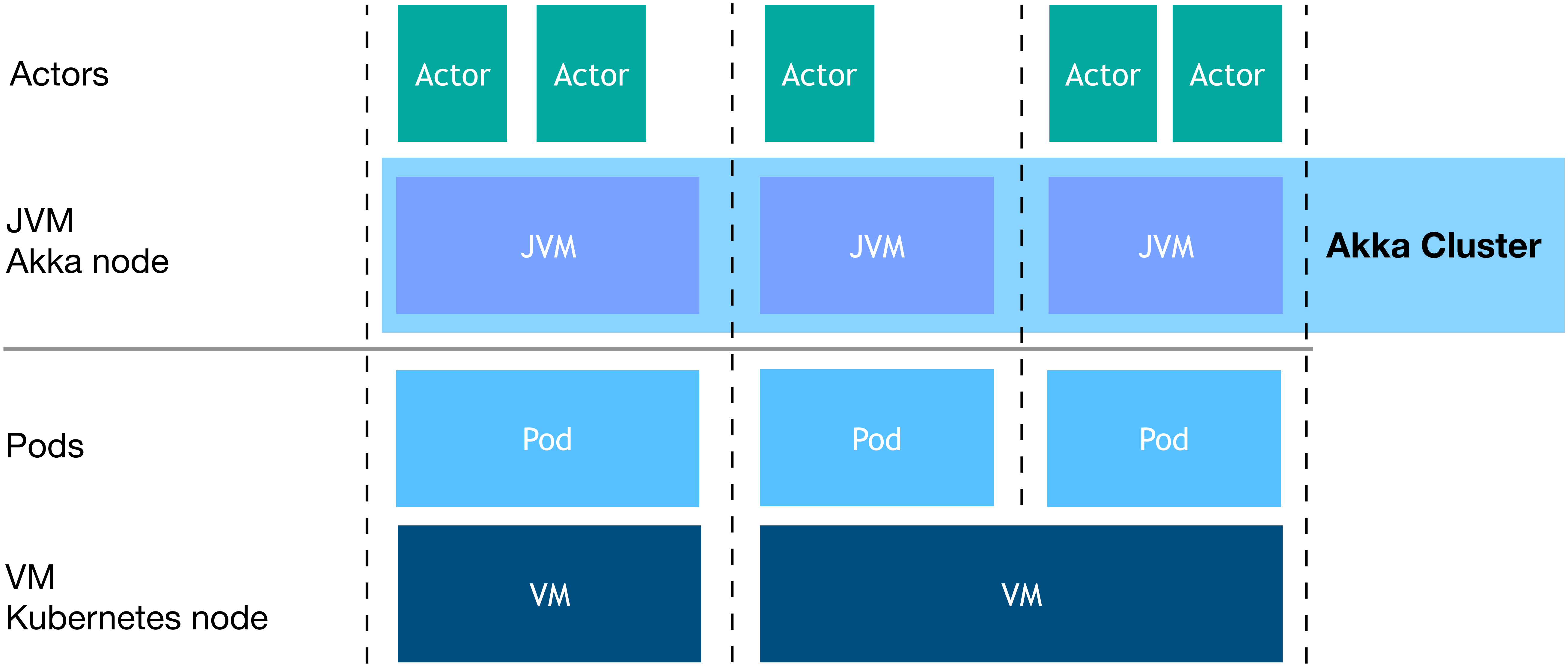
Words cluster: receptionist with resilient jobs

The receptionist creates a new job master when it notices that a master has died (up to x retries).



The job master stops itself when it notices that a job worker has died.

