delaware

Software Engineering Study Trip Berlin 2020

Scaling micro-service architectures up and out: a slightly opinionated review

VUB SOFTWARE LANGUAGES LAB

Coen De Roover
cderoove@vub.ac.be

Brussels

Dutch
6.23 million

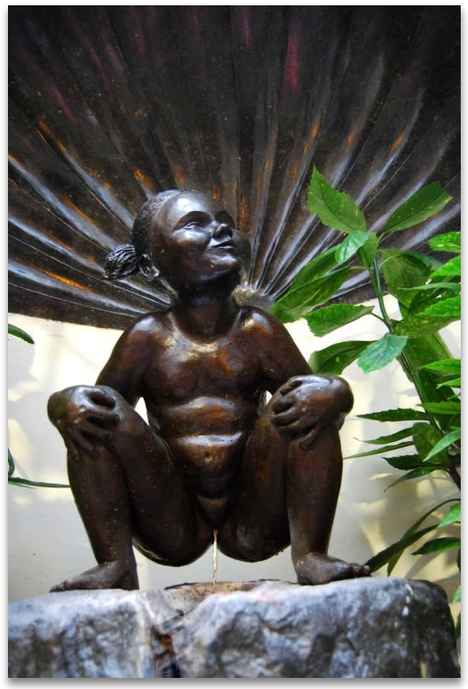
French
3.32 million

German
0.07 million

Sights

Grote Markt - Grand Place

Manneken pis



Jeanneke pis

Atomium

Zinneke pis

Food

ch Fries

Waffles (from Brussels)

Waffles (from Liège)

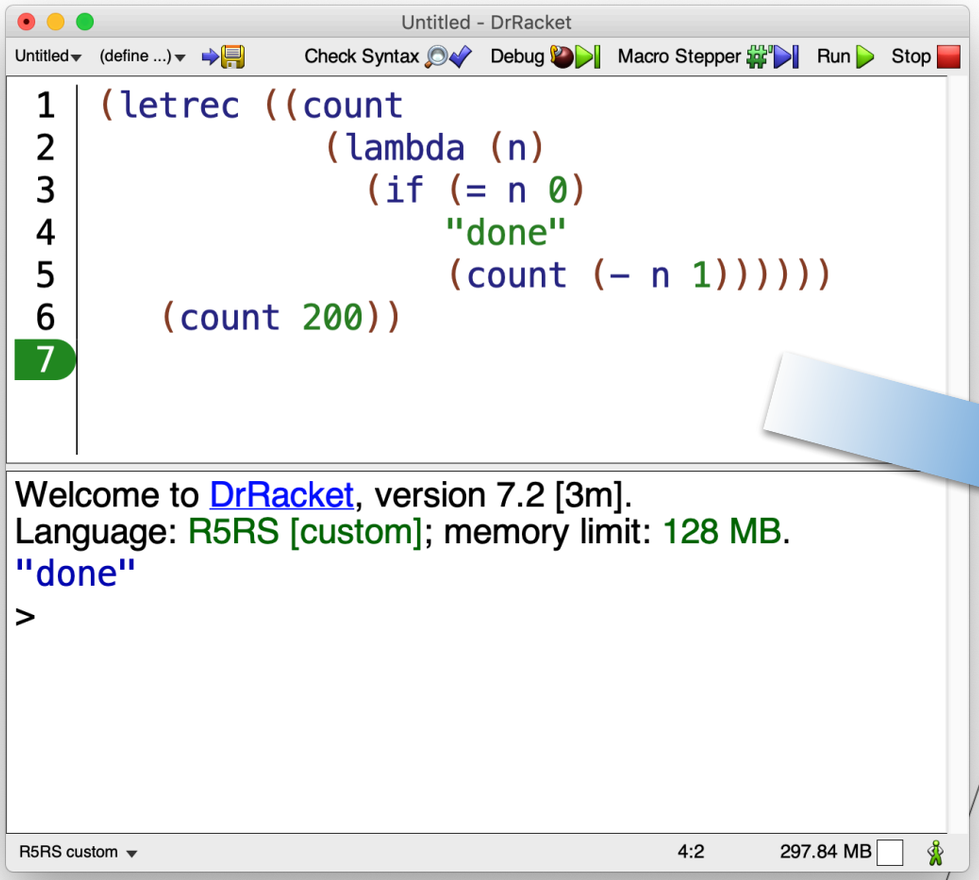
Software Languages Lab @ Vrije Universiteit Brussel

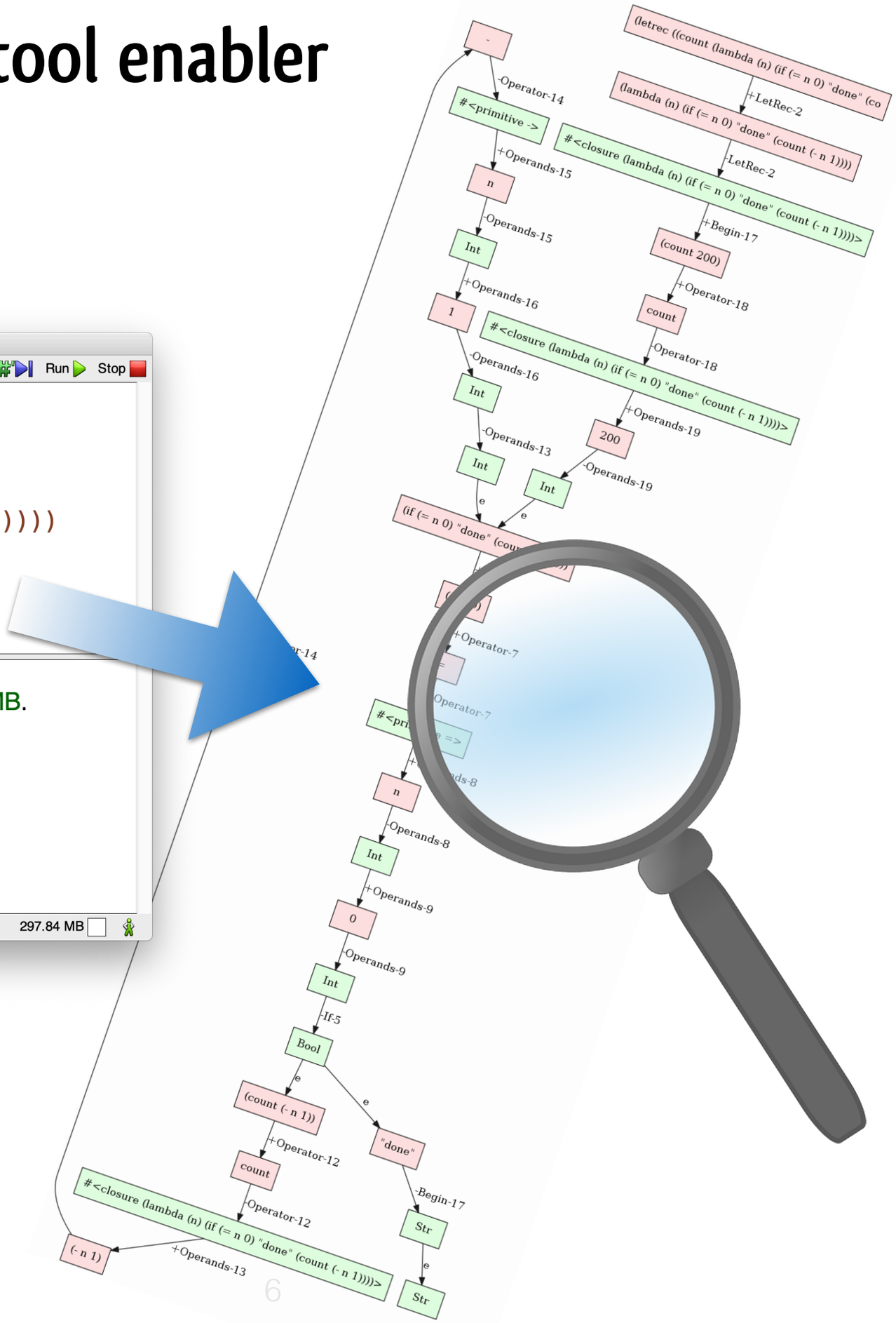
languages

tools

Design, implement, and formalize
programming languages, frameworks, and tools
for developing future software applications
in a less effort-intensive and less error-prone manner.

Program analysis as tool enabler



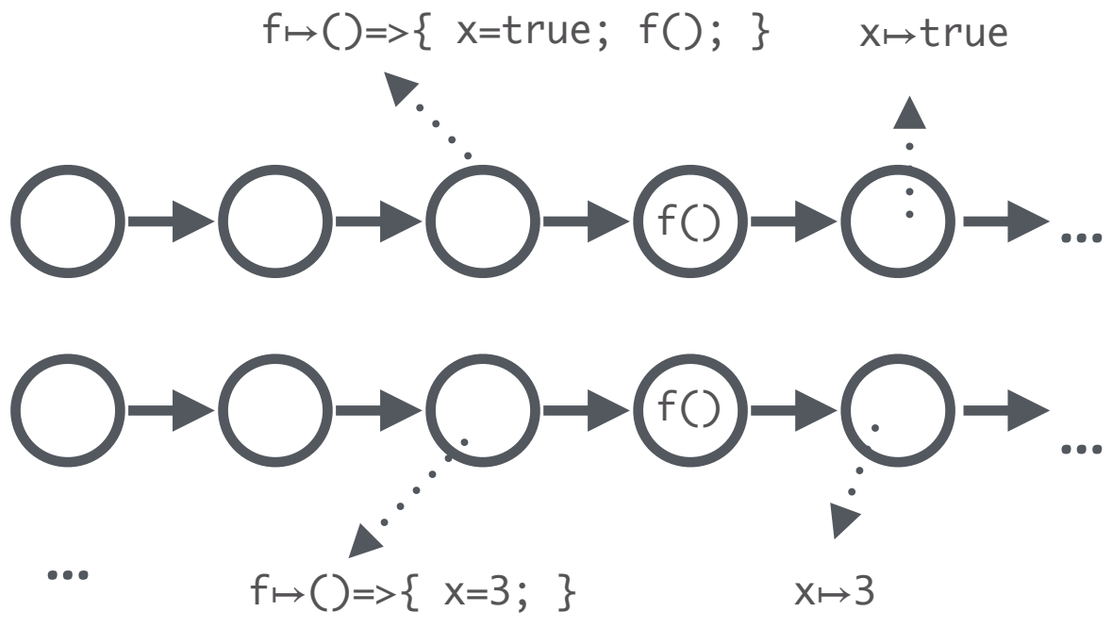


Program analysis as tool enabler

concrete interpretation

$$
\begin{aligned}
\varsigma_{\text{CESK}} \in \Sigma_{\text{CESK}} &= \mathit{Control} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \\
\mathit{Control} &= \mathit{Exp} + \mathit{Val} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Val} \\
\mathit{val} \in \mathit{Val} &= \mathit{Clo} + \mathit{Kont} \\
\kappa \in \mathit{Kont} &::= \mathbf{halt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(\mathit{clo}, a) \\
\mathit{clo} \in \mathit{Clo} &::= (\lambda v.e) \times \mathit{Env} \\
a, b \in \mathit{Addr} &\quad \text{an infinite set of addresses}
\end{aligned}
$$

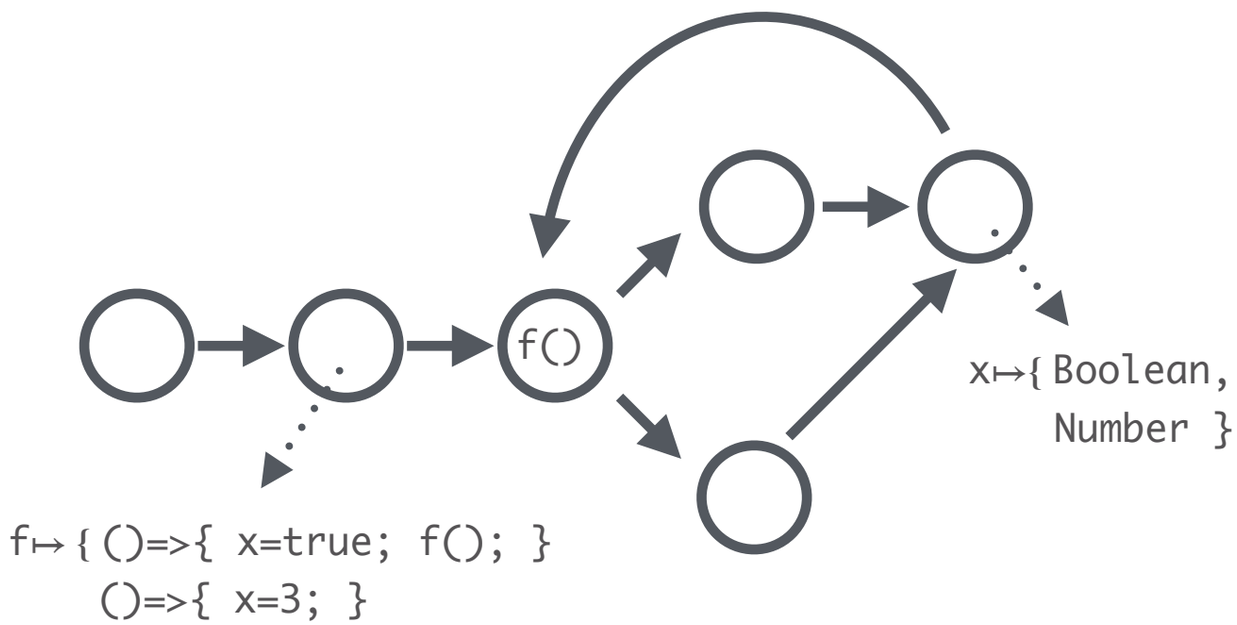
$$
\begin{aligned}
\langle v, \rho, \sigma, a, t\rangle &\to \langle \sigma(\rho(v)), \rho, \sigma, a, u\rangle. \\
\langle (\lambda v.e), \rho, \sigma, a, t\rangle &\to \langle ((\lambda v.e), \rho), \rho, \sigma, a, u\rangle. \\
\langle (e_0\ e_1), \rho, \sigma, a, t\rangle &\to \langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u\rangle. \\
\langle \mathit{clo}, \rho, \sigma, a, t\rangle &\to \langle e, \rho', \sigma[b \mapsto \mathbf{fn}(\mathit{clo}, a')], b, u\rangle \quad \text{if } \kappa = \mathbf{ar}(e, \rho', a'), \\
\langle \mathit{val}, \rho, \sigma, a, t\rangle &\to \langle e, \rho'[v \mapsto b], \sigma[b \mapsto \mathit{val}], a', u\rangle \quad \text{if } \kappa = \mathbf{fn}((\lambda v.e), \rho'), a').
\end{aligned}
$$



abstract interpretation

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

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



Example program transformation tool


```
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

Plug-in Development - /Users/cderoove/git/damp.ekeko.snippets/damp.ekeko.snippets.plugin.test/resources/EkekoX-Specifications/scam_demo

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

Pack | Plug-i

> TestCase-JDT-CompositeVisitor
> TestCase-TypeParameters Ekeko
src
be.ac.chaq.change
be.ac.chaq.model.ast.java
AbstractTypeDeclaration.ja
Annotation.java
AnnotationTypeDeclaration
AnnotationTypeMemberDec
AnonymousClassDeclaratic
ArrayAccess.java
ArrayCreation.java
ArrayInitializer.java
ArrayType.java
AssertStatement.java
Assignment.java
ASTIdentifier.java
ASTNode.java
Block.java
BlockComment.java
BodyDeclaration.java
BooleanLiteral.java
BreakStatement.java
CastExpression.java
CatchClause.java
CharacterLiteral.java
ClassInstanceCreation.java
Comment.java
CompilationUnit.java
ConditionalExpression.java
ConstructorInvocation.java
ContinueStatement.java
DoStatement.java
EmptyStatement.java

scam_demo3.ekx

```
?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
```

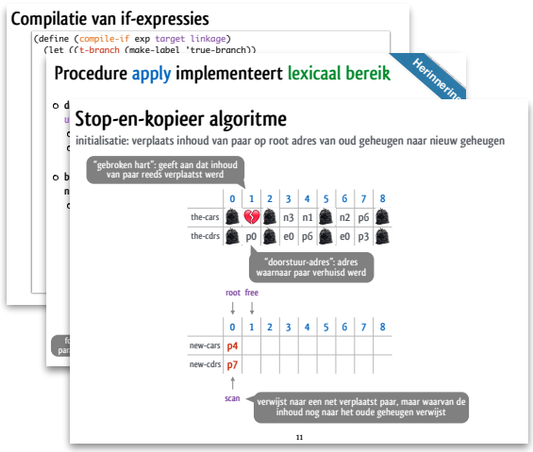
Overview | Search Templates | »1

ArrayCreation.java