Cluster in the cloud: Akka + Kubernetes



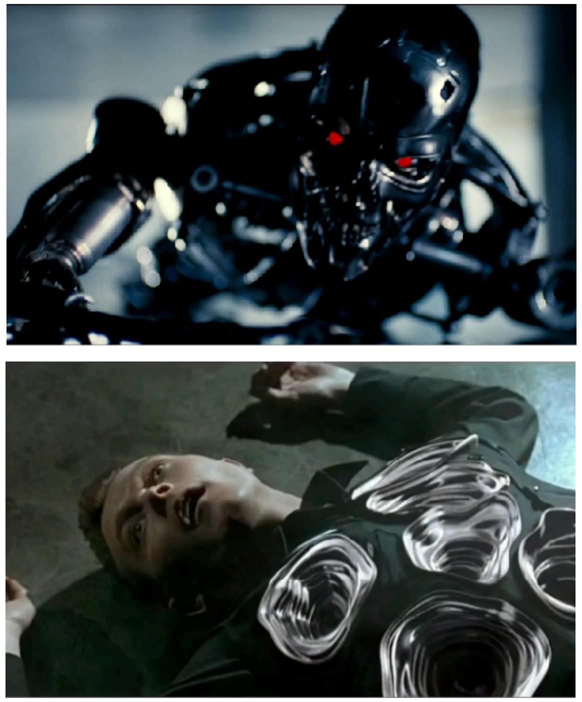
Fabio Triticco 2019: “Scala and Kubernetes: Reactive from Code to Cloud”

https://www.lightbend.com/blog/akka-and-kubernetes-reactive-from-code-to-cloud

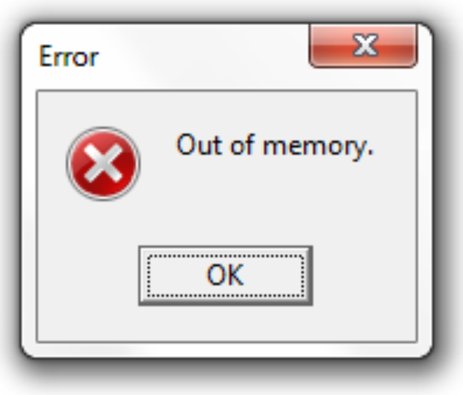
Scale of resilience

{ **Akka** } [**Kubernetes**] aws …

JVM Exceptions



JVM Errors

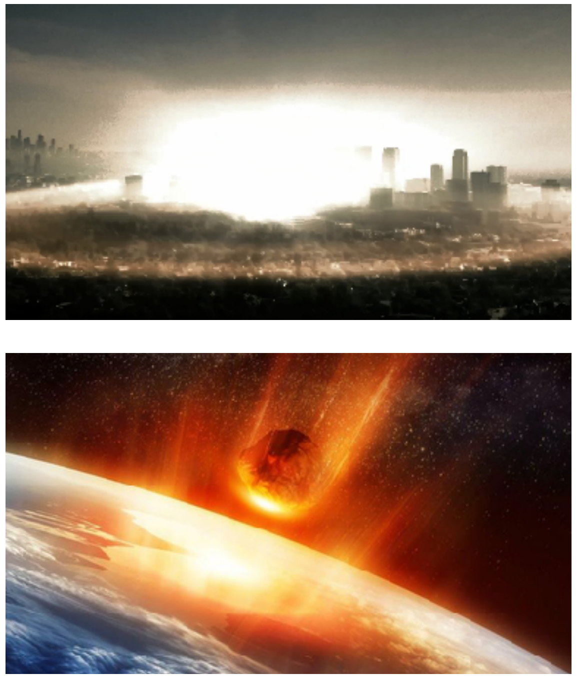


Error

Out of memory.

Hardware failure

Skynet attack
Meteorite hits



Fabio Triticco 2019: “Scala and Kubernetes: Reactive from Code to Cloud”
https://www.lightbend.com/blog/akka-and-kubernetes-reactive-from-code-to-cloud

Take-away 1: programming language matters

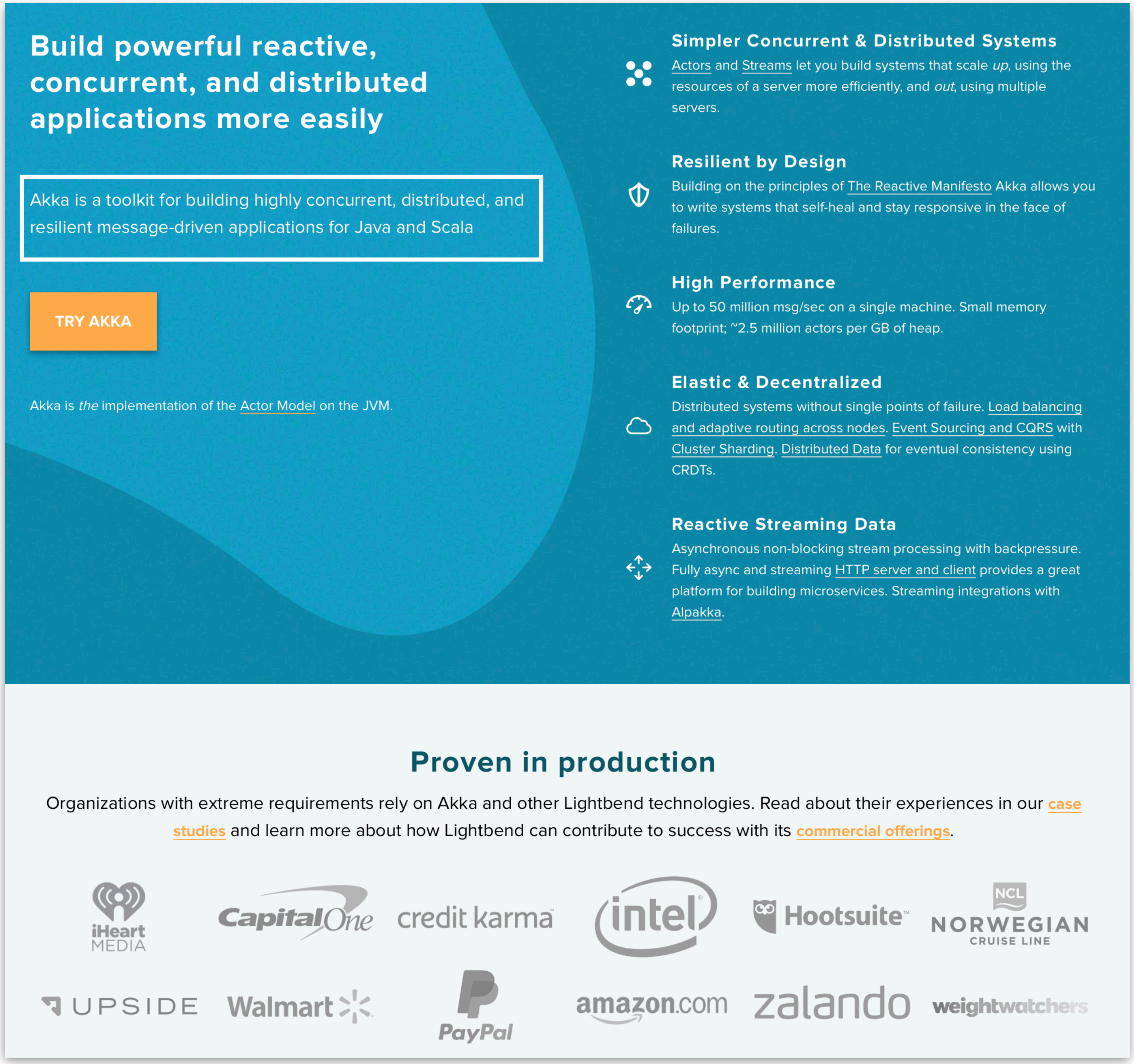
Scala

"Any general-purpose language has to be a scalable language"

released in 2003 by Martin Odersky
professor at EPFL

- Unifies and generalizes functional and object-oriented programming
- Features a strong static type system for safety
- Hosts multiple domain-specific languages
- Offers a read-eval-print loop for interactive prototyping
- Compatible with existing languages for the JVM

Take-away 2: programming model matters



Build powerful reactive, concurrent, and distributed applications more easily

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala

TRY AKKA

Akka is *the* implementation of the Actor Model on the JVM.