```
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;
```

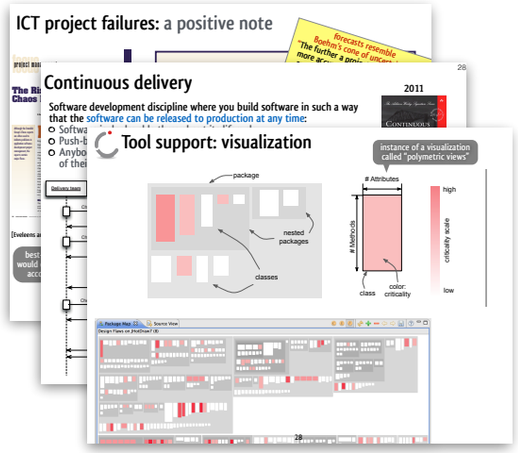
Console | Ekeko Query Results

Current courses



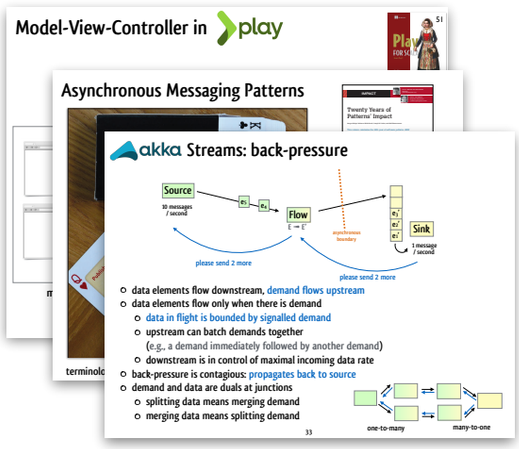
Interpretation I

meta-interpreters,
compilers,
garbage collection



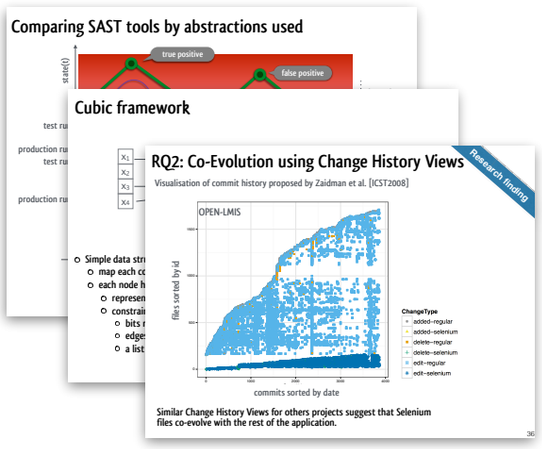
Software Engineering

continuous delivery,
design patterns
design & test metrics



Software Architecture

architectural patterns
message-based systems,
reactive architectures



Software Quality

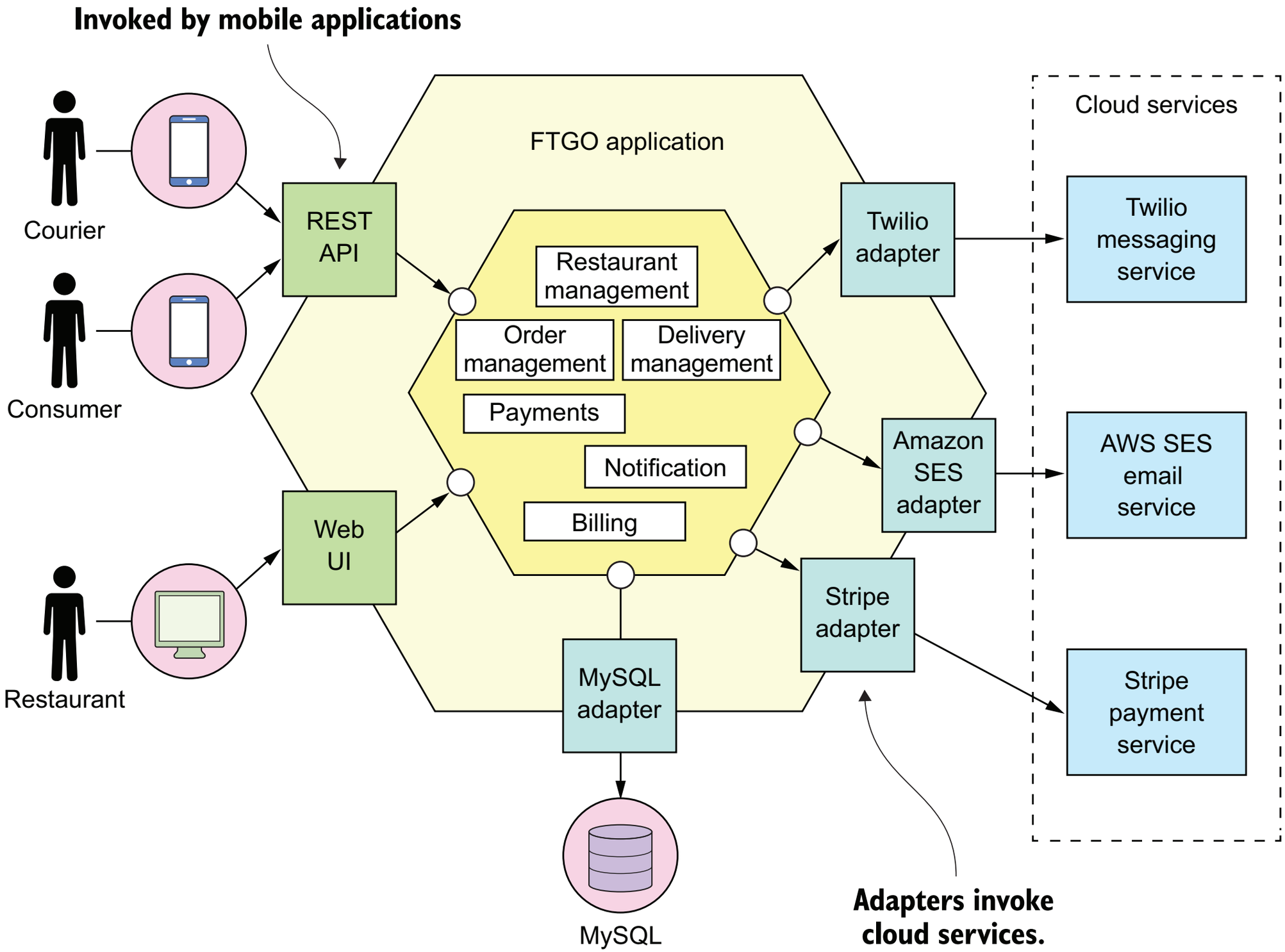
mining software repositories,
program analysis,
automated testing

μ-services architecture
introduction and motivation

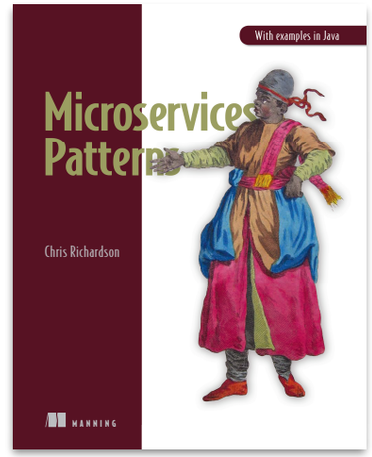
Scaling up
using concurrent actors

Scaling out
using distributed actors

FTGO monolithic architecture

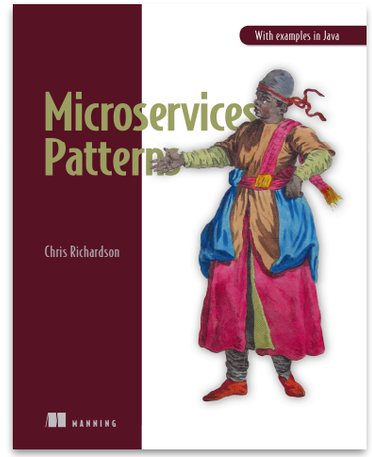


one large, but nicely modularised application

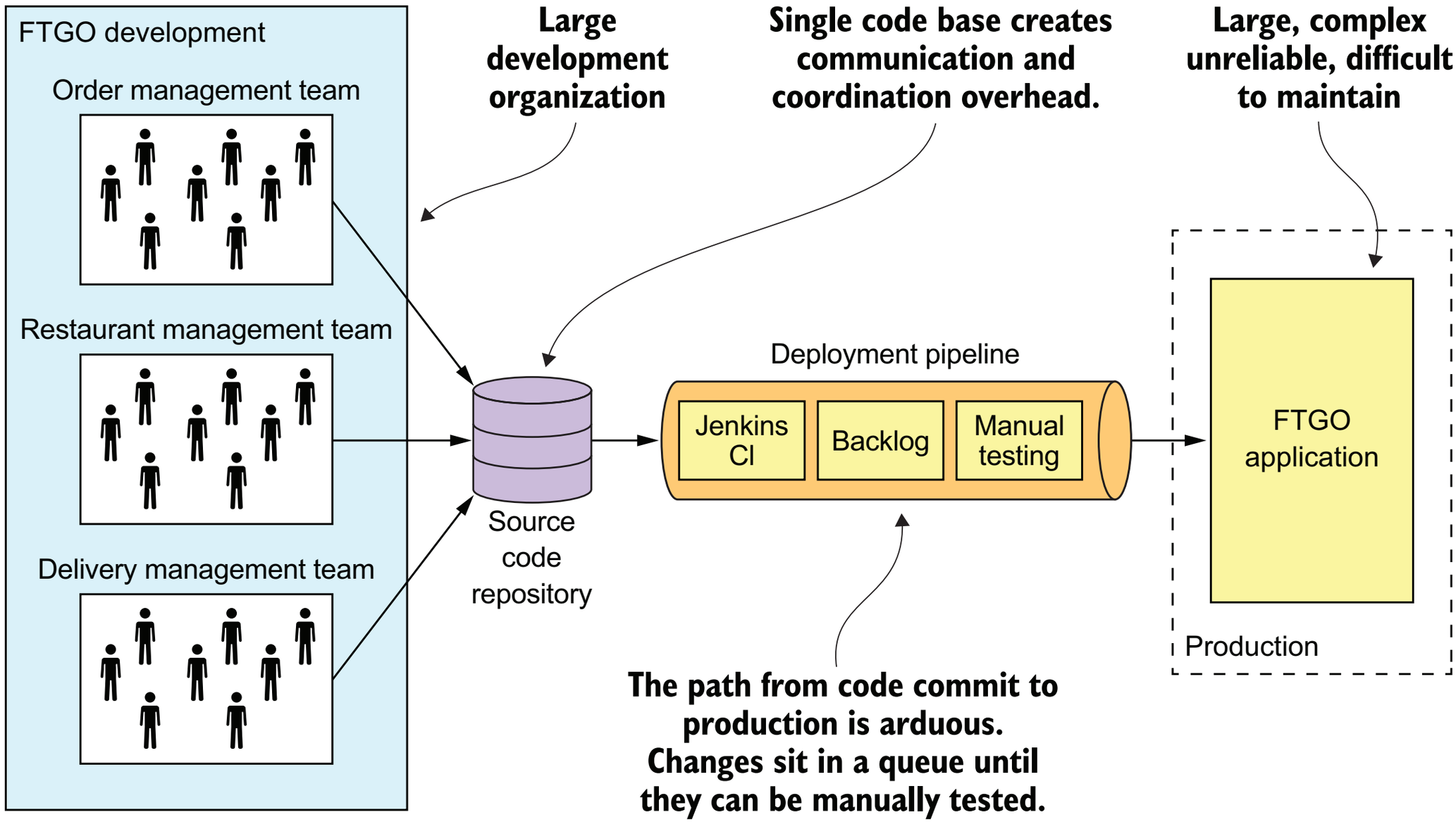


[Richardson 2019]

FTGO monolithic: development process

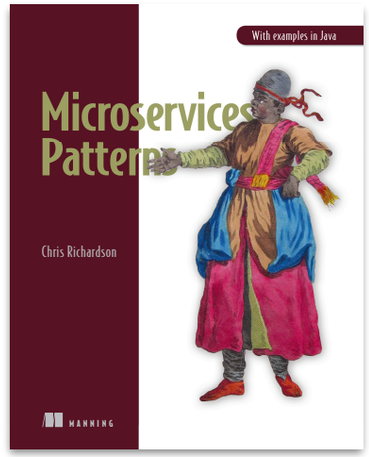


[Richardson 2019]

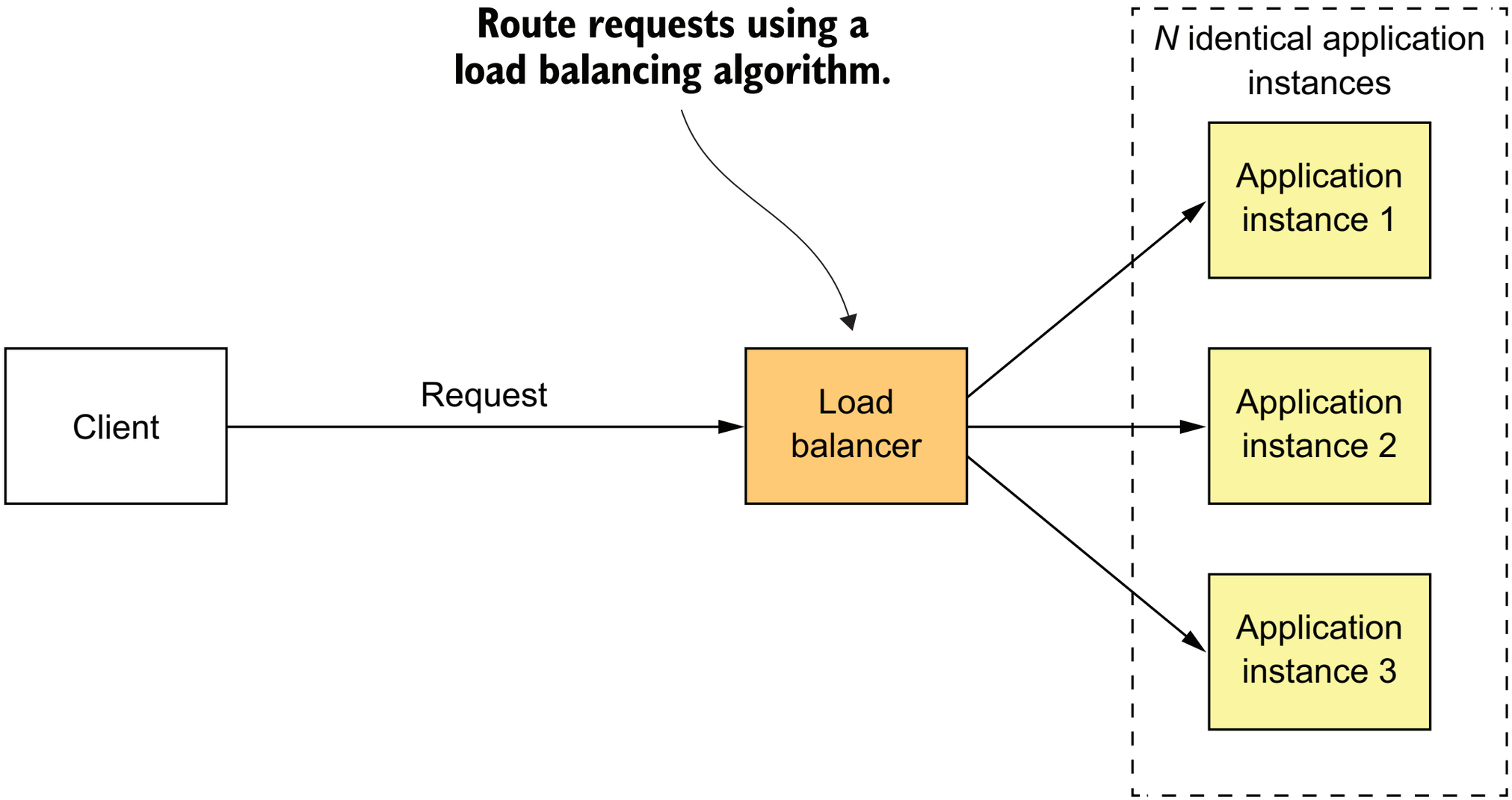


entire executable needs to be redeployed entirely upon smallest change

FTGO monolithic: scaling options

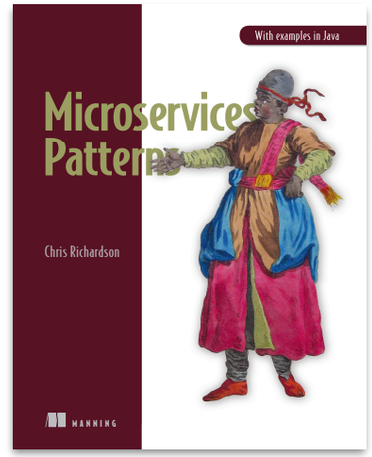


[Richardson 2019]

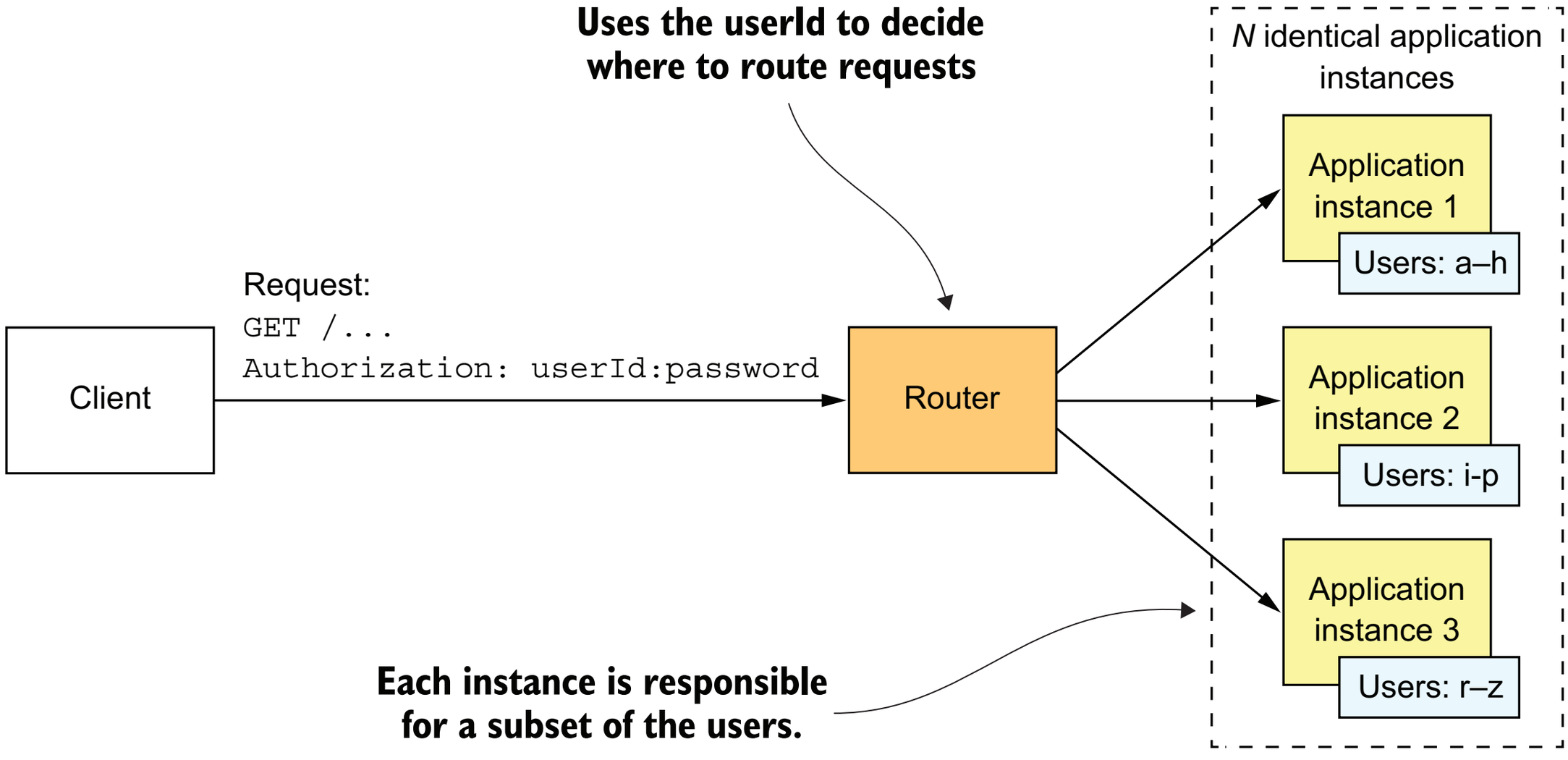


uniform replication
(X-axis scaling)

FTGO monolithic: scaling options

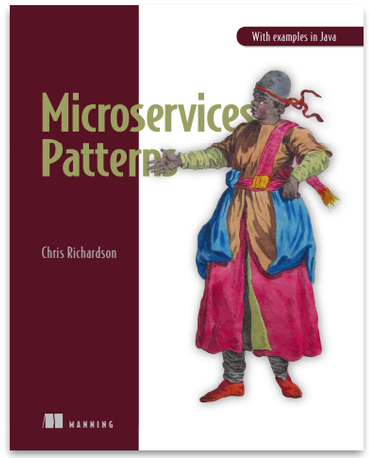


[Richardson 2019]

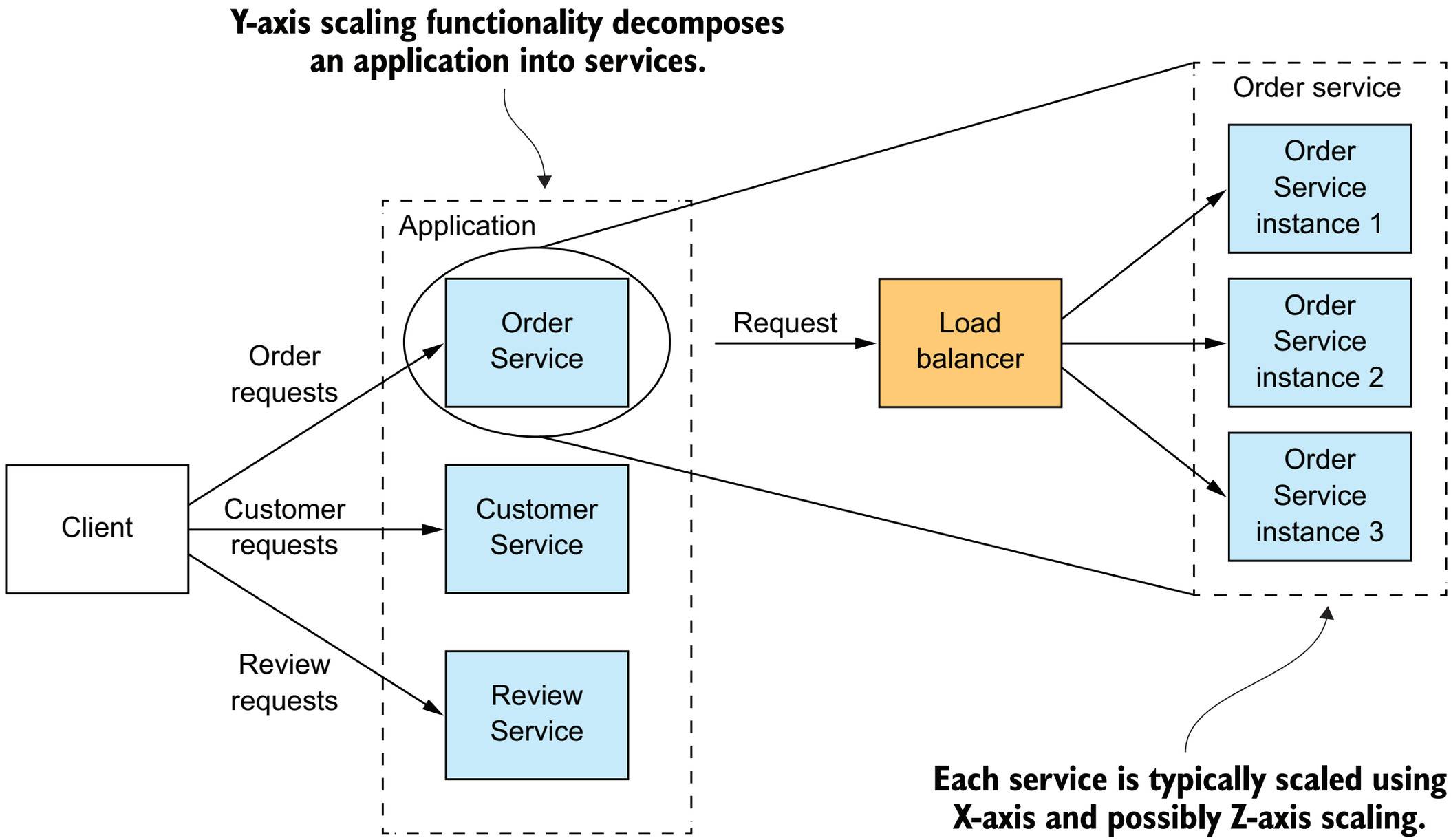


shard-based replication
(Z-axis scaling)

FTGO μ-services: scaling options

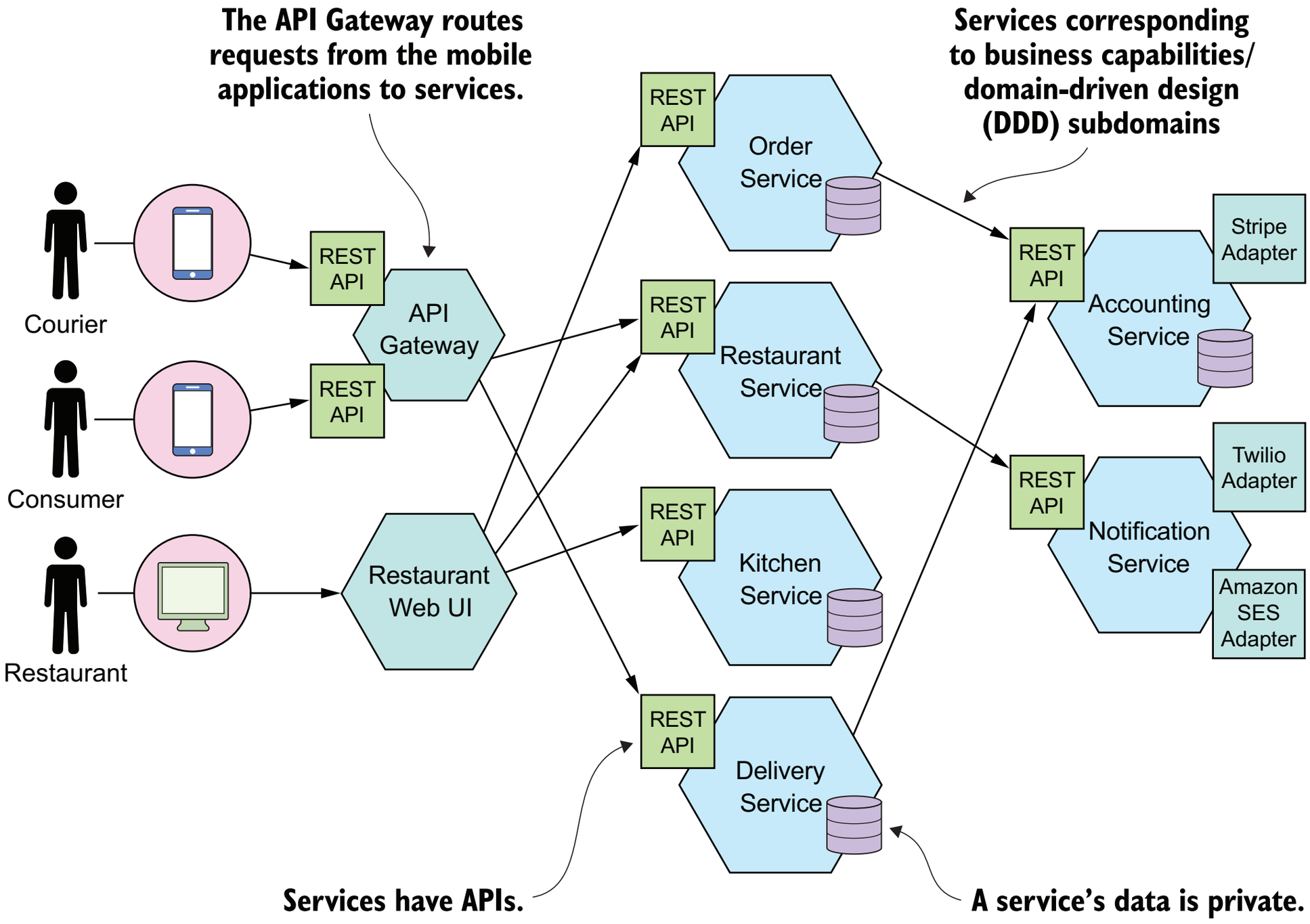


[Richardson 2019]

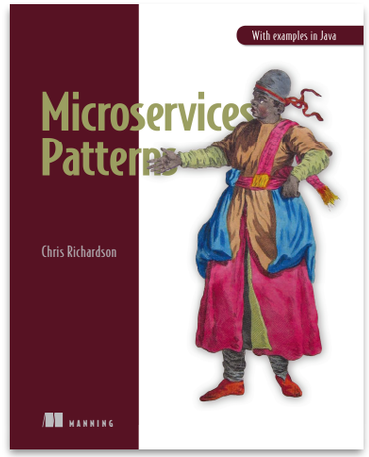


each service can be scaled independently

FTGO μ-services architecture

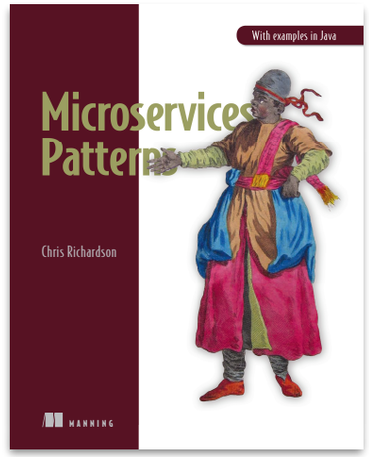


- monolith distributed vertically into services that are deployed independently
- each service provides and consumes functionality as a mini-application on its own

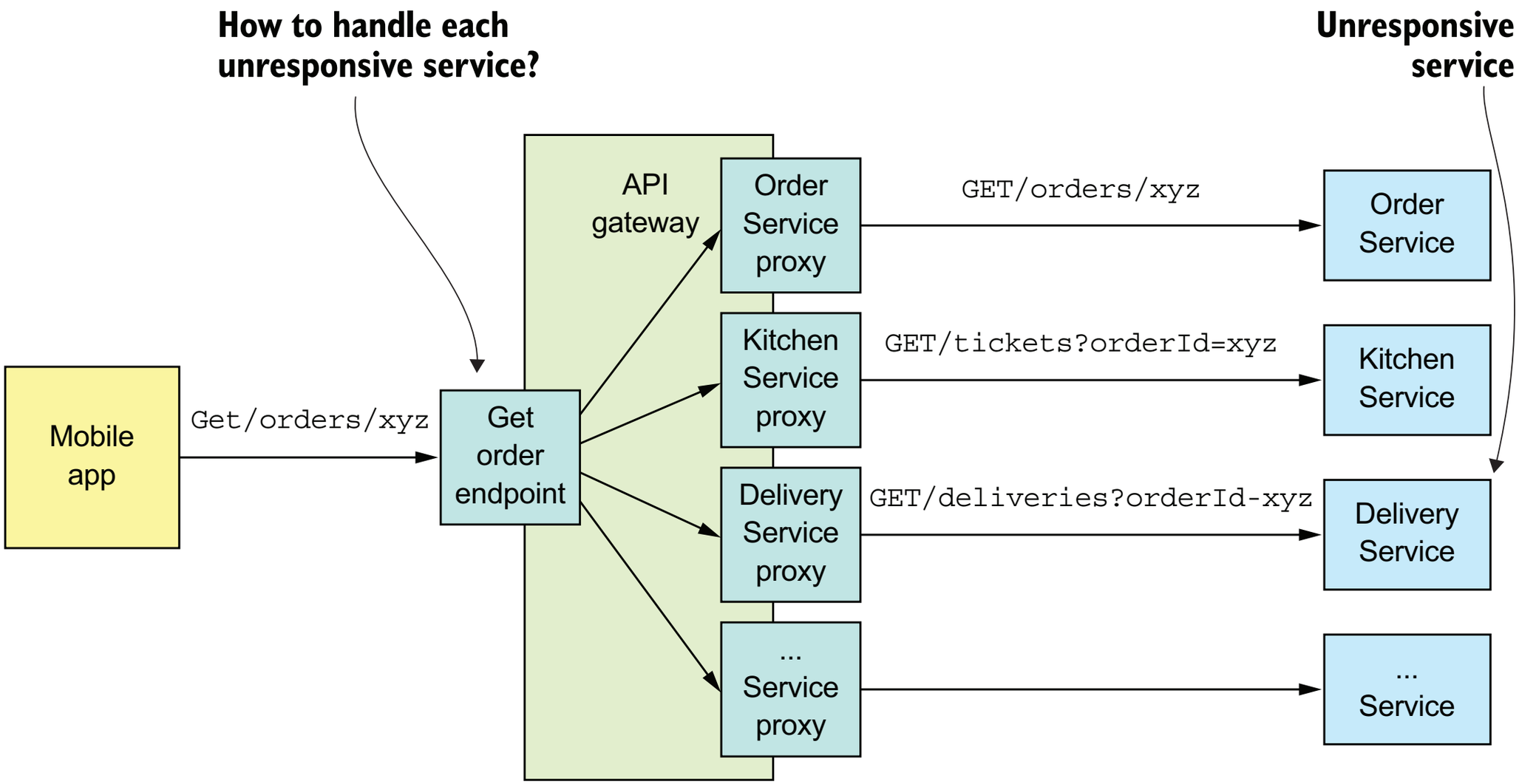


[Richardson 2019]

FTGO µ-services: REST calls (1/2)

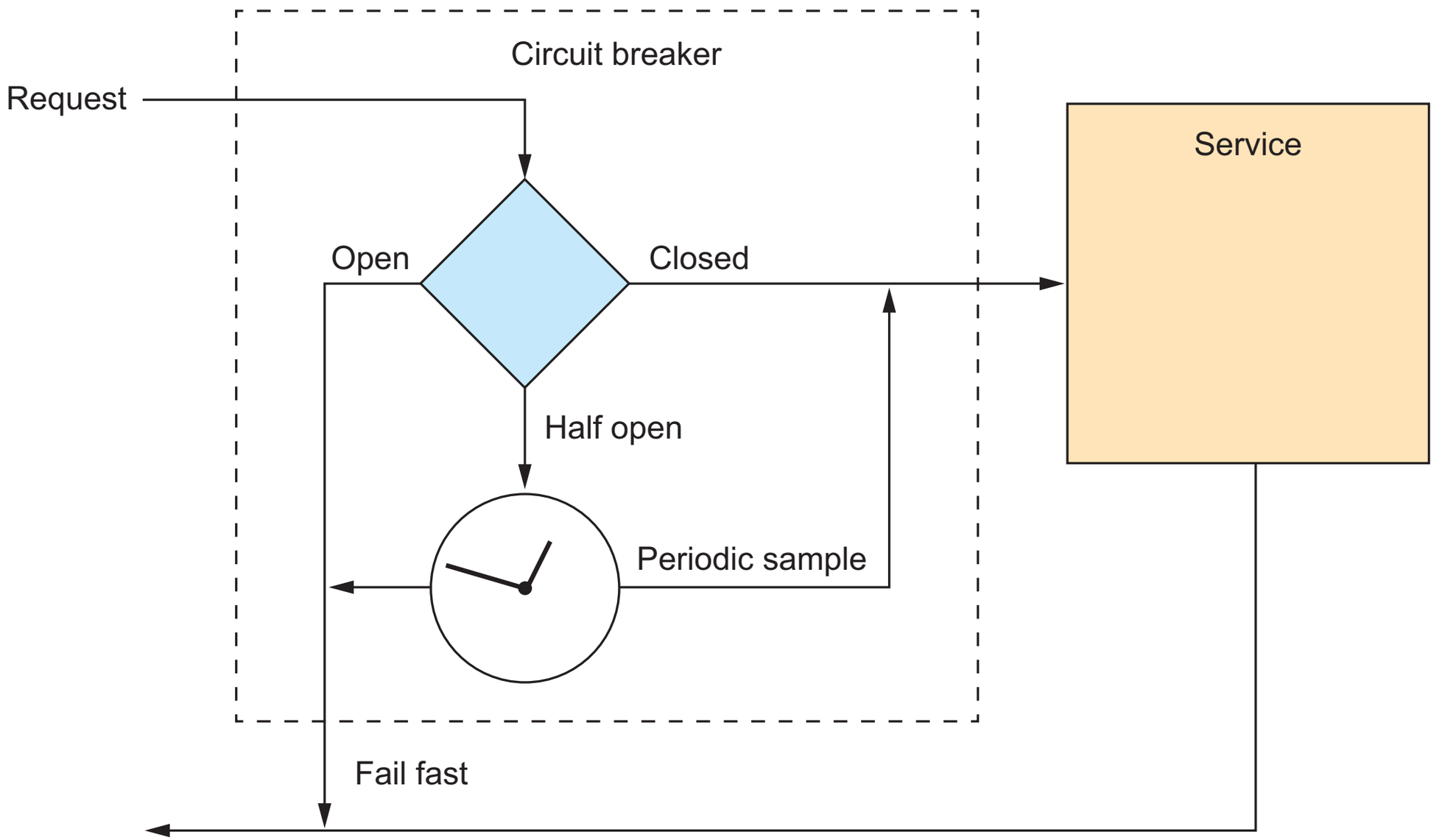


[Richardson 2019]



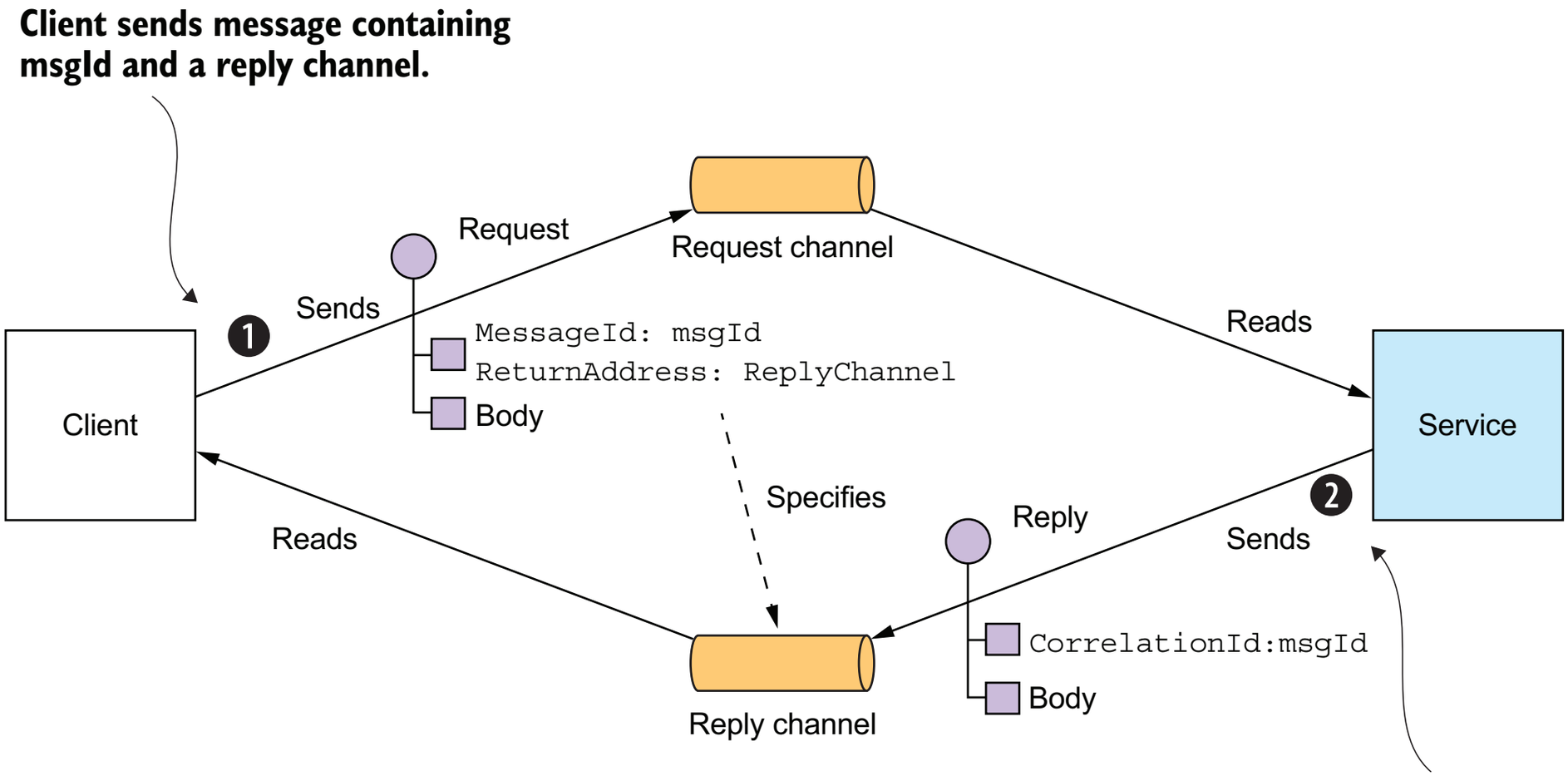
- simple and familiar, synchronous request/response cycle of HTTP
  - exposes business objects as resources at a URI
  - four primary HTTP operations on those resources: POST, GET, PUT, DELETE