Simpler Concurrent & Distributed Systems

Actors and Streams let you build systems that scale *up*, using the resources of a server more efficiently, and *out*, using multiple servers.

Resilient by Design

Building on the principles of The Reactive Manifesto Akka allows you to write systems that self-heal and stay responsive in the face of failures.

High Performance

Up to 50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.

Elastic & Decentralized

Distributed systems without single points of failure. Load balancing and adaptive routing across nodes. Event Sourcing and CQRS with Cluster Sharding. Distributed Data for eventual consistency using CRDTs.

Reactive Streaming Data

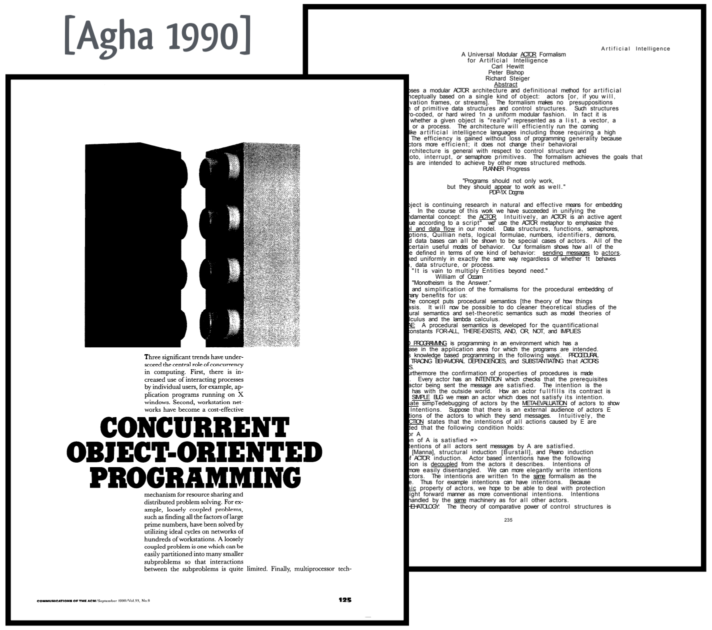
Asynchronous non-blocking stream processing with backpressure. Fully async and streaming HTTP server and client provides a great platform for building microservices. Streaming integrations with Alpakka.

Proven in production

Organizations with extreme requirements rely on Akka and other Lightbend technologies. Read about their experiences in our case studies and learn more about how Lightbend can contribute to success with its commercial offerings.

[Hewitt et al., 1973]

[Agha 1990]



CONCURRENT OBJECT-ORIENTED PROGRAMMING

A Universal Modular ACTOR Formalism for Artificial Intelligence
Carl Hewitt
Peter Bishop
Richard Steiger

actor model

akka

abstractions for concurrent and distributed programming:
strongly-encapsulated, location-transparent, resilient

Take-away 3: architecture matters

patterns for asynchronous messaging

IMPACT

Editor: **Michiel van Genuchten**
MTOnyx
genuchten@ieee.org

Editor: **Les Hatton**
Kingston University
l.hatton@kingston.ac.uk

Twenty Years of Patterns' Impact

Gregor Hohpe, Rebecca Wirfs-Brock, Joseph W. Yoder, and Olaf Zimmermann

This column celebrates the 20th year of software patterns. *IEEE Software* advisory board members teamed up with members of the Hillside Group, a nonprofit organization that promotes the use of patterns and pattern languages, to reflect on the state of the practice and impact of patterns. —*Michiel van Genuchten and Les Hatton*

GOOD ADVICE IN software design is difficult to come by. General design principles can guide us, but reality tends to force trade-offs between seemingly conflicting goals, such as flexibility and maintainability against size and complexity. Likewise, code libraries can go a long way in helping us avoid reinventing the wheel, but the vision of lesser-skilled developers effortlessly wiring together ready-made components remains fiction.