  - not prone to fallacy of transparent distribution

FTGO µ-services: REST calls (2/2)

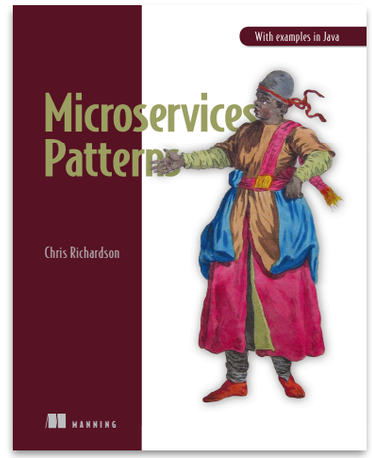


blocking calls require protection against unresponsive services to prevent cascading failures

FTGO μ-services: messaging (1/2)



- asynchronicity
  - sender does not have to wait for the receiver to receive and process the message
  - requires a send-and-forget approach to communication
- variable timing
  - messaging system queues up requests until the receiver is ready to process them
  - enables sender and receiver to produce and consume messages at their own pace



[Richardson 2019]

FTGO µ-services: messaging (2/2)

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

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.

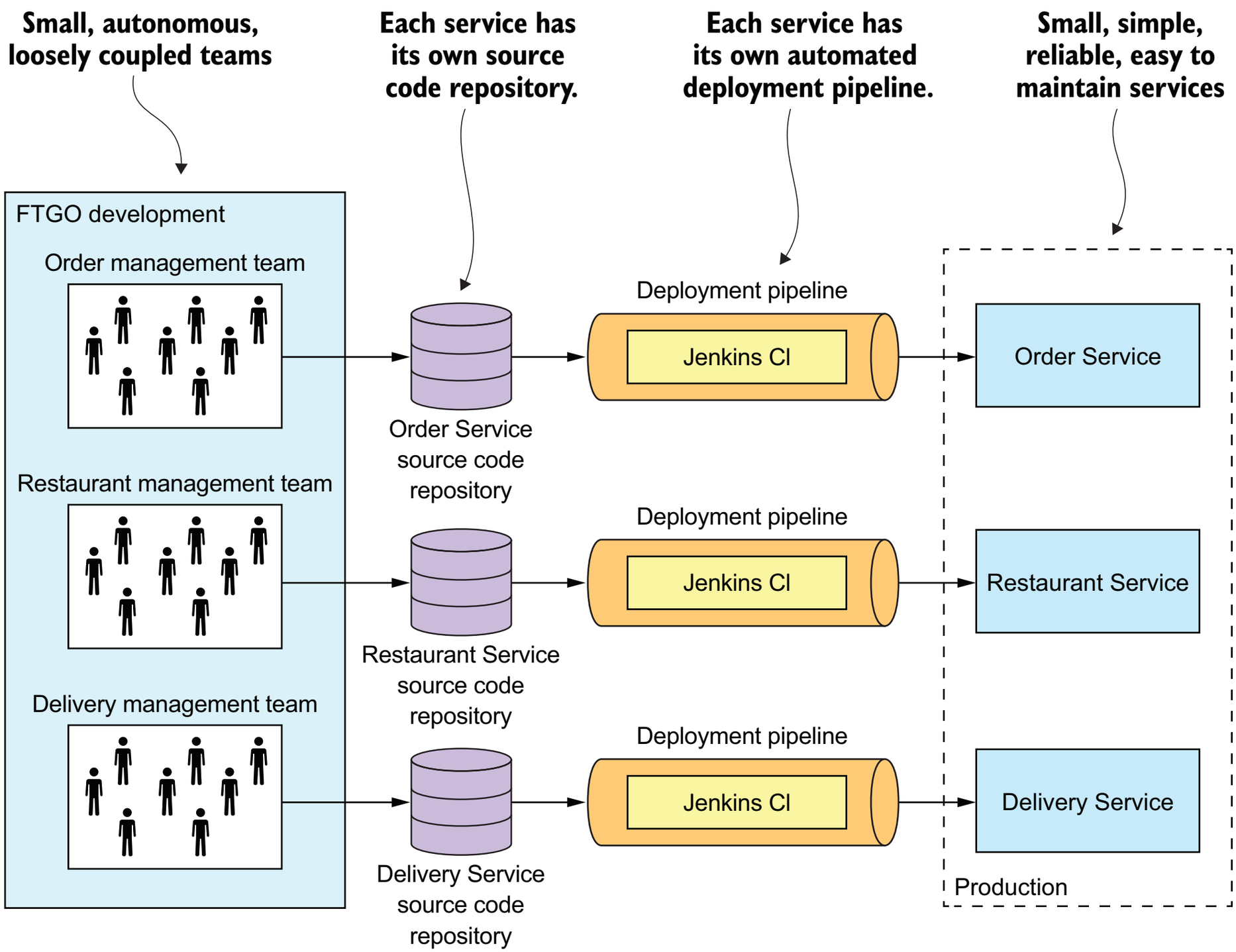
With encouragement from Martin and Kyle, we decided to combine our papers with the goal to turn them into a book. While there was some merging to be done, the two papers complemented each other well. We only had a handful of patterns that overlapped: Kyle's and my paper described message patterns ("message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

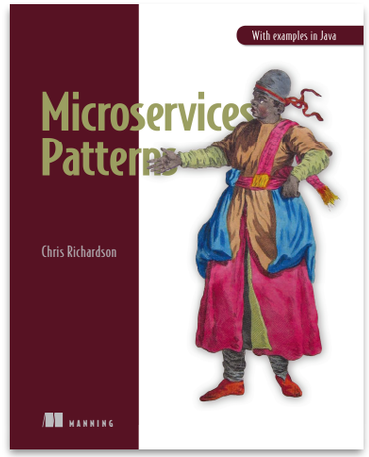
Gregor: Coming from the EAI perspective, my 17 patterns focused on what was "between" the endpoints: message routing, message transformation, and message management. It also contained an early version of the pattern icons.

0740-7459/16/$33.00 © 2016 IEEE

[Zimmermann et al., IEEE Software 2016]

FTGO μ-services: development process





[Richardson 2019]

- each team develops, tests, and deploys their services independently

FTGO µ-services: development process

Melvin Conway

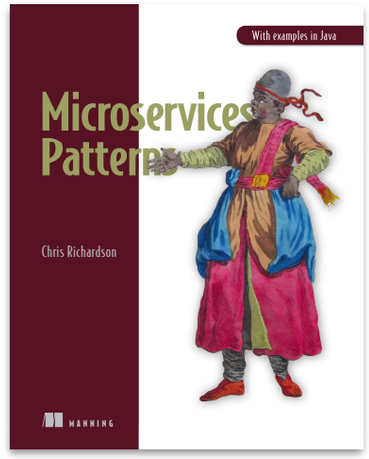
In 1967 I submitted a paper called "How Do Committees Invent?" to the *Harvard Business Review*. *HBR* rejected it on the grounds that I had not proved my thesis. I then submitted it to *Datamation*, the major IT magazine at that time, which published it April 1968. The text of the paper is here.

Here is one form of the paper's thesis:

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

reverse application: organize teams so that they mirror the ideal architecture

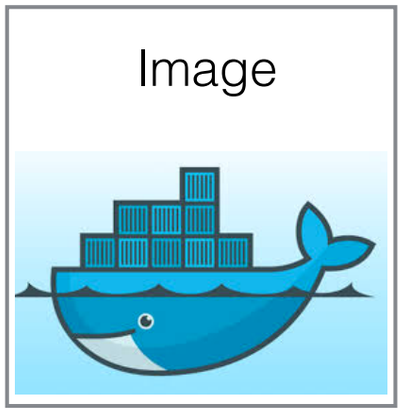
FTGO µ-services: containerization



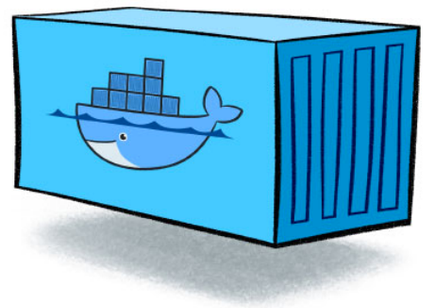
[Richardson 2019]

```
# 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"]
```

build



run



Dockerfile

Docker Image

Docker Container

```
1  FROM openjdk:8u171-jre-alpine
2  RUN apk --no-cache add curl
3  CMD java ${JAVA_OPTS} -jar ftgo-restaurant-service.jar
4  HEALTHCHECK --start-period=30s --interval=5s CMD curl -f http://localhost:8080/actuator/health || exit 1
5  COPY build/libs/ftgo-restaurant-service.jar .
```

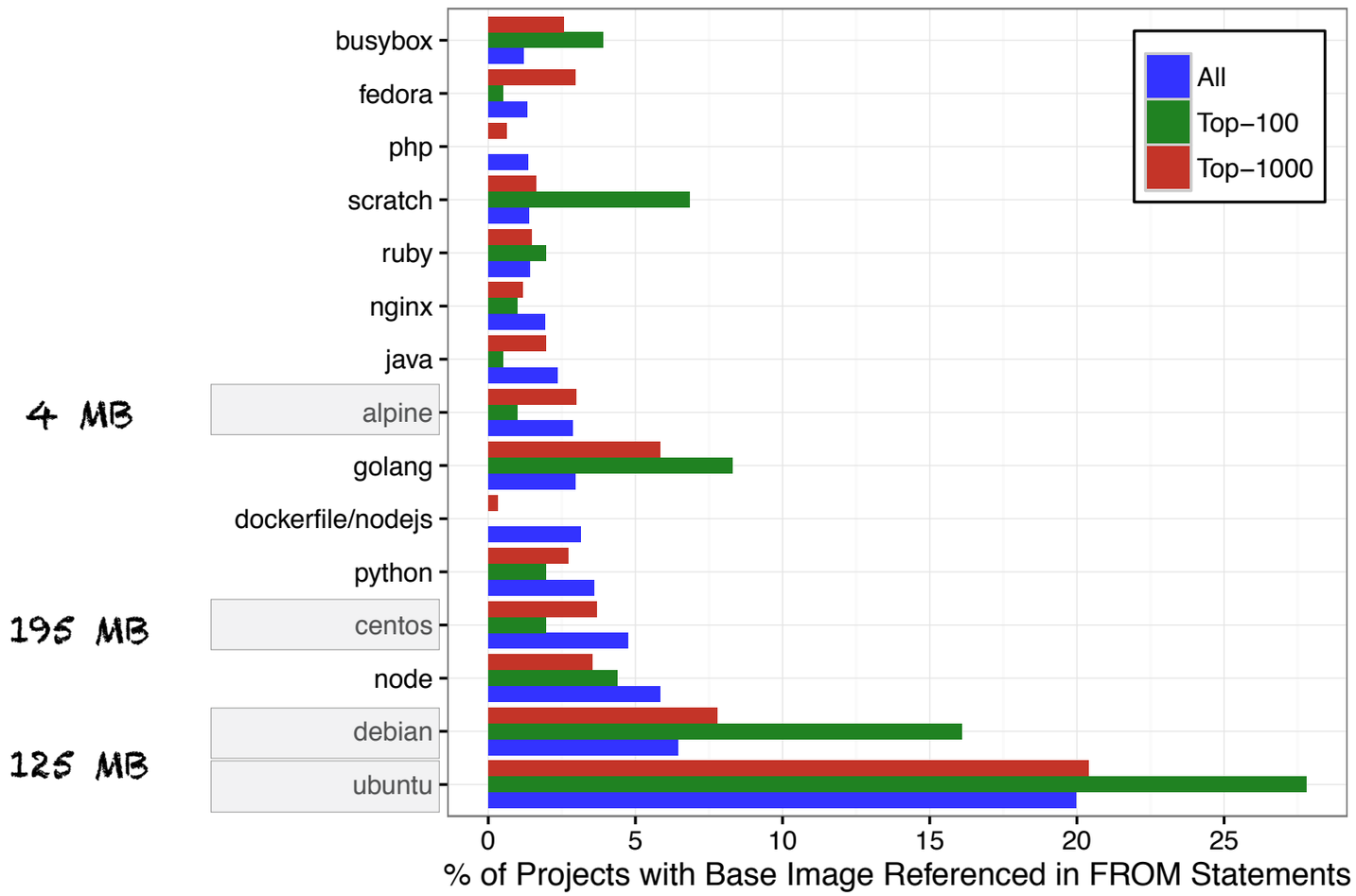
- service executables and dependencies are packaged into a container image
- multiple containers can be spun up from a container image

Containerization insights

Research finding

[Cito et al., MSR17]

Base Images & Sizes



- Reduce Image Size
- Base Image Recommendation

Containerization insights

Research finding

[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%

Containerization insights

Research finding

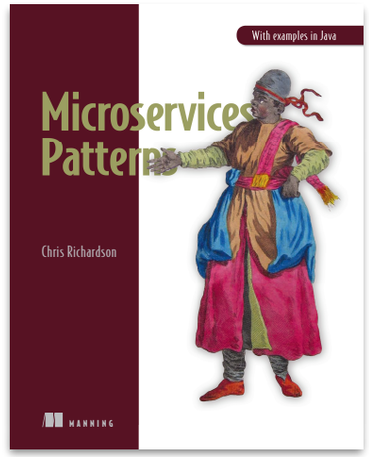
[Cito et al., MSR17]

Distribution of RUN Instructions

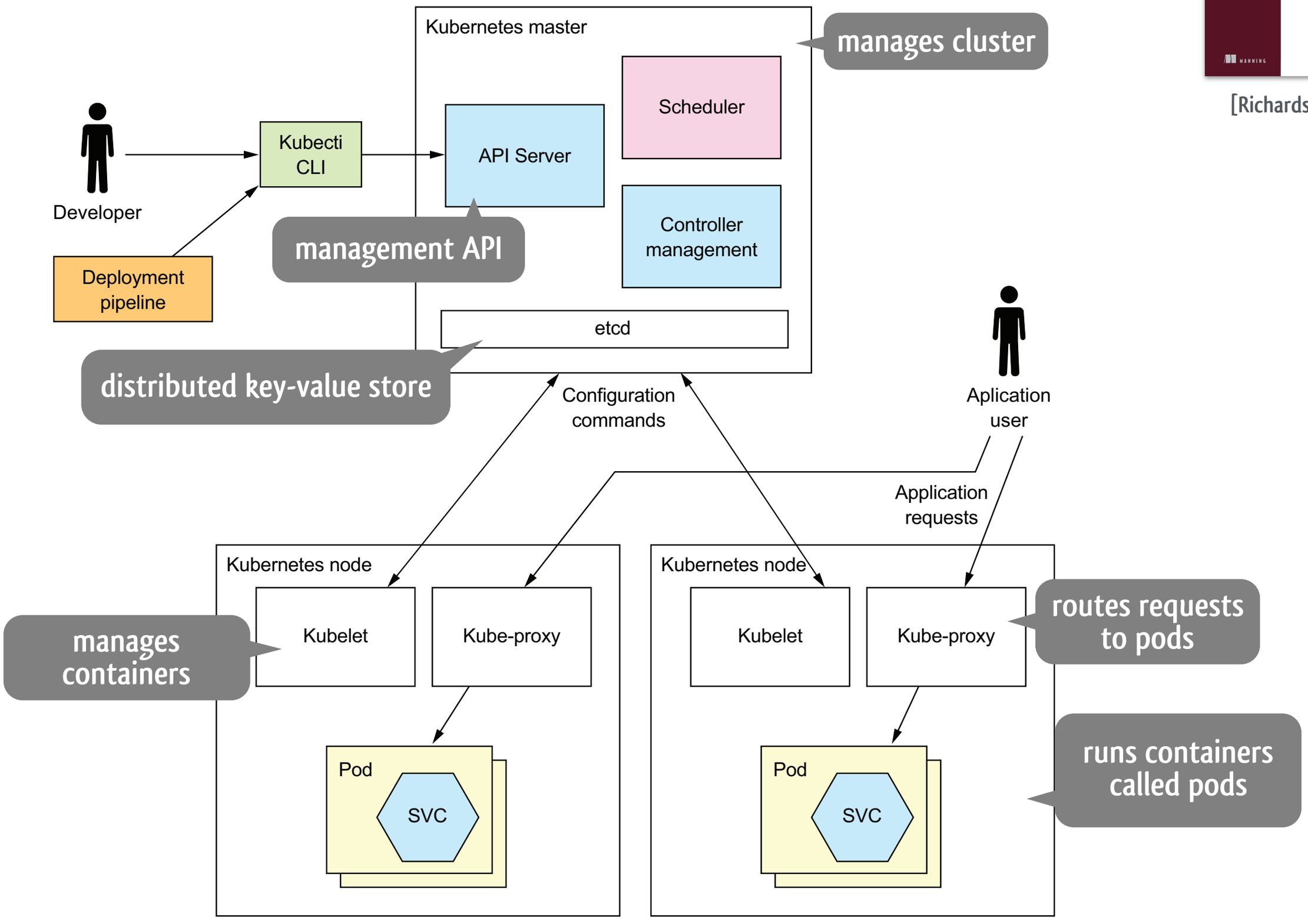
Category	Examples	All	Top-1000	Top-100
Dependencies	apt-get, yum, npm	45.2%	44.7%	45.2%
File System	mkdir, cd, cp, rm	30.4%	29.3%	29.4%
Permissions	chmod, chown	7.3%	5.2%	2.3%
Build / Execute	make, install	5.3%	8.3%	13.5%
Environment	set, export, source	0.6%	1.0%	0.2%
Other		11.3%	11.5%	9.4%

Abstraction for Dependencies

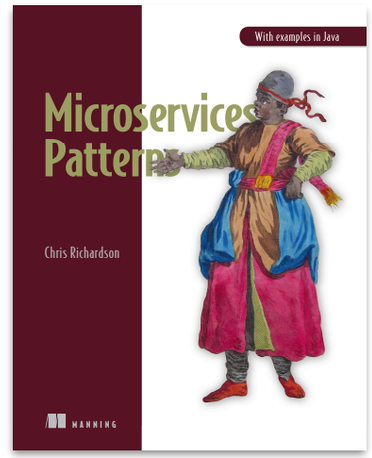
FTGO µ-services: resource provisioning



[Richardson 2019]



FTGO µ-services: infrastructure as code



[Richardson 2019]

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: ftgo-restaurant-service
  labels:
    application: ftgo
    svc: ftgo-restaurant-service
spec:
  replicas: 1
  strategy:
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      labels:
        svc: ftgo-restaurant-service
        application: ftgo
    spec:
      containers:
      - name: ftgo-restaurant-service
        image: msapatterns/ftgo-restaurant-service:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 8080
          name: httpport
        env:
          - name: JAVA_OPTS
            value: "-Dsun.net.inetaddr.ttl=30"
          - name: SPRING_DATASOURCE_URL
            value: jdbc:mysql://ftgo-mysql/eventuate
          - name: SPRING_DATASOURCE_USERNAME
            valueFrom:
              secretKeyRef:
                name: ftgo-db-secret
                key: username
          - name: SPRING_DATASOURCE_PASSWORD
            valueFrom:
              secretKeyRef:
                name: ftgo-db-secret
                key: password
          - name: SPRING_DATASOURCE_DRIVER_CLASS_NAME
            value: com.mysql.jdbc.Driver
          - name: EVENTUATELOCAL_KAFKA_BOOTSTRAP_SERVERS
            value: ftgo-kafka:9092
          - name: EVENTUATELOCAL_ZOOKEEPER_CONNECTION_STRING
            value: ftgo-zookeeper:2181
        livenessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 20
        readinessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 20
```

number of containers to maintain

container image

port

environment variables

determines whether container is ready to accept traffic

determines whether container should be terminated and restarted

Kubernetes

FTGO μ-services: infrastructure as code

Research finding

The Seven Sins: Security Smells in I… Code Scripts

Akond Rahman, Chris Parnin, and Laurie Williams
North Carolina State University, Raleigh, North Carolina
Email: aarahman@ncsu.edu, cjparnin@ncsu.edu, williams@csc.ncsu.edu

Abstract*—Practitioners use infrastructure as code (IaC) scripts to provision servers and development environments. While developing IaC scripts, practitioners may inadvertently introduce security smells. Security smells are recurring coding patterns that are indicative of security weakness and can potentially lead to security breaches. *The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.

We apply qualitative analysis on 1,726 IaC scripts to identify seven security smells. Next, we implement and validate a static analysis tool called Security Linter for Infrastructure as Code scripts (*SLIC*) to identify the occurrence of each smell in 15,232 IaC scripts collected from 293 open source repositories. We identify 21,201 occurrences of security smells that include 1,326 occurrences of hard-coded passwords. We submitted bug reports for 1,000 randomly-selected security smell occurrences. We obtain 104 responses to these bug reports, of which 67 occurrences were accepted by the development teams to be fixed. We observe security smells can have a long lifetime, e.g., a hard-coded secret can persist for as long as 98 months, with a median lifetime of 20 months.

***Index Terms*—devops, infrastructure as code, security smell**

I. Introduction

Infrastructure as code (IaC) scripts help practitioners to provision and configure their development environment and servers at scale [1]. IaC scripts are also known as configuration scripts [2] [1] or configuration as code scripts [1] [3]. Commercial IaC tool vendors, such as Chef [1] and Puppet [4], provide programming syntax and libraries so that programmers can specify configuration and dependency information as scripts.

Fortune 500 companies [2], such as Intercontinental Exchange (ICE) [3], use IaC scripts to maintain their development environments. For example, ICE, which runs millions of financial transactions daily [4], maintains 75% of its 20,000 servers using IaC scripts [5]. The use of IaC scripts has helped ICE decrease the time needed to provision development environments from 1∼2 days to 21 minutes [5].

However, IaC scripts can be susceptible to security weakness. Let us consider Figure 1 as an example. In Figure 1, we present a Puppet code snippet extracted from the 'aeolus-configure' open source software (OSS) repository [5]. In this code snippet, we observe a hard-coded password using… 'password' attribute. A hard-coded string 'v23zj59an' is a… signed as password for user 'aeolus'. Hard-coded passwords in software artifacts is considered as a software security weakness ('CWE-798: Use of Hard-coded Credentials') by Common Weakness Enumerator (CWE) [6]. According to CWE [6], "If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question".

IaC scripts similar to Figure 1, which contain hard-coded credentials or other *security smells*, can be susceptible to security breaches. Security smells are recurring coding patterns that are indicative of security weakness. A security smell does not always lead to a security breach, but deserves attention and inspection. Existence and persistence of these smells in IaC scripts leave the possibility of another programmer using these smelly scripts, potentially propagating use of insecure coding practices. We hypothesize through systematic empirical analysis, we can identify security smells and the prevalence of the identified security smells.

The goal of this paper is to help practitioners avoid insecure coding practices while developing infrastructure as code (IaC) scripts through an empirical study of security smells in IaC scripts.

We answer the following research questions:

- **RQ1**: What security smells occur in infrastructure as code scripts? (Section III)
- **RQ2**: How frequently do security smells occur in infrastructure as code scripts? (Section VI)
- **RQ3**: What is the lifetime of the identified security smell occurrences for infrastructure as code scripts? (Section VI)
- **RQ4**: Do practitioners agree with security smell occurrences? (Section VI)

We answer our research questions by analyzing IaC scripts collected from OSS repositories. We apply qualitative analysis [7] on 1,726 scripts to determine security smells. Next, we construct a static analysis tool called Security Linter for Infrastructure as Code scripts (**SLIC**) to automatically identify the occurrence of these security smells in 15,232 IaC scripts collected by mining 293 OSS repositories from four sources: Mozilla [6], Openstack [7], Wikimedia Commons [8], and GitHub [9].

[1]https://www.chef.io/chef/
[2]http://fortune.com/fortune500/list/
[3]https://www.theice.com/index
[4]https://www.theice.com/publicdocs/ICE_at_a_glance.pdf
[5]https://github.com/aeolusproject/aeolus-configure
[6]https://hg.mozilla.org/
[7]https://git.openstack.org/cgit
[8]https://gerrit.wikimedia.org/r/
[9]https://github.com/

[Rahman et al. ICSE 2019]

TABLE VIII: Smell Occurrences, Smell Density, and Proportion of Scripts for the Four Datasets

	Occurrences				Smell Density (per KLOC)				Proportion of Scripts (Script%)			
Smell Name	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**	**GH**	**MOZ**	**OST**	**WIK**
Admin by default	52	4	35	6	0.1	0.06	0.1	0.04	0.6	0.2	1.1	0.2
Empty password	136	18	21	36	0.3	0.2	0.1	0.2	1.4	0.4	0.5	0.3
Hard-coded secret	10,892	792	3,552	1,716	25.6	11.9	16.5	12.7	21.9	9.9	24.8	17.0
Invalid IP address binding	188	20	114	41	0.4	0.3	0.5	0.3	1.7	0.7	2.9	1.4
Suspicious comment	758	202	305	343	1.7	3.0	1.4	2.5	5.9	8.5	7.2	9.1
Use of HTTP without TLS	1,018	57	460	164	2.4	0.8	2.1	1.2	6.3	1.6	8.5	3.7
Use of weak crypto algo.	177	48	20	26	0.4	0.7	0.1	0.2	0.9	1.1	0.5	0.4
Combined	13,221	1,141	4,507	2,332	31.1	17.2	21.0	17.2	29.3	17.9	32.9	26.7

Cloud-native applications

operate at global scale,
yet remain responsive,
are resilient against failures,
and are elastic under load variations

1990's

SPAGHETTI-ORIENTED
ARCHITECTURE
(aka Copy & Paste)

2000's

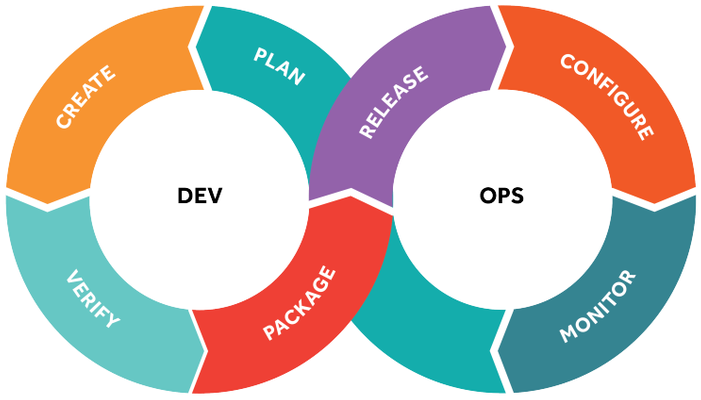
LASAGNA-ORIENTED
ARCHITECTURE
(aka Layered Monolith)

2010's

RAVIOLI-ORIENTED
ARCHITECTURE
(aka Microservices)

architecture

\+



process

\+

infrastructure

µ-services architecture
introduction and motivation

Scaling up
using concurrent actors

Scaling out
using distributed actors

Scaling up through concurrent programming

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

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;

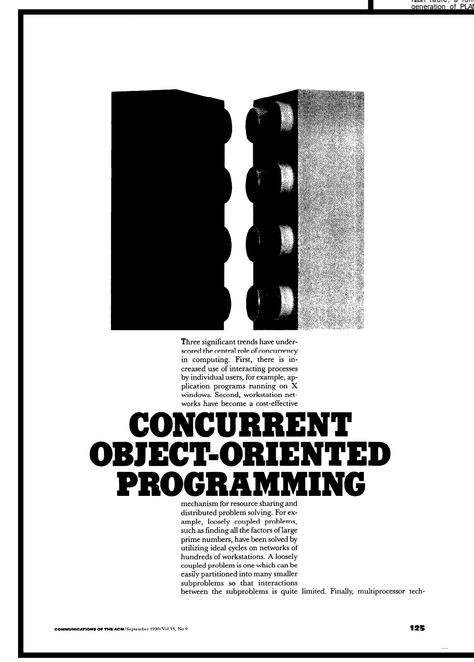
`send to:` sending a message to an actor; and

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

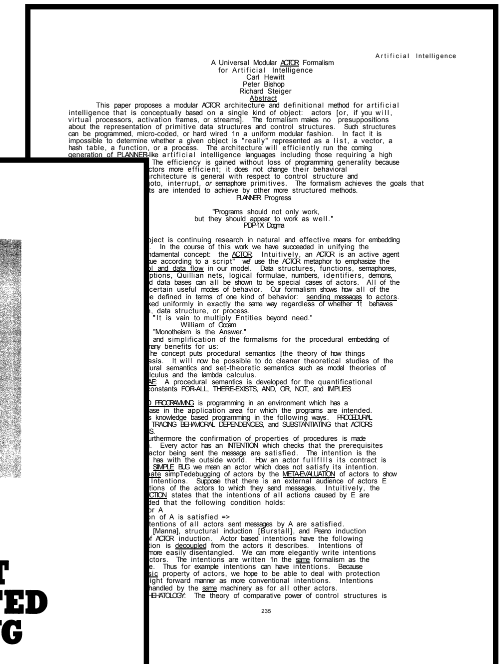
These primitives form a simple but powerful set upon which to build a wide range of higher-level abstractions and concurrent programming paradigms [3]. The actor creation primitive is to concurrent programming what lambda abstraction is to se-quential style sharing to concurrent computation. The send 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



[Agha 1990]

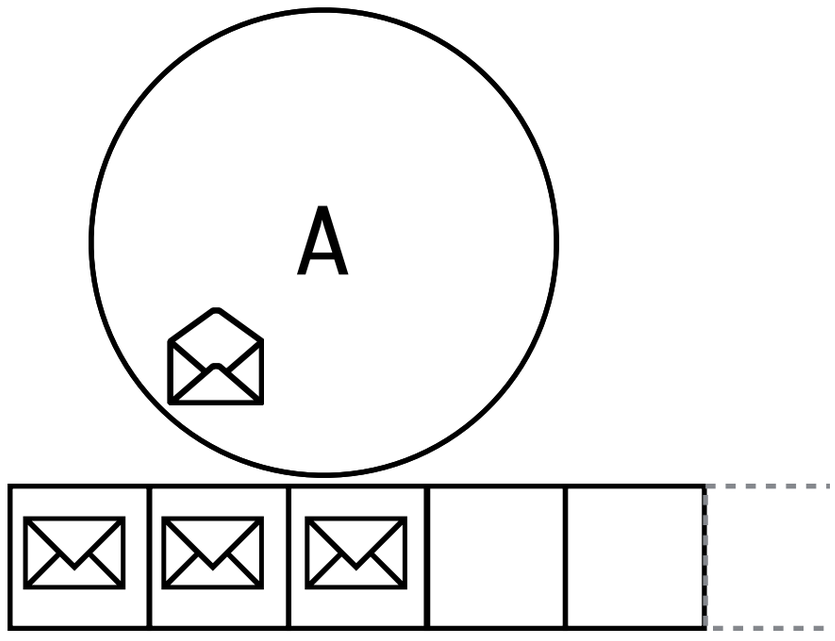


[Hewitt et al., 1973]

Concurrent actor programming

drawal request. In response, as soon as it has computed the new balance in the account, it is free to process the next request—even if other actions implied by the withdrawal request are still being carried out. To put it another way, the concurrent specification of replacement behaviors guarantees noninterference of state changes with potentially numerous threads running through an actor under a multiple-readers, single-writer constraint.