Design patterns have helped narrow this gap by documenting a well-working solution to a problem that occurs repeatedly in a given context. Instead of presenting a copy-and-paste-ready code snippet, patterns discuss forces impacting the solution design. Examples of such forces are performance and security in Web applications: encryption and decryption algorithms improve security but introduce processing overhead. Ward Cunningham once described the best patterns as your older brother teaching you how to do something right.[1]

Although patterns have become popular, their impact as a design technique is more difficult to quantify than the impact of a specific software product (which is what previous installments of this column have examined). This installment highlights both the breadth of patterns available after 20 years of pattern-writing conferences and the depth of impact some patterns have had on open source software.

continued on p. 84

88 **IEEE SOFTWARE** | PUBLISHED BY THE IEEE COMPUTER SOCIETY

0740-7459/13/$31.00 © 2013 IEEE

[Hophe et al., IEEE Software 2013]

INSIGHTS

Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmerman**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

A Decade of *Enterprise Integration Patterns*

A Conversation with the Authors

Olaf Zimmermann, Cesare Pautasso, Gregor Hohpe, and Bobby Woolf

IN AN INDUSTRY that thrives on constant change, few books can survive the test of time. *Enterprise Integration Patterns* (*EIP*)[1]—with its highly influential collection of messaging patterns—is definitely one of those few. So, we interviewed the authors Gregor Hohpe and Bobby Woolf; here, we have the pleasure of sharing their reflections with you. You can discover the inside story of their book project as well as their views on pattern language design and on integration technology's evolution. We also thank them for their precious advice for the next generation of pattern authors and integration solution designers.

> Each pattern represents a decision, so the language walks the reader through the decisions that need to be made.

A General Retrospective

Olaf Zimmermann: How did your book come to be? How did you get together, and how did you find your contributors and reviewers?

Bobby Woolf: Martin Fowler was the matchmaker. When he wrote *Patterns of Enterprise Application Architecture*,[2] Kyle Brown pointed out that his pattern language was not addressing asynchronous messaging. Martin felt that he already had plenty of patterns to write, which motivated Kyle and me to submit a collection of 27 patterns to the 2002 Pattern Language of Programs [PLoP] conference under the title, "Patterns of System Integration with Enterprise Messaging."[3]

Gregor Hohpe: I was based some 3,000 miles away, using enterprise application integration [EAI] tools, such as TIBCO and Vitria, in my consulting job. I had a nagging feeling that these tools share underlying concepts, which are obfuscated by different terminology. Martin encouraged me to document my findings in the form of patterns, also to be submitted to PLoP 2002,[4] where I first met Bobby and Kyle.

Bobby: So Kyle brought me into the effort he'd started with Martin, then Martin brought Gregor in. While Martin and Kyle contributed a lot of material and guidance, they eventually lessened their involvement, leaving Gregor and me to write and complete the book. Gregor and I hadn't known each other before, so it was a crash getting-to-know-you opportunity.

With encouragement from Martin and Kyle, we decided to combine our papers with the goal to turn them into a book. While there was some merging to be done, the two papers complemented each other well. We only had a handful of patterns that overlapped: Kyle's and my paper described message patterns ("message construction" in the book) and message client patterns (later "messaging endpoints").

Gregor: Coming from the EAI perspective, my 17 patterns focused on what was "between" the endpoints: message routing, message transformation, and message management. It also contained an early version of the pattern icons.

0740-7459/16/$33.00 © 2016 IEEE

JANUARY/FEBRUARY 2016 | **IEEE SOFTWARE** 13

[Zimmermann et al., IEEE Software 2016]

Take-away 4: application-level before infrastructure-level

Akka has a cloud-native programming model, ready to scale from day 1

It enables transparent communication between different nodes of a service

Resilience is *built in* your service with granular control

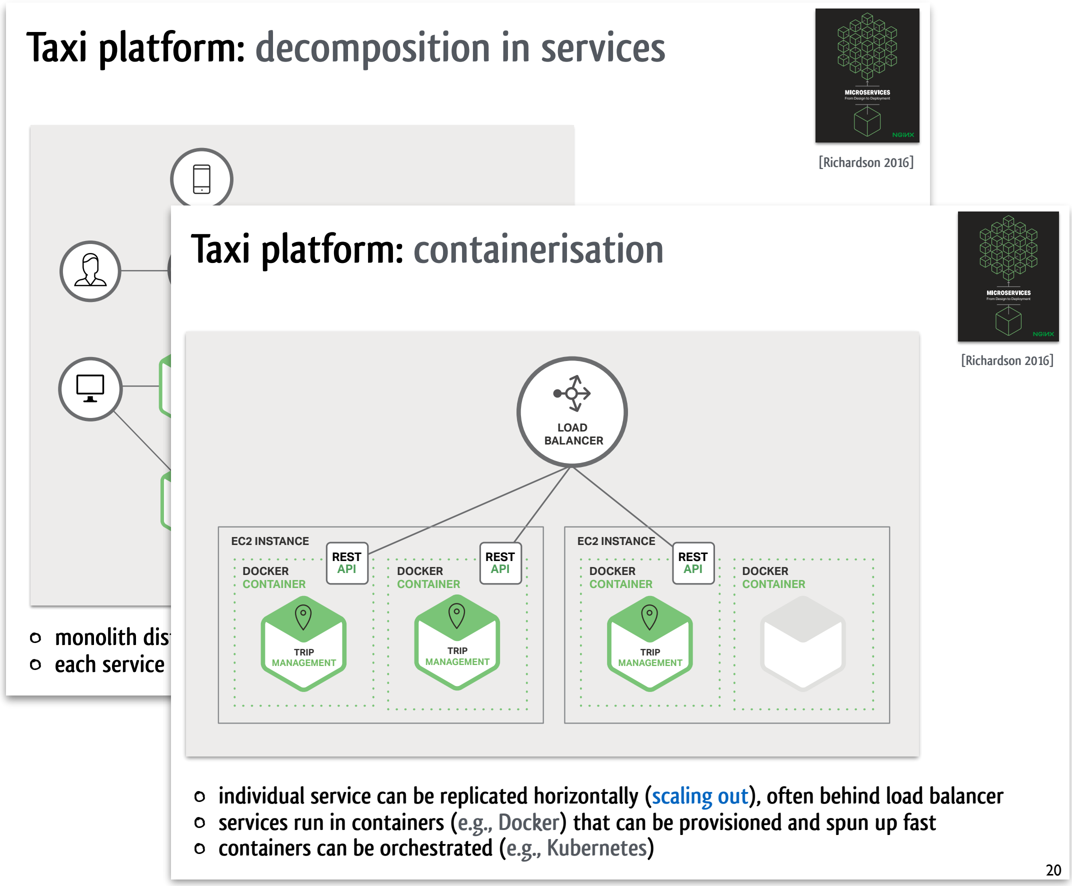
Kubernetes is a great infrastructure choice for your clustered application

It provides location transparency with cluster formation

It introduces resilience at an infrastructure level

Fabio Triticco 2019: “Scala and Kubernetes: Reactive from Code to Cloud”