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



Concurrent actor programming

[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):

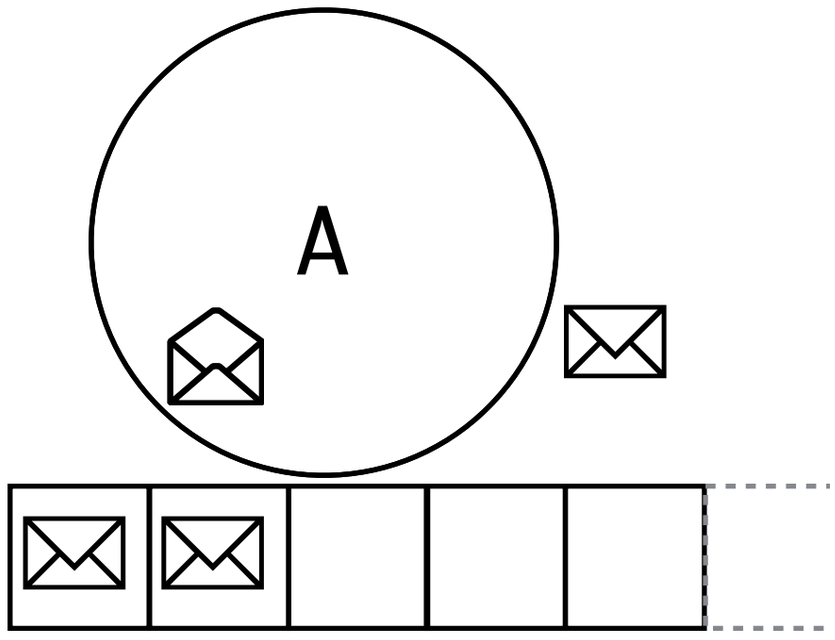
`create:` creating an actor from a behavior description and a set of parameters, possibly including existing actors;

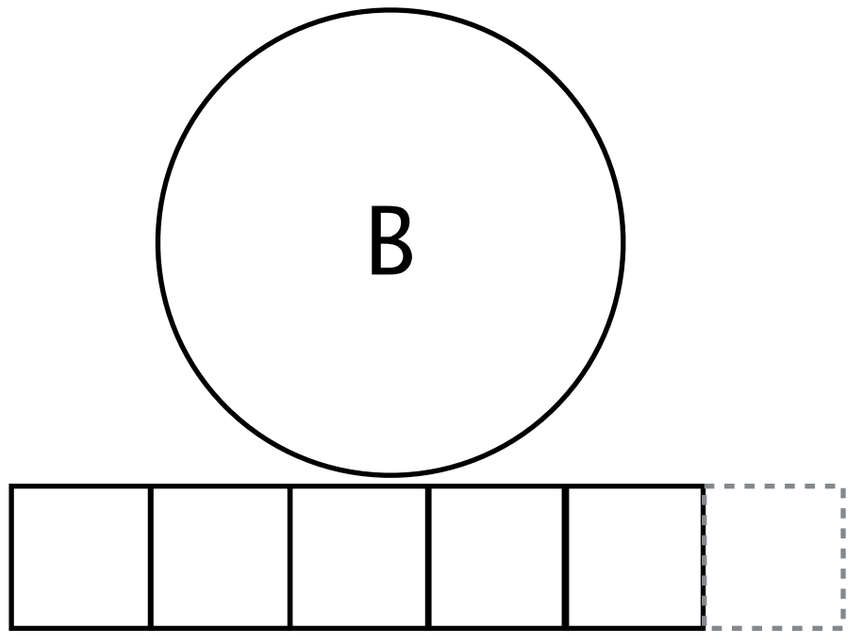
- An actor can only:
  - process messages one-by-one from a mailbox
  - create other actors

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



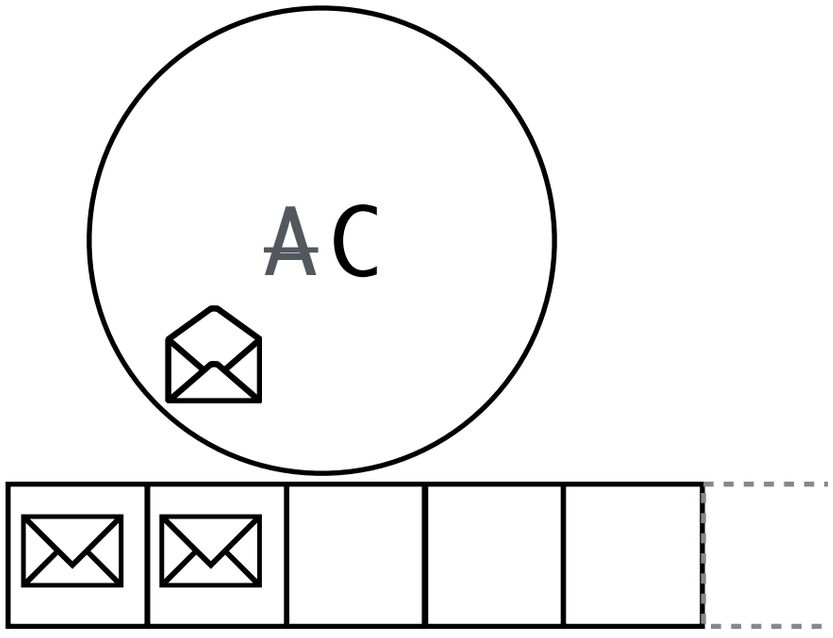


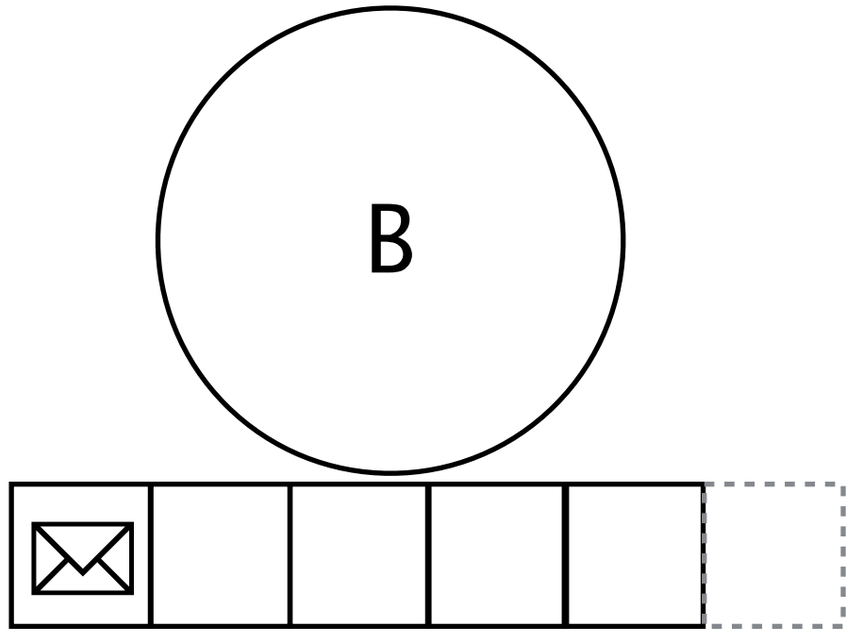
Concurrent actor programming

message.

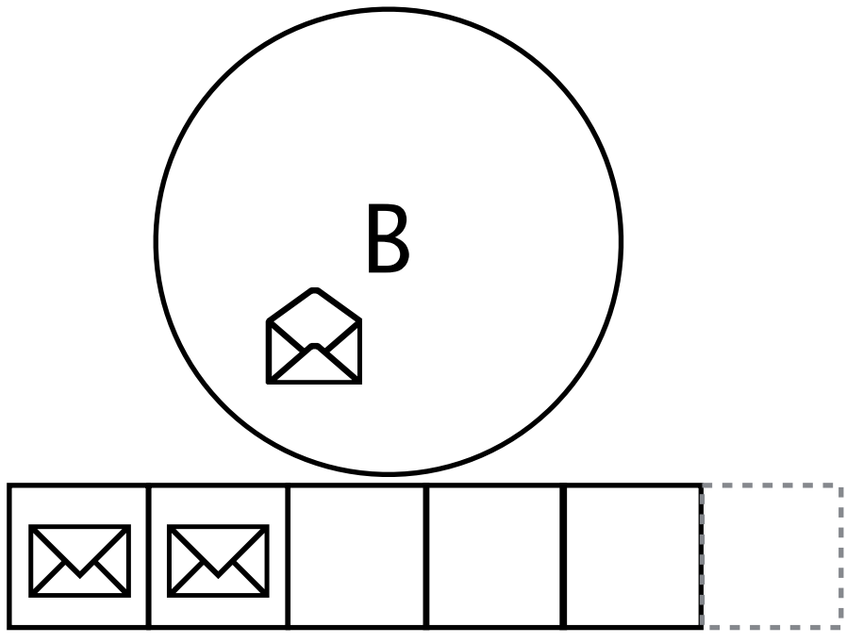
In the actor model, state change is specified using replacement behaviors. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. The replacement behavior for a purely functional actor is identical to the

- 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





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!

akka actors

we will focus on concurrent actors first

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.

iHeart MEDIA · Capital One · credit karma · intel · Hootsuite · NCL NORWEGIAN CRUISE LINE · UPSIDE · Walmart · PayPal · amazon.com · zalando · weightwatchers

akka.net

akka actors

Roger Johansson
@RogerAlsing

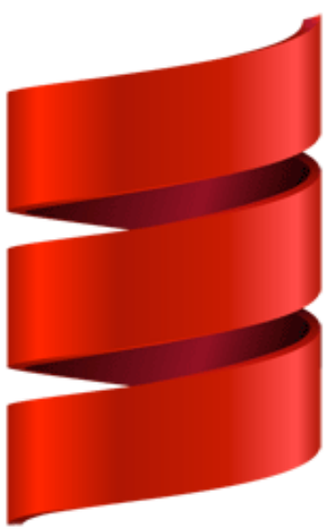
It had to be done, Running @AkkaDotNET from VB.NET, and yes it works fine..
cc @rolandkuhn

```
Module Module1

    Sub Main()
        Dim system = ActorSystem.Create("hellovb")
        Dim actor = system.ActorOf(Of MyActor)()
        actor.Tell("Akka")
        actor.Tell(5)
        Console.ReadLine()
    End Sub

End Module

Public Class MyActor
    Inherits UntypedActor

    Protected Overrides Sub OnReceive(ByVal message As Object)
        Select Case True
            Case TypeOf message Is String
                Console.WriteLine("Hello {0}", message)
            Case TypeOf message Is Integer
                Dim res = 10 + message
                Console.WriteLine("result is {0}", res)
        End Select
    End Sub

End Class
```

9:51 PM · Apr 12, 2015 · Twitter Web Client

akka.net

Scala for Java programmers



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

Functional basics: expressions

```
var x = 3
> x : Int = 3
var y = 4
> y : Int = 4

if(x > y)
  println(x)
else
  println(y)
4

println(if(x>y) x else y)
4

var max = (x : Int, y : Int) => if(x>y) x else y
> max : (Int, Int) => Int = <function2>

max(3,4)
> res0 : Int = 4

def max(x : Int, y : Int ) =
  if(x>y) x else y
```

types are inferred

every expression has a value

functions can be assigned to variables

functions are values

Functional basics: pattern matching

```
def process(input : Any) : String =
  input match {
    case "apple" => "Fruit"

    case input : Int => {
      println("match!")
      input.toString()
    }

    case (a : Int, b : Int) => "Int tuple"
    case (a : Double, b : Double) => "Double tuple"

    case _ => "Unknown input"
  }

List("apple", 2, (0.5, 1.1), (3,4), List()).map(process)
match!
> res0: List[String] =
          List(“Fruit”, “2", “Double tuple, “Int tuple, “Unknown input”)

def factorial(n: Int): Int = n match {
  case 0 => 1
  case _ => n * factorial(n-1)
}
```

matching on value

matching on type

deep match

wildcard

OO basics: inheritance

```
class Animal {
   def makeSound() = "Sound!"
}

class Dog extends Animal {
   override def makeSound() = "Bark!"
}

class Duck extends Animal {
   override def makeSound() = "Quack!"
}

class DecoyDuck extends Duck {
   override def makeSound() = ""
}

var animals = List(new Dog, new Duck, new DecoyDuck)
> animals : List[Animal] = List(Dog@21bcffb, Duck@380fb434, DecoyDuck@668bc3d5)

animals.map(a => a.makeSound())
> res0: List[String] = List(Bark!, Quack!, "")
```

inheritance

overrides inherited definition

inferred parametric type

polymorphic method invocations are supported as expected

OO basics: traits

will differ depending on which class the trait is "mixed in"

```
trait Loud extends Animal {
   override def makeSound() = super.makeSound().toUpperCase()
}

trait Tired extends Animal {
   var count = 0;
   override def makeSound = {
      count = count + 1
      if(count > 1) "" else super.makeSound()
   }
}
```

reusable state and behavior

```
class LoudDog extends Dog with Loud
var fifi = new LoudDog
> fifi : LoudDog = LoudDog$1@73f792cf
fifi.makeSound()
> res1: String = BARK!
class TiredLoudDuck extends Duck with Loud with Tired
var donald = new TiredLoudDuck
> donald : TiredLoudDuck = TiredLoudDuck$1@2ed94a8b
donald.makeSound()
> res2: String = QUACK!
donald.makeSound()
> res3: String = ""
donald.count
> res4: Int = 2
```

classes can be composed with reusable pieces of state and behavior (no equivalent in Java)

OO basics: case classes

```
trait Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr) extends Expr
case class BinOp(operator: String, left: Expr, right: Expr) extends Expr
```

```
val v = Var(“x")
> v: Var = Var(x)
val addop = BinOp("+", Number(1), v)
> addop: BinOp = BinOp(+,Number(1.0),Var(x))
val l = addop.left
> l: Expr = Number(1.0)
println(addop)
BinOp(+,Number(1.0),Var(x))

addop.right == Var("x")
> res1: Boolean = true
val subop = addop.copy(operator = "-")
> subop: BinOp = BinOp(-,Number(1.0),Var(x))

def simplifyTop(expr: Expr): Expr = expr match {
    case UnOp("-", UnOp("-", e))  => e
    case BinOp("+", e, Number(0)) => e
    case BinOp("*", e, Number(1)) => e
    case _ => expr
}
simplifyTop(UnOp("-", UnOp("-", v)))
> res2: Expr = Var(x)
```

compiler has generated factory method in each companion object

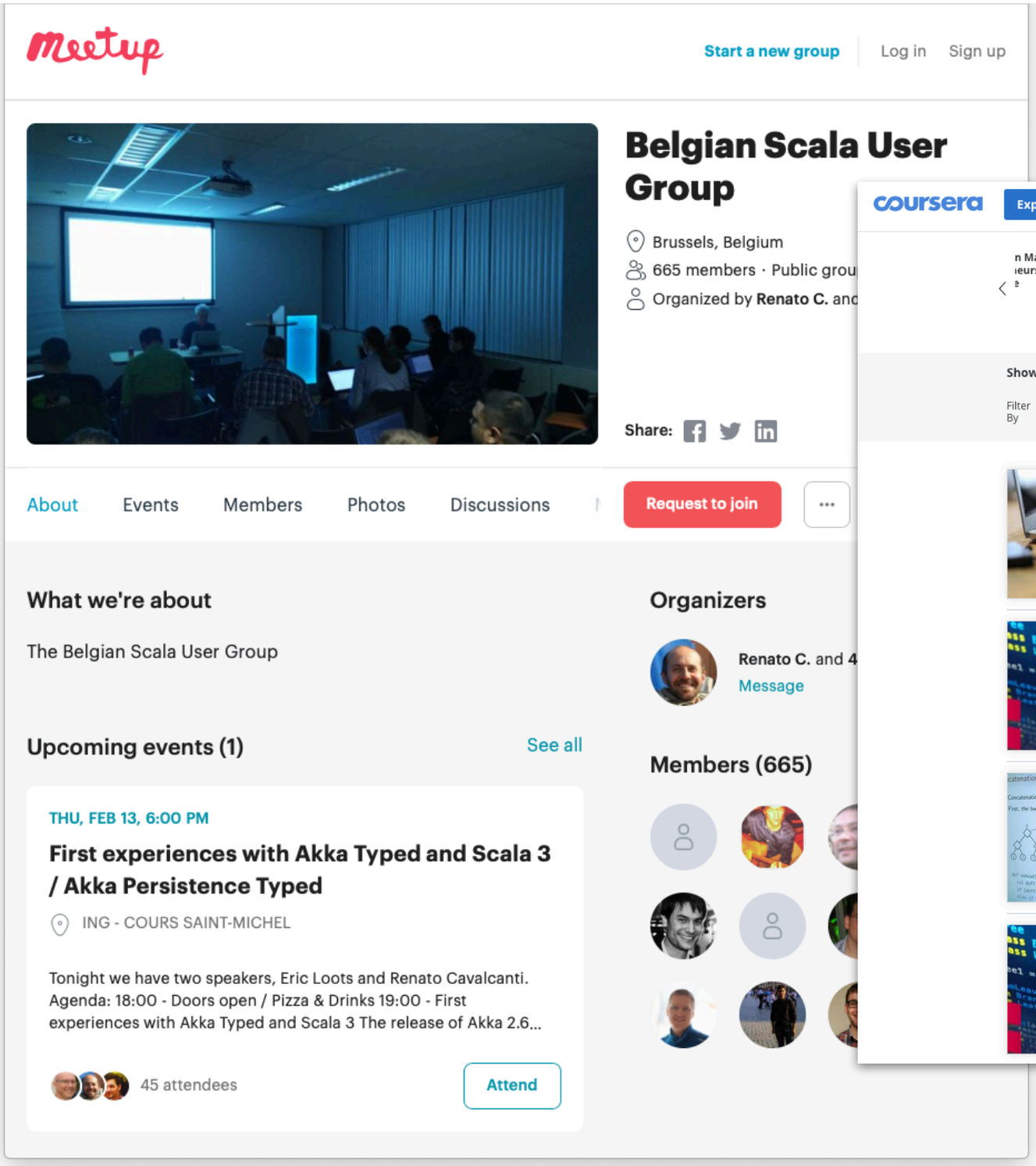
compiler has generated toString methods

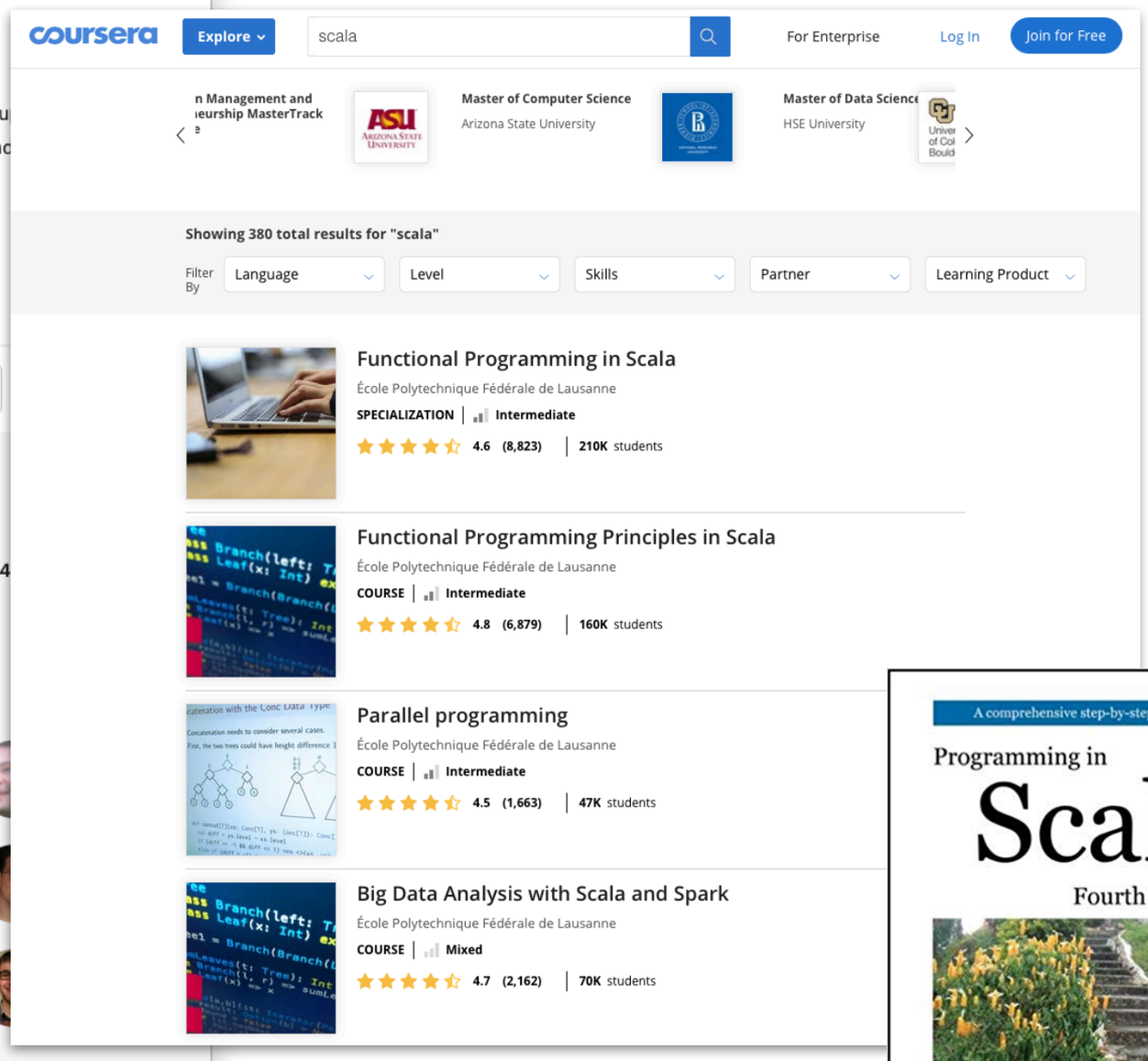
compiler has generated equals and hashCode methods

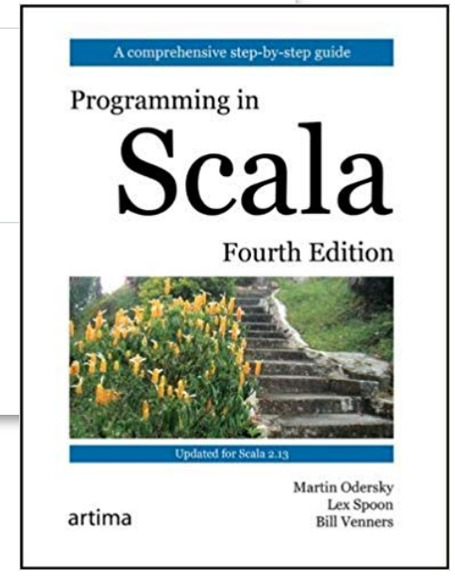
recursive comparison of structure, rather than addresses

compiler has generated copy methods for “changing” constructor arguments

For those who would like to know more







akka actors

we will focus on concurrent actors first

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.

iHeart MEDIA · Capital One · credit karma · intel · Hootsuite · NCL NORWEGIAN CRUISE LINE · UPSIDE · Walmart · PayPal · amazon.com · zalando · weightwatchers

akka.net

akka actors: Ping Pong example

```
object PingPong extends App {

  case class StartPingPong(partner : ActorRef)
  case object Ball

  class Paddle extends Actor {
    var counter : Integer = 0

    def receive = {
      case StartPingPong(partner : ActorRef) =>
        counter = 0
        partner ! Ball

      case Ball =>
        counter += 1
        println(s"Count of ${self.path.name}: ${counter}")
        Thread.sleep(1000)
        sender ! Ball
    }
  }

  val system = ActorSystem("PingPong")
  val ping = system.actorOf(Props[Paddle], "ping")
  val pong = system.actorOf(Props[Paddle], "pong")
  ping ! StartPingPong(pong)

}
```

messages exchanged

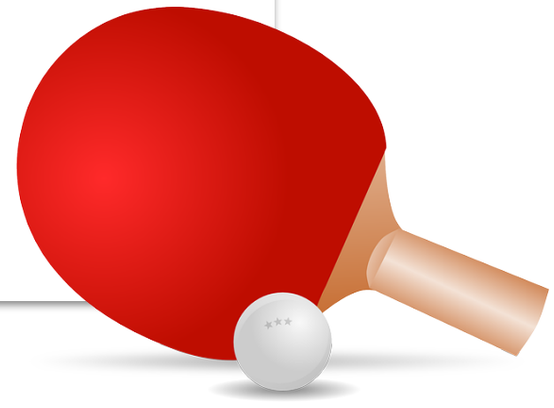
address at which an actor lives

method returning message processor

return ball to sender

start game

```
Pong: 1
Ping: 1
Pong: 2
Ping: 2
Pong: 3
Ping: 3
Pong: 4
Ping: 4
Pong: 5
Ping: 5
Pong: 6
Ping: 6
…
```



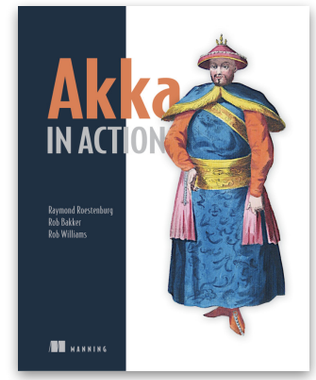
akka actors: strong encapsulation

- No direct access possible to the actor state
  - state can only be accessed through messages sent to known addresses (represented by an ActorRef)

- Three ways to obtain an address:
  - every actor knows its own address (self), useful for sending messages to other actors and informing them where to reply to
  - actor creation returns an address (an ActorRef), it is not possible to call methods directly on the newly created actor
  - addresses can be exchanged through messages (cf. automatically captured sender in Akka)

- Actors are completely independent agents of computation, more isolated from each other than regular objects
  - all actors run fully concurrently to each other
  - message-passing primitive is asynchronous

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

Demo

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

Demo

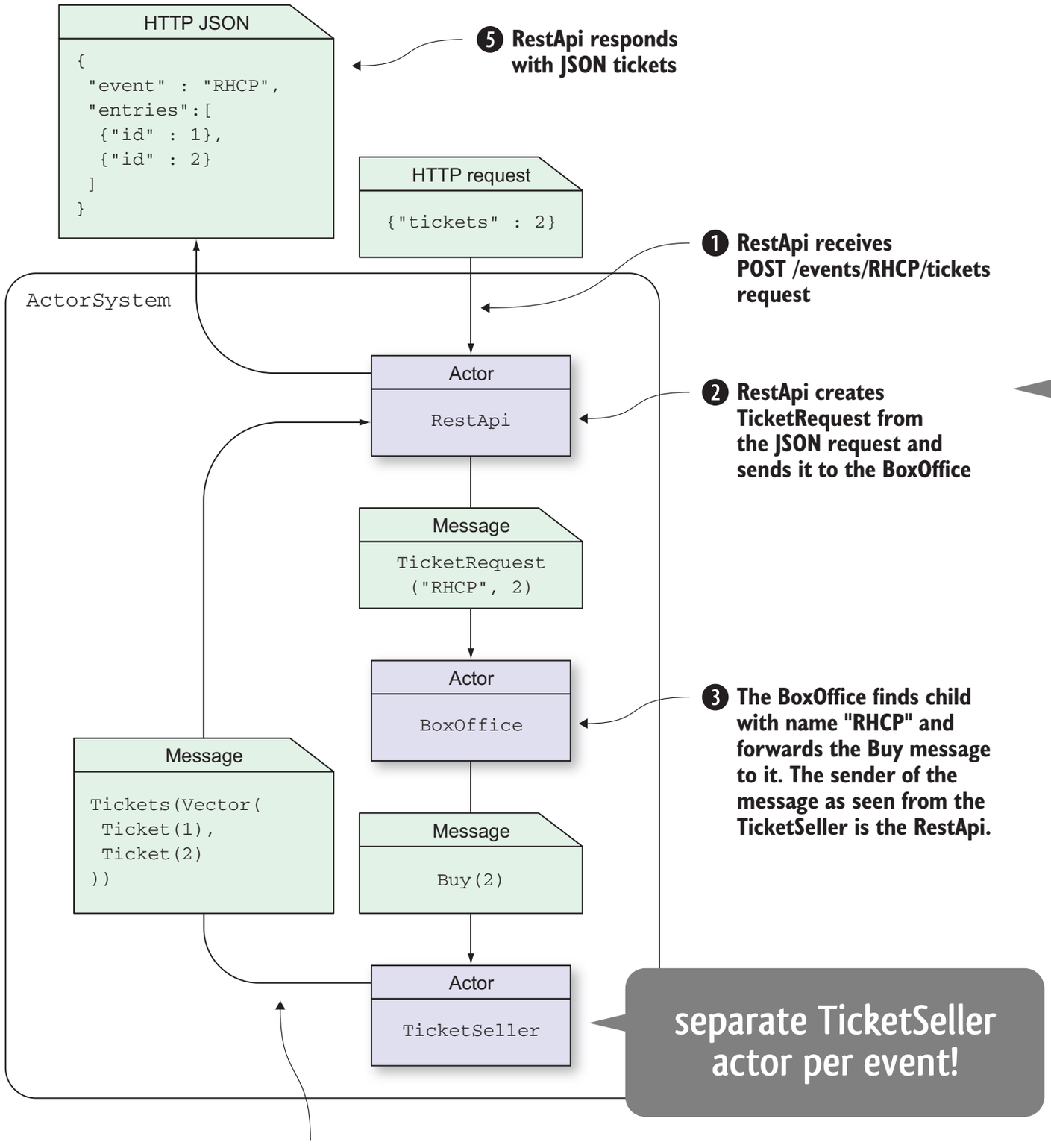
purchase two tickets for Red Hot Chilli Peppers event

```
:~ 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

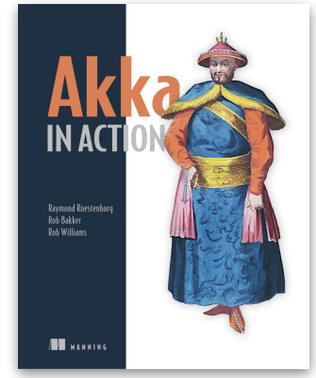
```
:~ cderoove$ http GET localhost:5000/events/
HTTP/1.1 200 OK
Content-Length: 73
Content-Type: application/json
Date: Tue, 06 Feb 2018 12:23:14 GMT
Server: GoTicks.com REST API

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

GoTicks.com: REST API



processing tickets and answering REST calls happens concurrently

separate TicketSeller actor per event!



[Roestenburg et al. 2016]

Actors facilitate fine-grained upscaling within a container

GoTicks example: scaling upwards

```
class TicketSeller(event: String) extends Actor {

  var tickets : List[Ticket] = List()

  def receive = {
    case Add(newTickets) =>
      tickets = tickets ++ newTickets

    case Buy(nrOfTickets) if nrOfTickets <= tickets.size =>
      val entries = tickets.take(nrOfTickets)
      tickets = tickets.drop(nrOfTickets)
      sender() ! Tickets(event, entries)

    case Buy(_) =>
      sender() ! Tickets(event, List())

    case GetEvent =>
      sender() ! BoxOffice.Event(event, tickets.size)

    case Cancel =>
      sender() ! BoxOffice.Event(event, tickets.size)
      self ! PoisonPill
  }
}
```

list of numbered tickets

populated by the BoxOffice actor

answer the requested number of tickets

or an empty list of tickets

answer the remaining tickets for the event

terminates the TicketSeller actor when the event is canceled

GoTicks example: scaling upwards

```
class BoxOffice extends Actor with ActorLogging {
  import BoxOffice._

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

  def receive : Receive = {
    case CreateEvent(name, tickets) =>
      context.child(name) match {
        case None =>
          val newSellerRef = createTicketSeller(name)
          val newTickets = (1 to tickets).map(ticketId => TicketSeller.Ticket(ticketId))
          newSellerRef ! TicketSeller.Add(newTickets.toList)
          sender ! EventCreated(Event(name, tickets))
        case Some(_) => sender ! EventExists
      }

    case GetTickets(event, tickets) =>
      context.child(event) match {
        case None =>  sender ! TicketSeller.Tickets(event, Nil)
        case Some(seller : ActorRef) => seller.forward(TicketSeller.Buy(tickets))
      }
```

creates a TicketSeller for the given event as a child actor

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

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

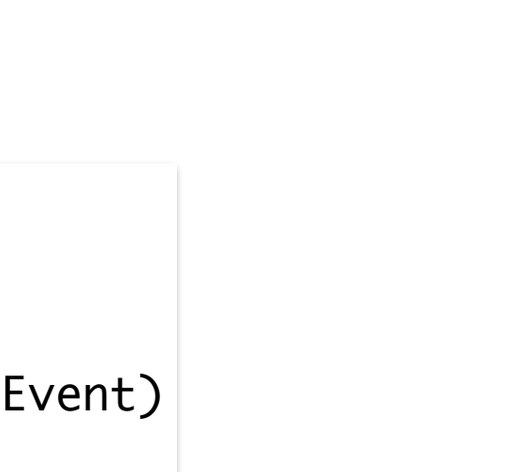
communicates success back to RestAPI actor

forwards, rather than sends, a Buy message to the appropriate child actor
this ensures responses will go to the REST API

GoTicks example: scaling upwards

```
case GetEvent(event) =>
  context.child(event) match {
    case None => sender ! None
    case Some(seller : ActorRef) => seller.forward(TicketSeller.GetEvent)
  }

case CancelEvent(event) =>
  context.child(event) match {
    case None => sender ! None
    case Some(seller: ActorRef) => seller.forward(TicketSeller.Cancel)
  }

case GetEvents =>
  context.children.foreach(seller => seller ! TicketSeller.GetEvent)
  context.setReceiveTimeout(100 microseconds)
  context.become(receiveResponses(sender, Nil))
}
```

how to collect responses?



switch to another message processing function to accumulate responses!

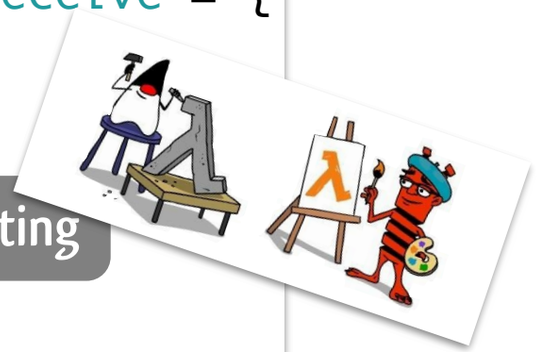
GoTicks example: scaling upwards

destination for accumulation

responses accumulated so far

```
def receiveResponses(replyTo : ActorRef, responses : List[Event]) : Receive = {
    case response : Event =>
      context.become(receiveResponses(replyTo, response :: responses))
    case ReceiveTimeout =>
      replyTo ! Events(responses)
      context.setReceiveTimeout(Duration.Undefined)
      context.become(receive)
  }
}
```

continue accumulating



change back to regular message processing

send accumulation upon response timeout

actually an ad-hoc implementation of Aggregator pattern!
could also use built-in support for futures

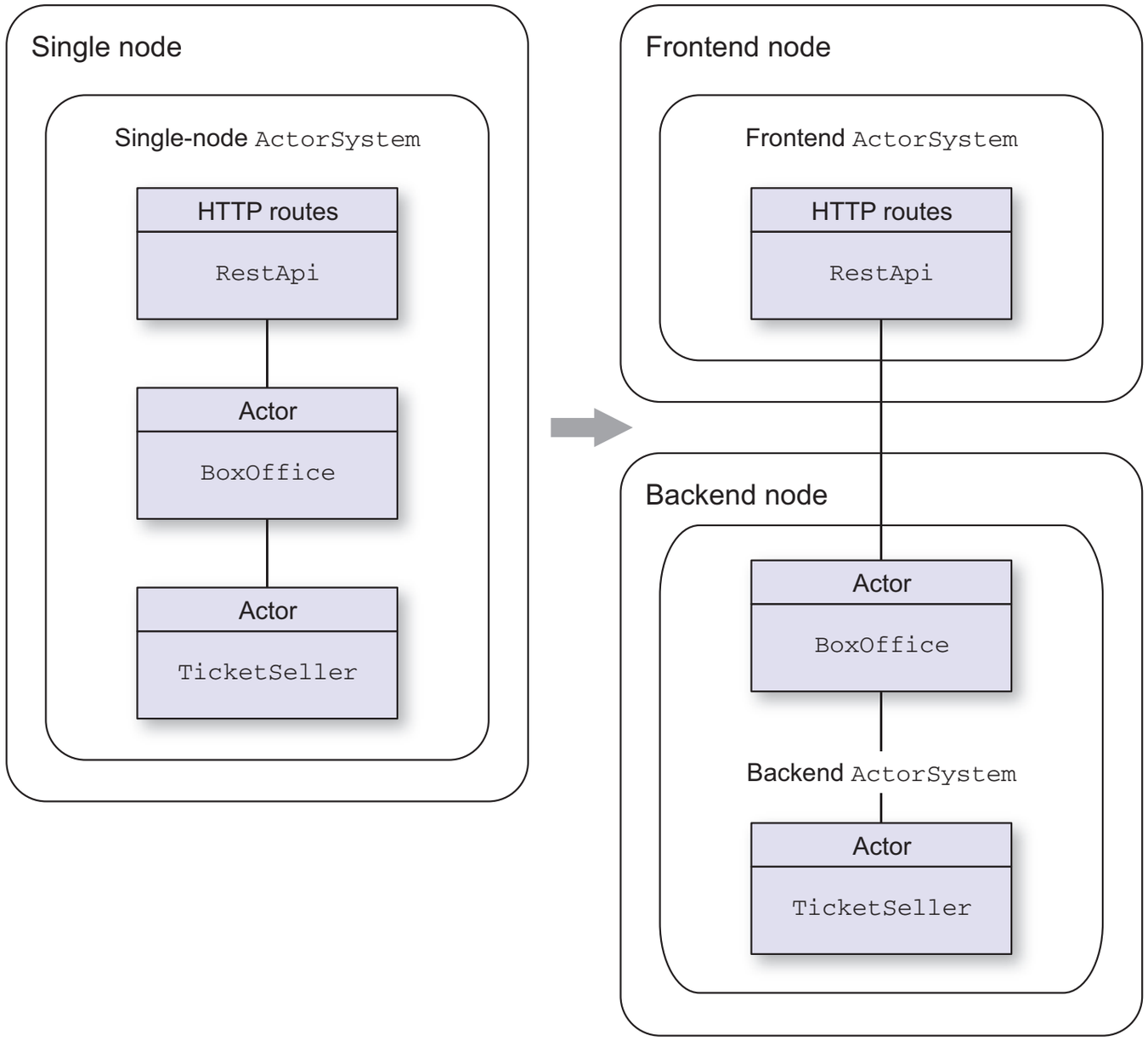
μ-services architecture
introduction and motivation

Scaling up
using concurrent actors

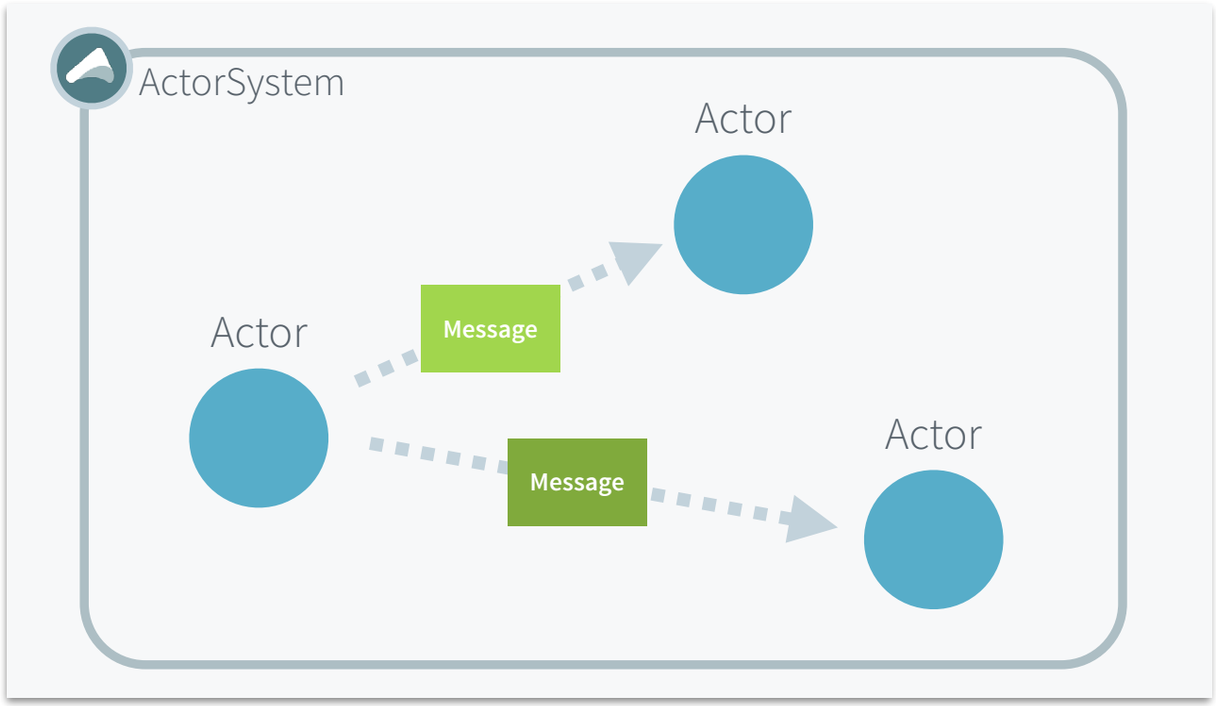
Scaling out
using distributed actors

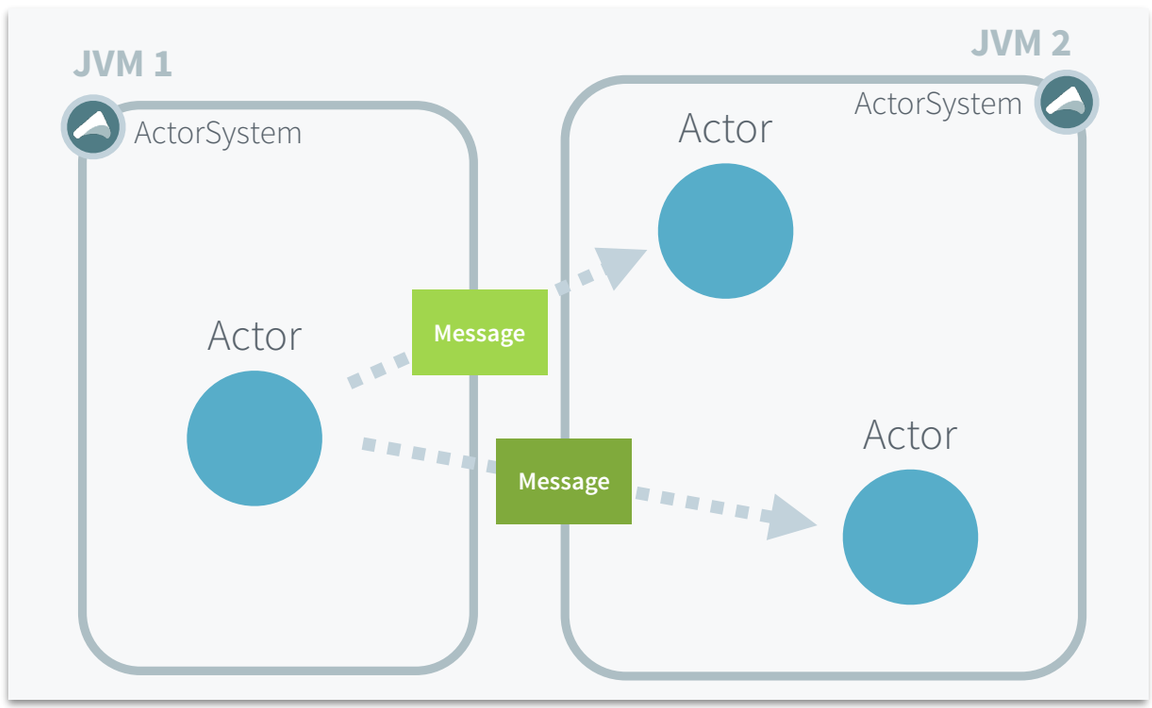
Scaling out through distributed programming

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



Distributed actor programming





actor systems are distributable by design

- actors are strongly-encapsulated: no shared data
- communication through addresses (ActorRefs) is location-transparent:
  same ! for sending asynchronous message to local and to remote ActorRef

Reality strikes

``A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.''

Leslie Lamport,
2013 Turing Award winner

Resilience against delivery failures

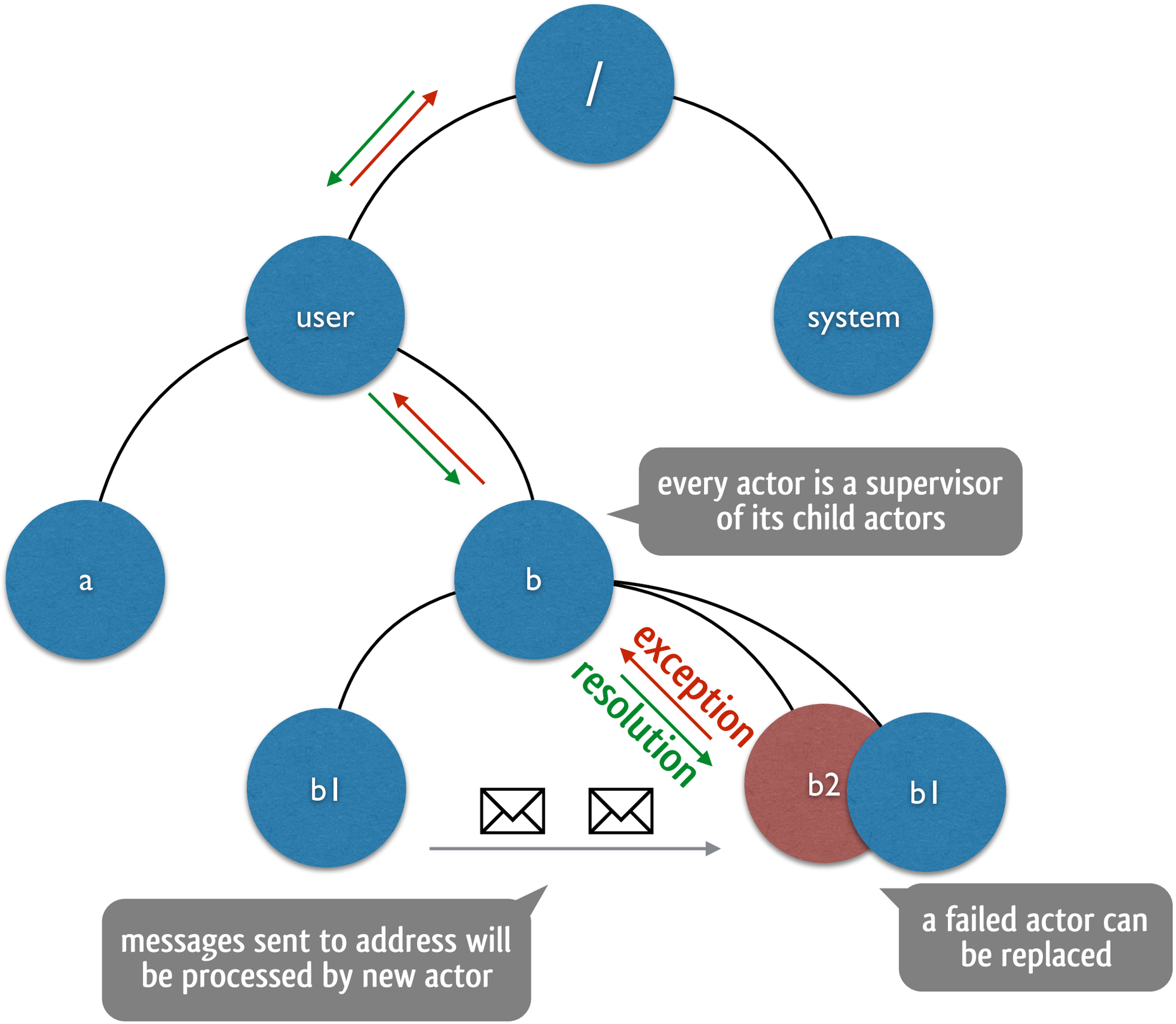
unfortunately, (distributed) communication is inherently unreliable
delivery of a message requires eventual availability of channel and recipient

rendering messages first-class entities enables implementing delivery guarantees:

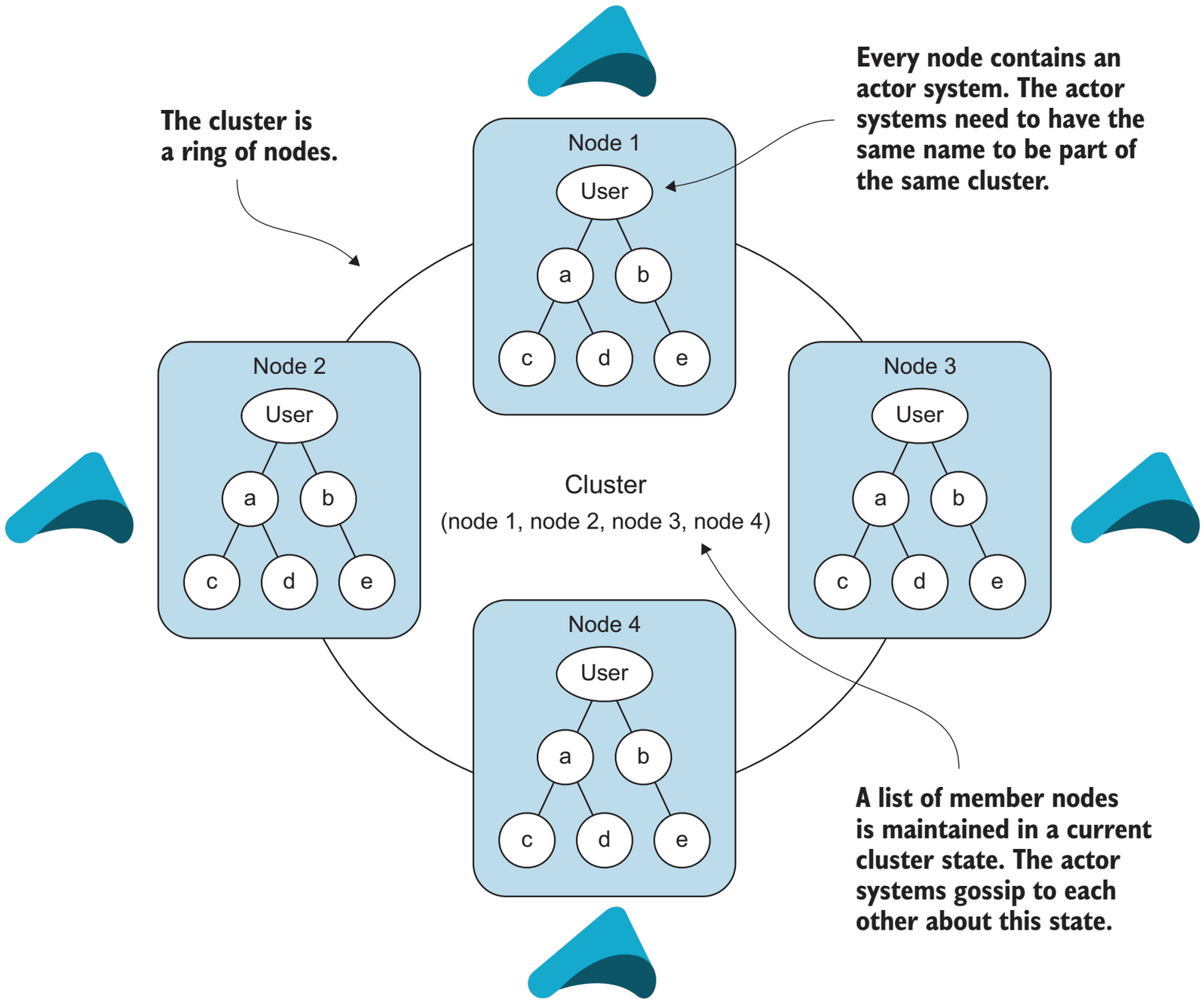
- at-most-once delivery:
  - no state required at sender nor receiver, a message sent once will either arrive or not
  - message will be delivered [0,1] times
- at-least-once:
  - keep state at the sender to ensure that a message will be resent until it has been acknowledged by the recipient
  - message 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

Resilience against unexpected exceptions

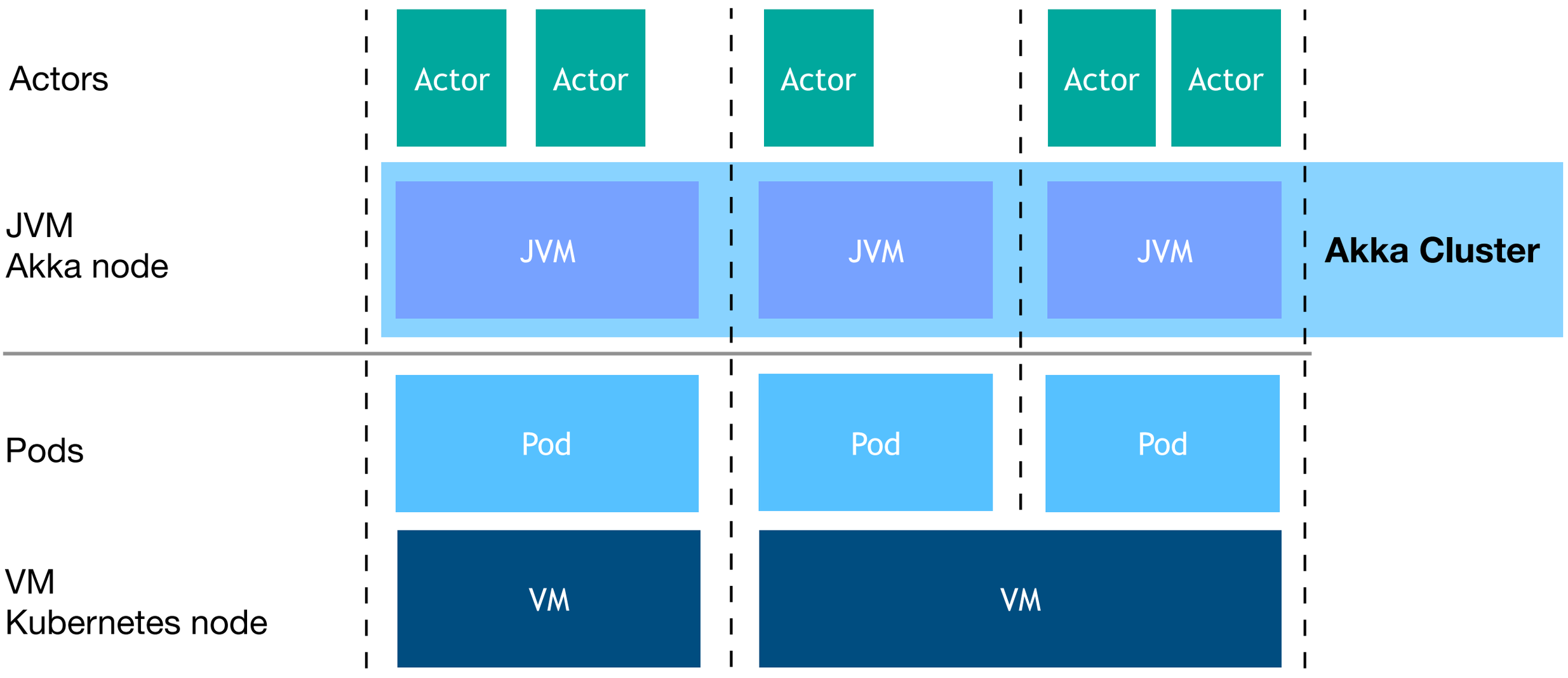


Application-level resilience against node failures



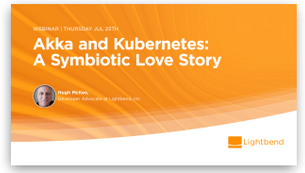
- decentralised peer-to-peer cluster membership: gossip protocol and failure detection
- message routing: load balancing, topic-based publish/subscribe

Infrastructure-level resilience against node failures

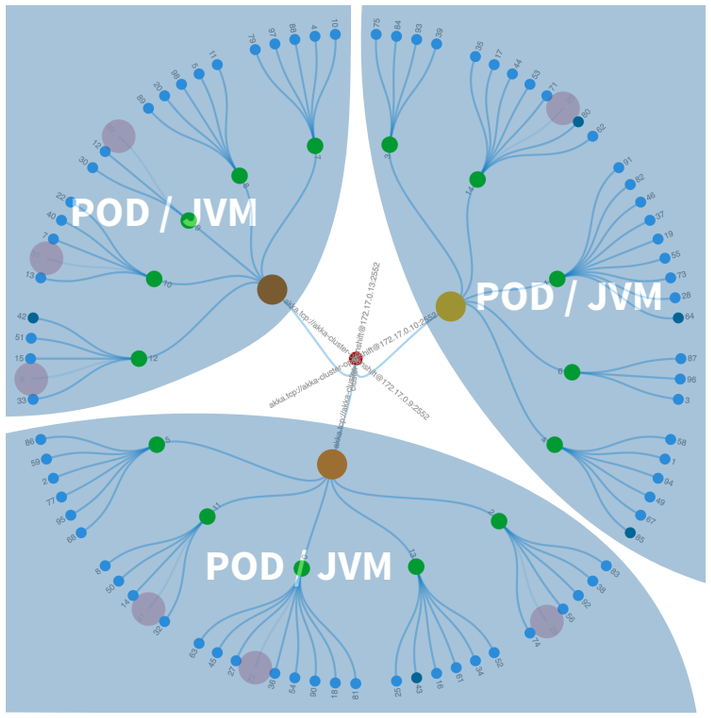


[https://www.lightbend.com/blog/akka-and-kubernetes-reactive-from-code-to-cloud]

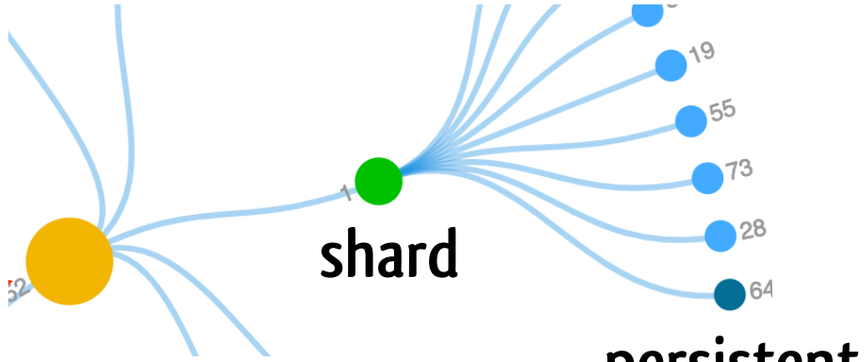
Application-level + infrastructure-level



[Hugh McKee 2019]



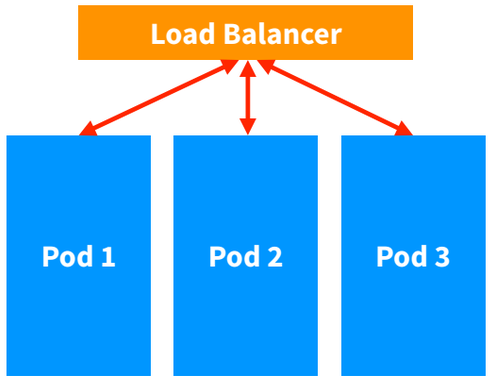
load balancer

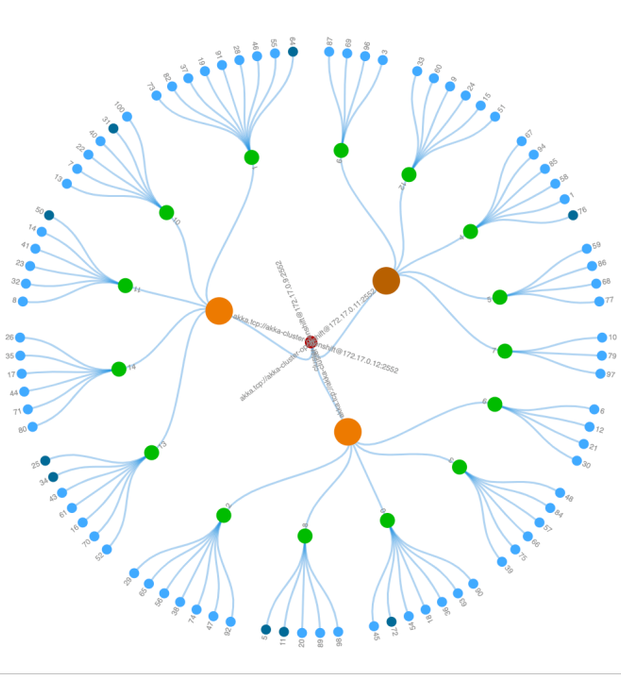


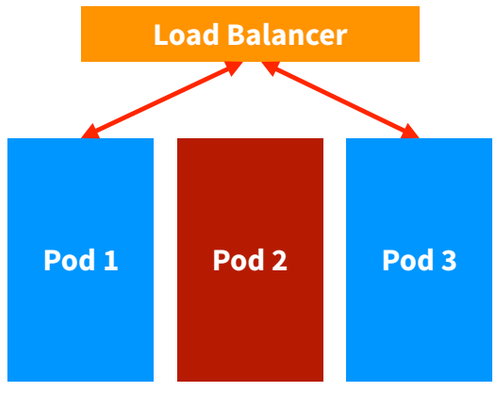
REST API

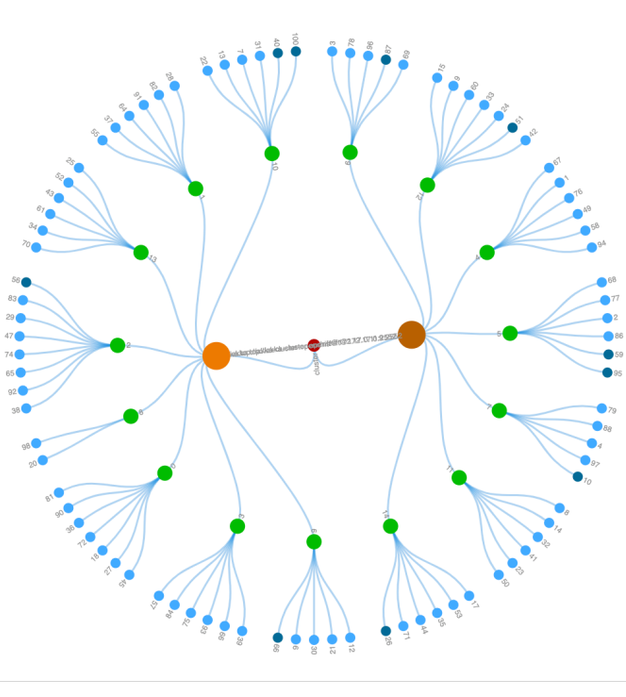
shard

persistent actors

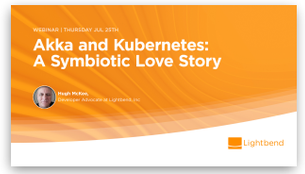




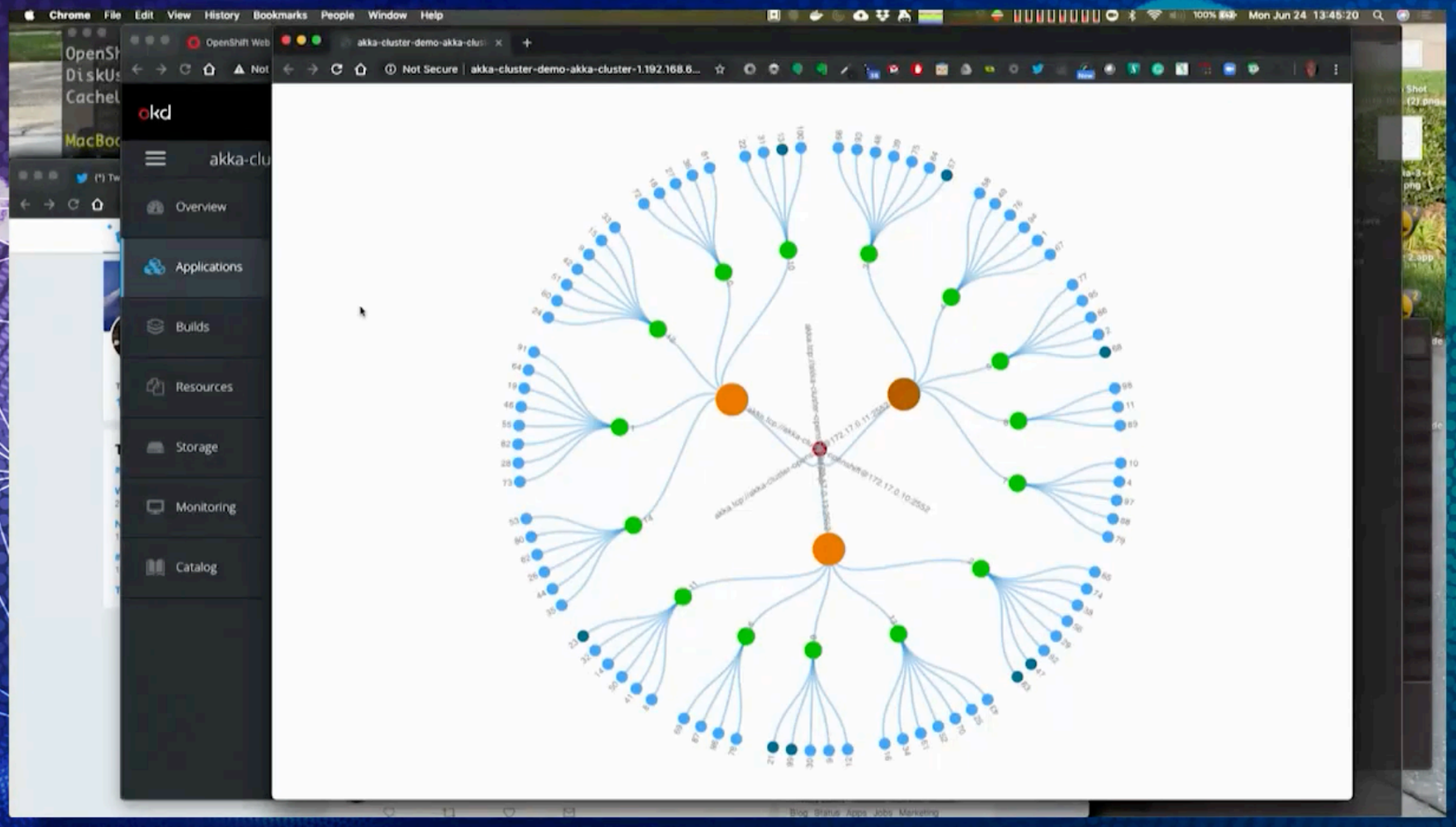




Application-level + infrastructure-level



[Hugh McKee 2019]



DEVOXX™

But resilience remains difficult to get right …

Ongoing research

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

TEAMS What's this?

Free 30 Day Trial

Akka persistence with confirmed delivery gives inconsistent results

Ask Ques…

Asked 5 years, 1 month ago Active 5 years, 1 month ago Viewed 2k times

7

5

I have been playing around with Akka Persistence and have written the following program to test my understanding. The problem is that I get different results each time I run this program. The correct answer is 49995000 but I don't always get that. I have cleaned out the journal directory between each run but it does not make any difference. Can anyone see what's going wrong? The program simply sums all the numbers from 1 to n (where n is 9999 in the code below).

The correct answer is : (n * (n+1)) / 2. For n=9999 that's 49995000.

EDIT: Seems to work more consistently with JDK 8 than with JDK 7. Should I be using JDK 8 only?

```
package io.github.ourkid.akka.aggregator.guaranteed

import akka.actor.Actor
import akka.actor.ActorPath
import akka.actor.ActorSystem
import akka.actor.Props
import akka.actor.actorRef2Scala
import akka.persistence.AtLeastOnceDelivery
import akka.persistence.PersistentActor

case class ExternalRequest(updateAmount : Int)
case class CountCommand(deliveryId : Long, updateAmount : Int)
case class Confirm(deliveryId : Long)

sealed trait Evt
case class CountEvent(updateAmount : Int) extends Evt
case class ConfirmEvent(deliveryId : Long) extends Evt

class TestGuaranteedDeliveryActor(counter : ActorPath) extends PersistentActor wit

  override def persistenceId = "persistent-actor-ref-1"

  override def receiveCommand : Receive = {
    case ExternalRequest(updateAmount) => persist(CountEvent(updateAmount))(update
    case Confirm(deliveryId) => persist(ConfirmEvent(deliveryId)) (updateState)
  }

  override def receiveRecover : Receive = {
    case evt : Evt => updateState(evt)
  }

  def updateState(evt:Evt) = evt match {
    case CountEvent(updateAmount) => deliver(counter, id => CountCommand(id, updat
    case ConfirmEvent(deliveryId) => confirmDelivery(deliveryId)
  }
}
```

SBT file

```
resolvers ++= Seq(
    "Typesafe Repository" at "http://repo.typesafe.com/typesafe/releases/"
)
```

Blog

- Does your web app need a front-end framework?
- Podcast: Your Buddy is Typing

Featured on Meta

- Planned Maintenance scheduled for Wednesday, February 5, 2020 for Data Explorer
- In case you missed it: Shog9 and Robert Cartaino are no longer staff members…
- An account of my meeting with the Stack Overflow management team

[https://stackoverflow.com/questions/27592304/akka-persistence-with-confirmed-delivery-gives-inconsistent-results]

But resilience remains difficult to get right …

Ongoing research

```
4  trait Event
5  case class Plus(amount: Int)
6  case class PlusEvent(amount : Int) extends Event
7  case class CountCommand(id : Long, amount : Int)
8  case class ConfirmEvent(id : Long) extends Event
9  case class Confirm(id : Long)
10
11 class GuaranteedDeliveryActor(ref: ActorRef)
12   extends PersistentActor with AtLeastOnceDelivery {
13
14     override def receiveCommand: Receive = {
15         case Plus(amount) =>
16           persist(PlusEvent(amount))(updateState)
17         case Confirm(id) =>
18           persist(ConfirmEvent(id))(updateState)
19     }
20
21     override def receiveRecover: Receive = {
22         case e : Event => updateState(e)
23     }
24
25     def updateState(e: Event): Any = e match {
26         case PlusEvent(amount) =>
27           deliver(ref.path)(id => CountCommand(id, amount))
28         case ConfirmEvent(id) =>
29           confirmDelivery(id)
30     }
31
32     override def persistenceId: String = "actor-1"
33 }
34
35 class Accumulator extends Actor {
36     var count: Int = 0
37
38     override def receive: Receive = {
39         case CountCommand(id: Long, amount: Int) =>
40             count = count + amount
41             sender() ! Confirm(id)
42         case "result" =>
43             sender() ! count
44     }
45 }
```