https://www.lightbend.com/blog/akka-and-kubernetes-reactive-from-code-to-cloud



Taxi platform: decomposition in services

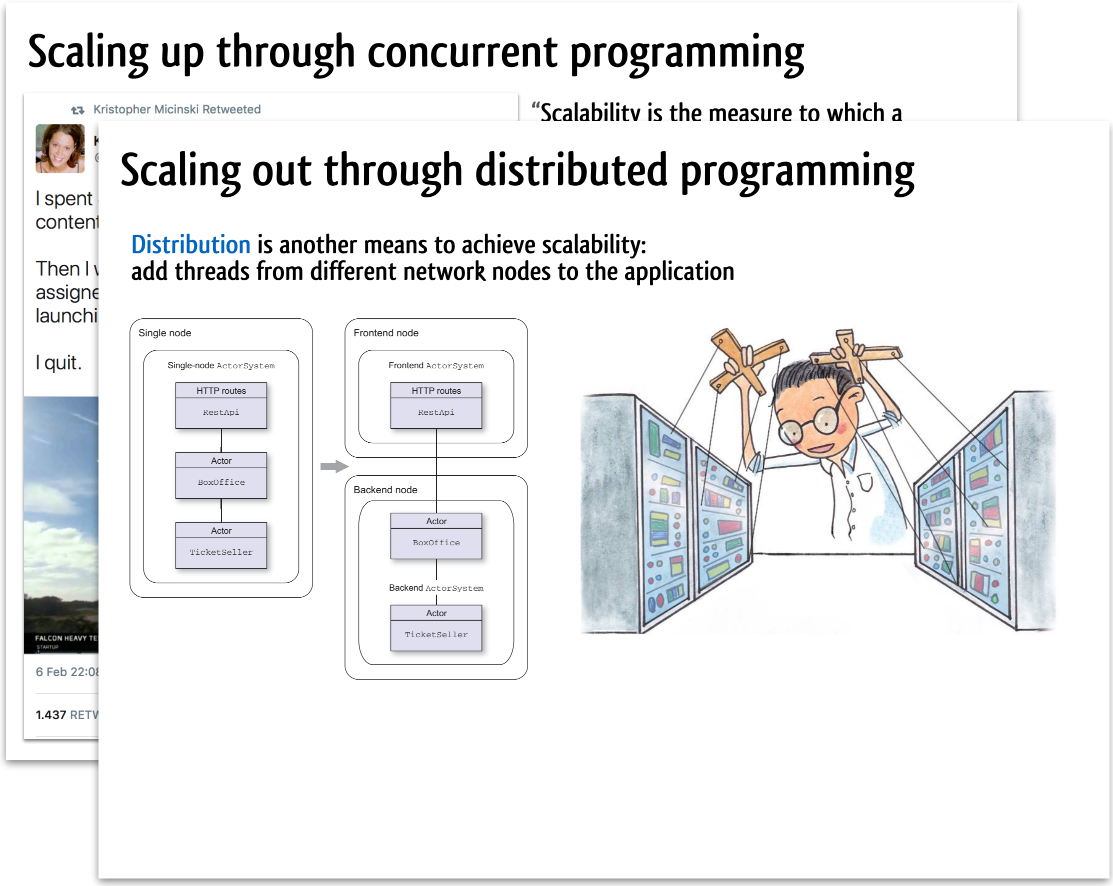
[Richardson 2016]

- monolith dis
- each service

Taxi platform: containerisation

[Richardson 2016]

- individual service can be replicated horizontally (scaling out), often behind load balancer
- services run in containers (e.g., Docker) that can be provisioned and spun up fast
- containers can be orchestrated (e.g., Kubernetes)



Scaling up through concurrent programming

Kristopher Micinski Retweeted

I spent
content

Then I v
assigne
launchi

I quit.

6 Feb 22:0

1.437 RETV

"Scalability is the measure to which a

Scaling out through distributed programming

Distribution is another means to achieve scalability:
add threads from different network nodes to the application

Take-away 1: programming language matters

- U
- Fe
- H
- O
- C

Take-away 2: programming model matters

Build powerful reactive, concurrent, and distributed applications more easily

Simpler Concurrent & Distributed Systems

Actors and Streams let you build systems that scale *up*, using the resources of a server more efficiently, and *out*, using multiple servers.

Akka is a toolkit for buildi
resilient message-driven

TRY AKKA

Akka is *the* implementation of th

Organizations with extr
studies

[Hewitt

[Agha 1990]

abstractio
strongly-e

Take-away 3: architecture matters

patterns

Take-away 4: application-level be

Akka has a cloud-native programming model, ready to scale from day 1

It enabl
commun
different n

Kubernetes is a great infrastructure choice for your clustered application

It prov
transp
clust

Fabio Triticco 2019: "Scala and Kubernetes:
https://www.lightbend.com/blog/akka-and-kube