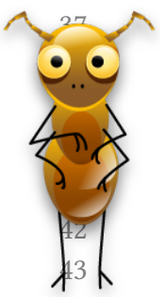
regular message handler

recovery handler used to rehydrate state from journal

message send with at-least-once delivery

framework can stop resending

message processing acknowledgement sent back

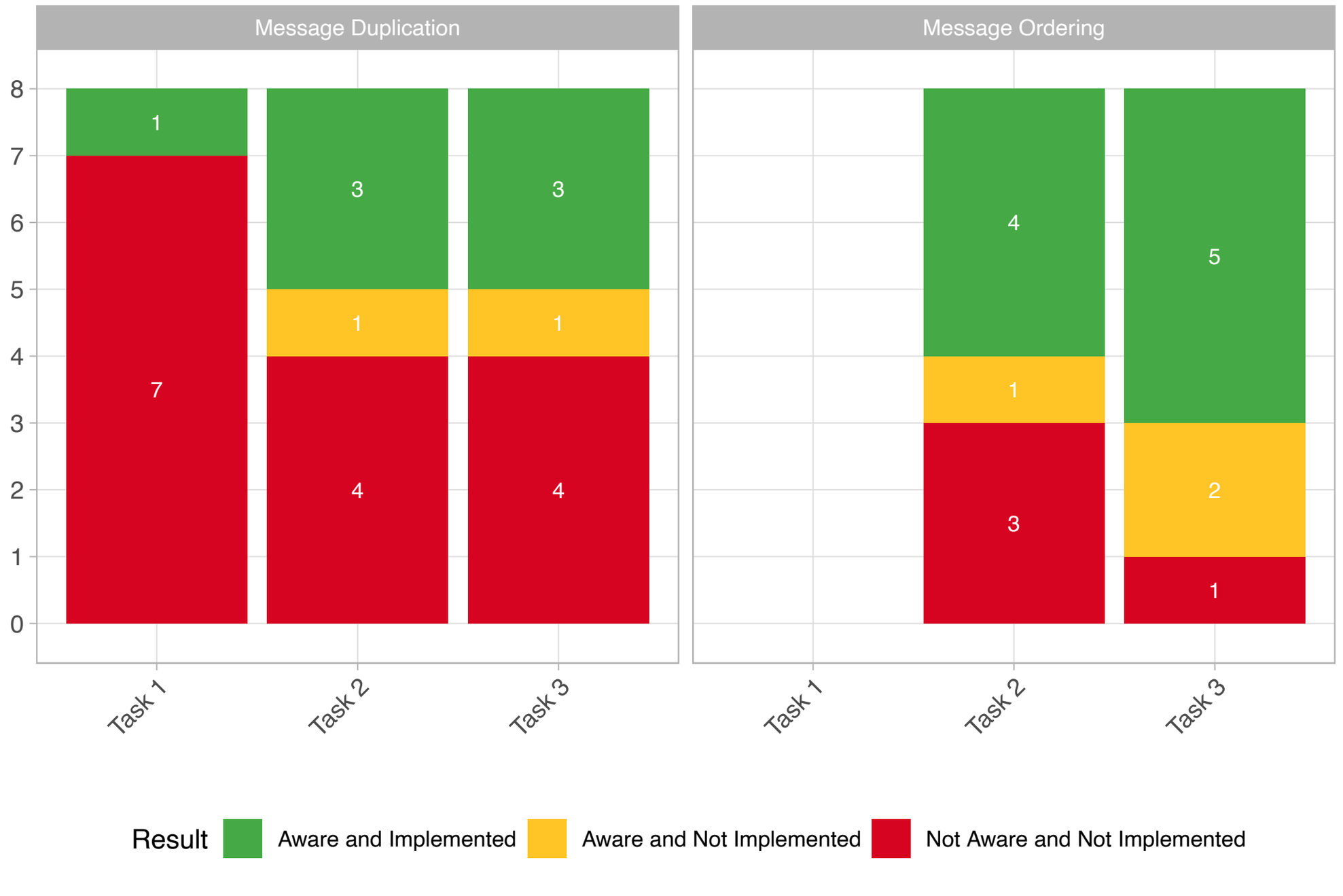


... even by PhD students

Ongoing research

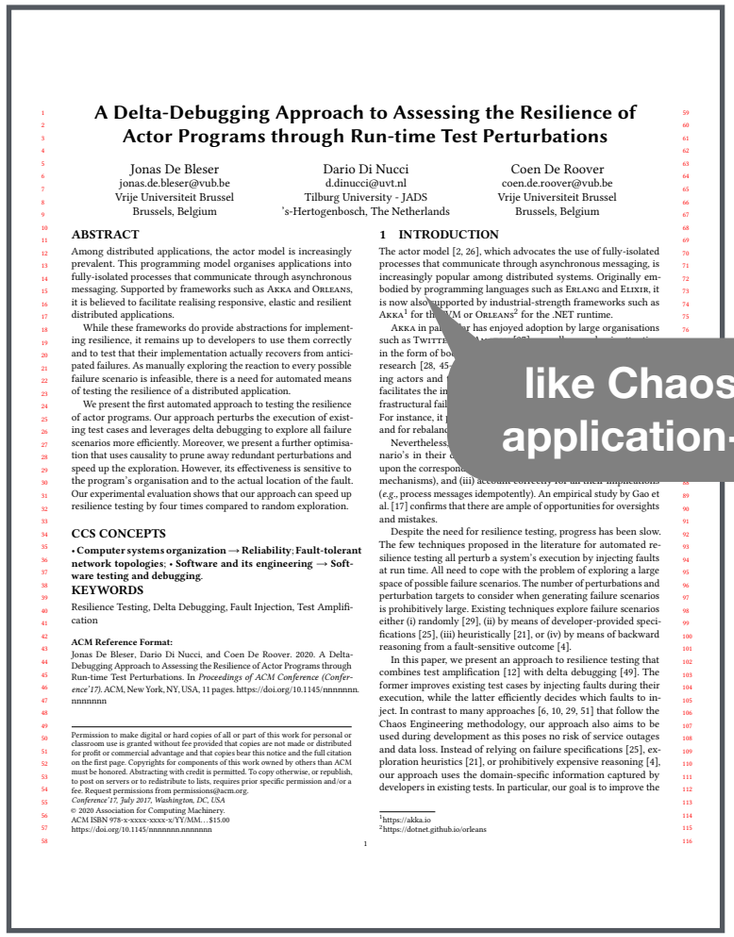
In summary

The participants of our study became increasingly aware about message duplication and message ordering over the study period. However, only a minority of them were able to correctly implement resilient systems. Most of the participants consider tool support for resilience testing as important but under certain conditions.



Ongoing research: resilience testing

Ongoing research



A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations

Jonas De Bleser
jonas.de.bleser@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Dario Di Nucci
d.dinucci@uvt.nl
Tilburg University - JADS
's-Hertogenbosch, The Netherlands

Coen De Roover
coen.de.roover@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Among distributed applications, the actor model is increasingly prevalent. This programming model organises applications into fully-isolated processes that communicate through asynchronous messaging. Supported by frameworks such as Akka and Orleans, it is believed to facilitate realising responsive, elastic and resilient distributed applications.

While these frameworks do provide abstractions for implementing resilience, it remains up to developers to use them correctly and to test that their implementation actually recovers from anticipated failures. As manually exploring the reaction to every possible failure scenario is infeasible, there is a need for automated means of testing the resilience of a distributed application.

We present the first automated approach to testing the resilience of actor programs. Our approach perturbs the execution of existing test cases and leverages delta debugging to explore all failure scenarios more efficiently. Moreover, we present a further optimisation that uses causality to prune away redundant perturbations and speed up the exploration. However, its effectiveness is sensitive to the program's organisation and to the actual location of the fault. Our experimental evaluation shows that our approach can speed up resilience testing by four times compared to random exploration.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Fault-tolerant network topologies**; • **Software and its engineering** → **Software testing and debugging.**

KEYWORDS

Resilience Testing, Delta Debugging, Fault Injection, Test Amplification

ACM Reference Format:

Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2020. A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The actor model [2, 26], which advocates the use of fully-isolated processes that communicate through asynchronous messaging, is increasingly popular among distributed systems. Originally embodied by programming languages such as Erlang and Elixir, it is now also supported by industrial-strength frameworks such as Akka[1] for the JVM or Orleans[2] for the .NET runtime.

Akka in pa… has enjoyed adoption by large organisations such as Twitte… in the form of bo… research [28, 45… ing actors and … facilitates the in… frastructural fail… For instance, it … and for rebalanc…

Nevertheless… nario's in their … upon the correspon… mechanisms), and (iii) … (e.g., process messages idempotently). An empirical study by Gao et al. [17] confirms that there are ample of opportunities for oversights and mistakes.

Despite the need for resilience testing, progress has been slow. The few techniques proposed in the literature for automated resilience testing all perturb a system's execution by injecting faults at run time. All need to cope with the problem of exploring a large space of possible failure scenarios. The number of perturbations and perturbation targets to consider when generating failure scenarios is prohibitively large. Existing techniques explore failure scenarios either (i) randomly [29], (ii) by means of developer-provided specifications [25], (iii) heuristically [21], or (iv) by means of backward reasoning from a fault-sensitive outcome [4].

In this paper, we present an approach to resilience testing that combines test amplification [12] with delta debugging [49]. The former improves existing test cases by injecting faults during their execution, while the latter efficiently decides which faults to inject. In contrast to many approaches [6, 10, 29, 51] that follow the Chaos Engineering methodology, our approach also aims to be used during development as this poses no risk of service outages and data loss. Instead of relying on failure specifications [25], exploration heuristics [21], or prohibitively expensive reasoning [4], our approach uses the domain-specific information captured by developers in existing tests. In particular, our goal is to improve the

[1] https://akka.io
[2] https://dotnet.github.io/orleans

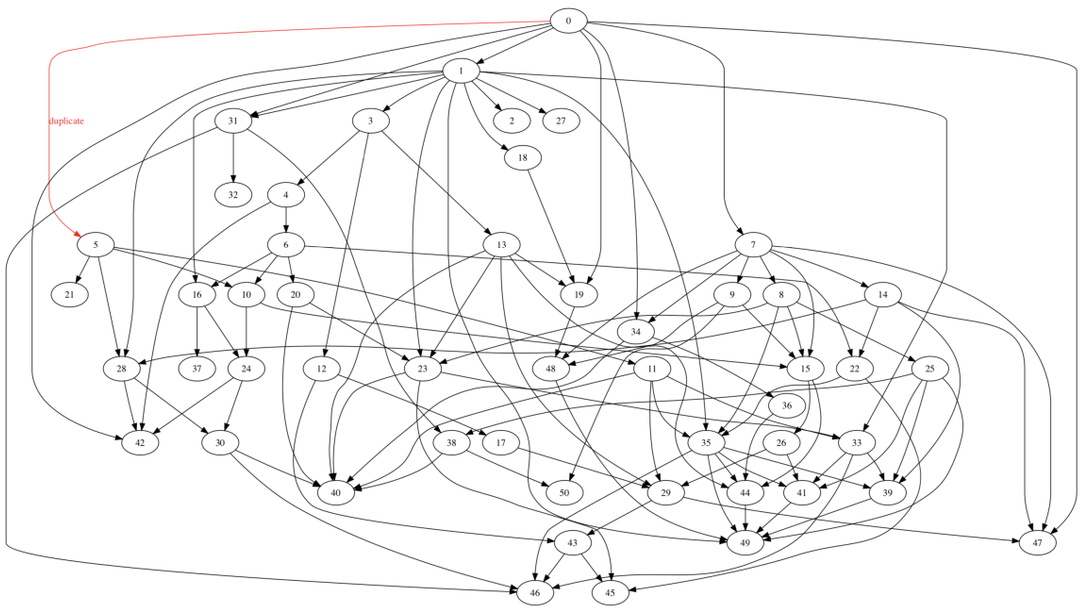
like Chaos Engineering, but at the application-level and during testing

[Submitted to AST2020]

1. execute test and record outcome
2. analyse execution trace for all perturbation targets
3. until change in test outcome or test budget exhausted:
   re-execute test under adverse conditions
   - terminate persistent actor after a message has been processed
   - duplicate or delay at-least-once delivery messages

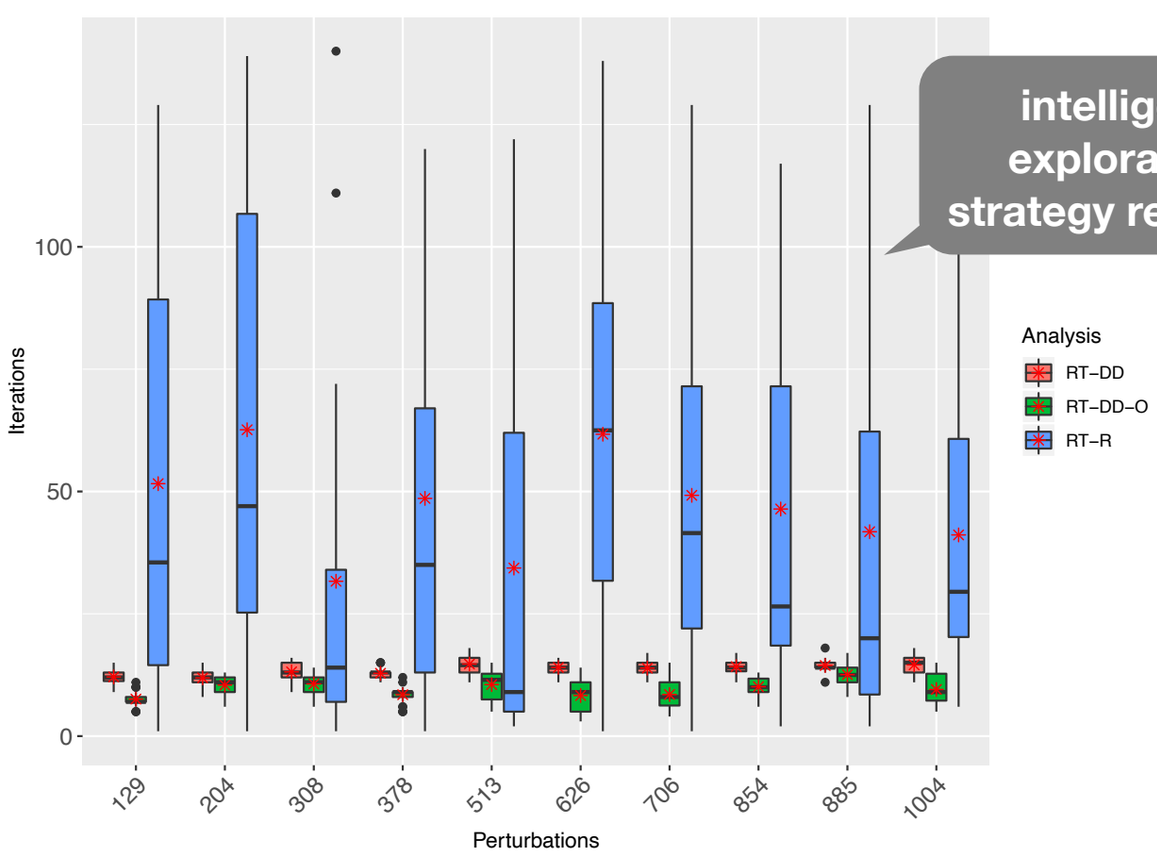
Ongoing research: resilience testing

Ongoing research

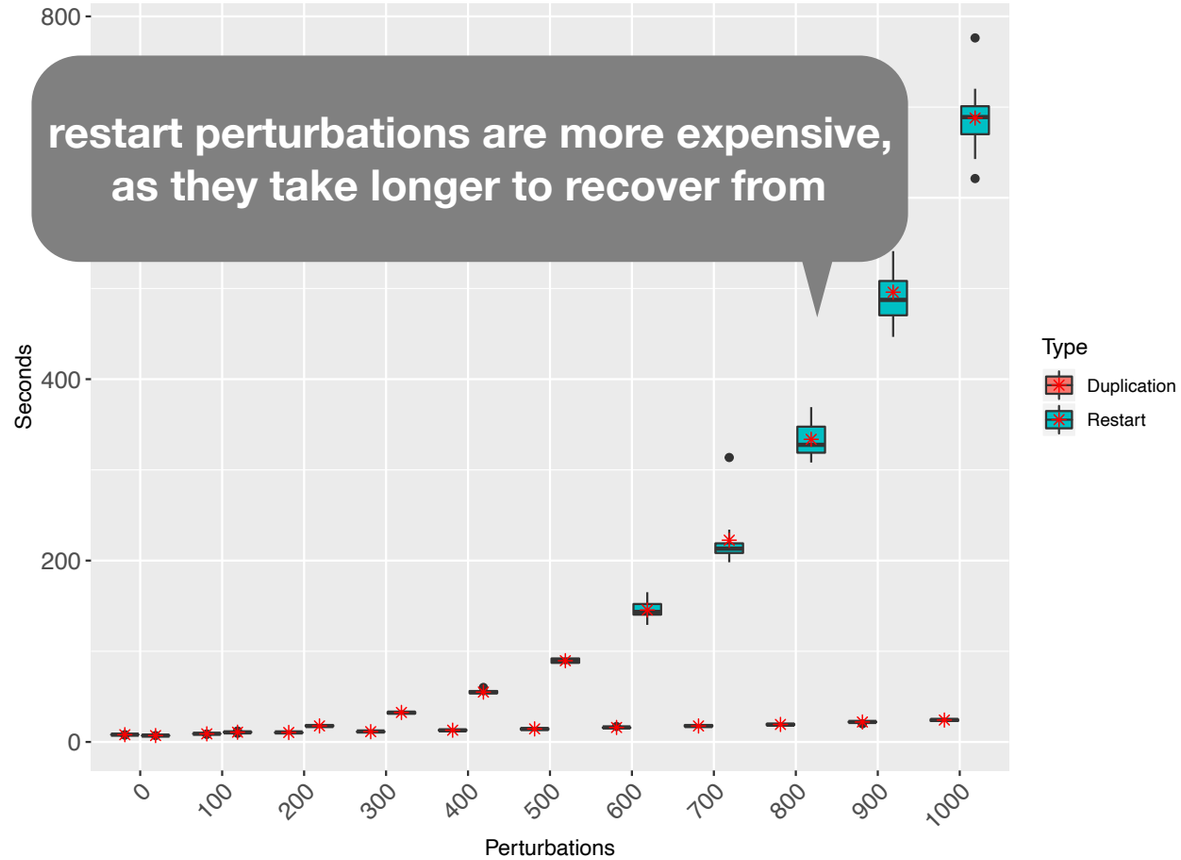


Messages		258	408	616	378	1026	626	706	854	1770	
Resilience Defect		D	D	D	R	D	R	R	R	D	
Perturbations		129	204	308	378	513	626	706	854	885	100
RT-R	A	52	63	32	49	34	62	49	46	42	41
	M	36	47	14	35	9	62	42	26	20	30
	T	0	4	13	1	13	6	2	6	12	14
RT-DD	A	12	12	13	13	15	14	14	14	14	15
	M	12	12	13	13	14	14	14	14	14	15
	T	0	0	0	0	0	0	0	0	0	0
RT-DD-O	A	8	10	11	8	11	8	8	10	13	10
	M	7	11	11	9	12	9	8	10	12	9
	T	0	0	0	0	0	0	0	0	0	0

automatically-generated + defect-seeded systems

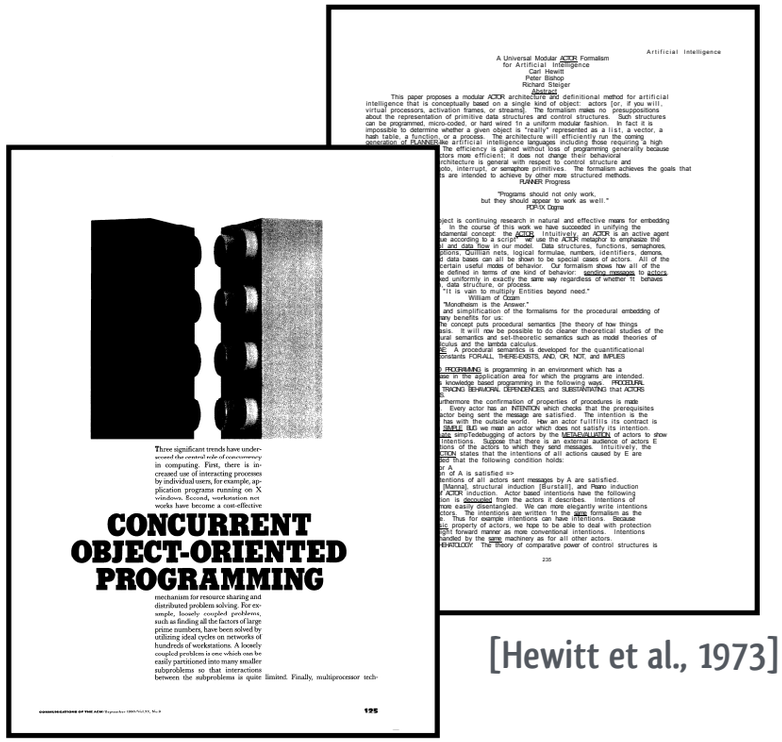


intelligent exploration strategy required



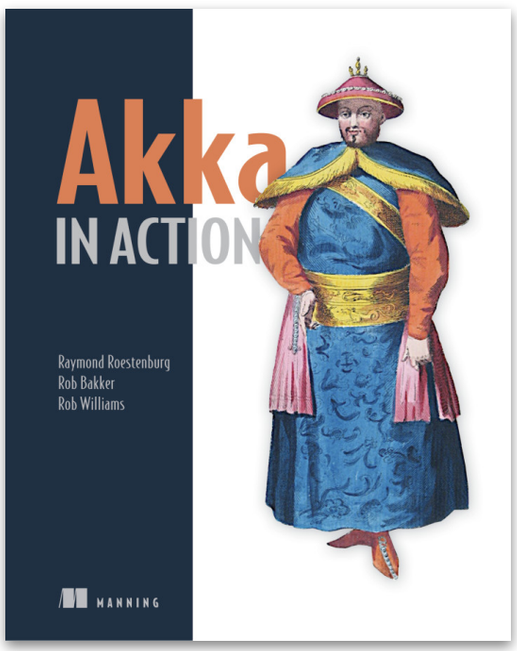
restart perturbations are more expensive, as they take longer to recover from

For those who would like to know more



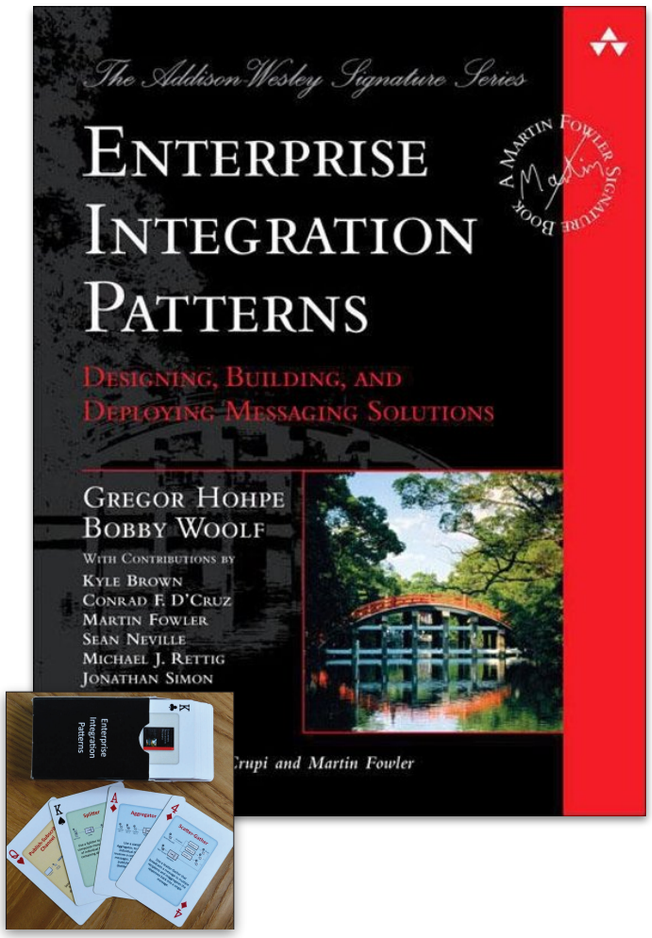
[Hewitt et al., 1973]

[Agha 1990]

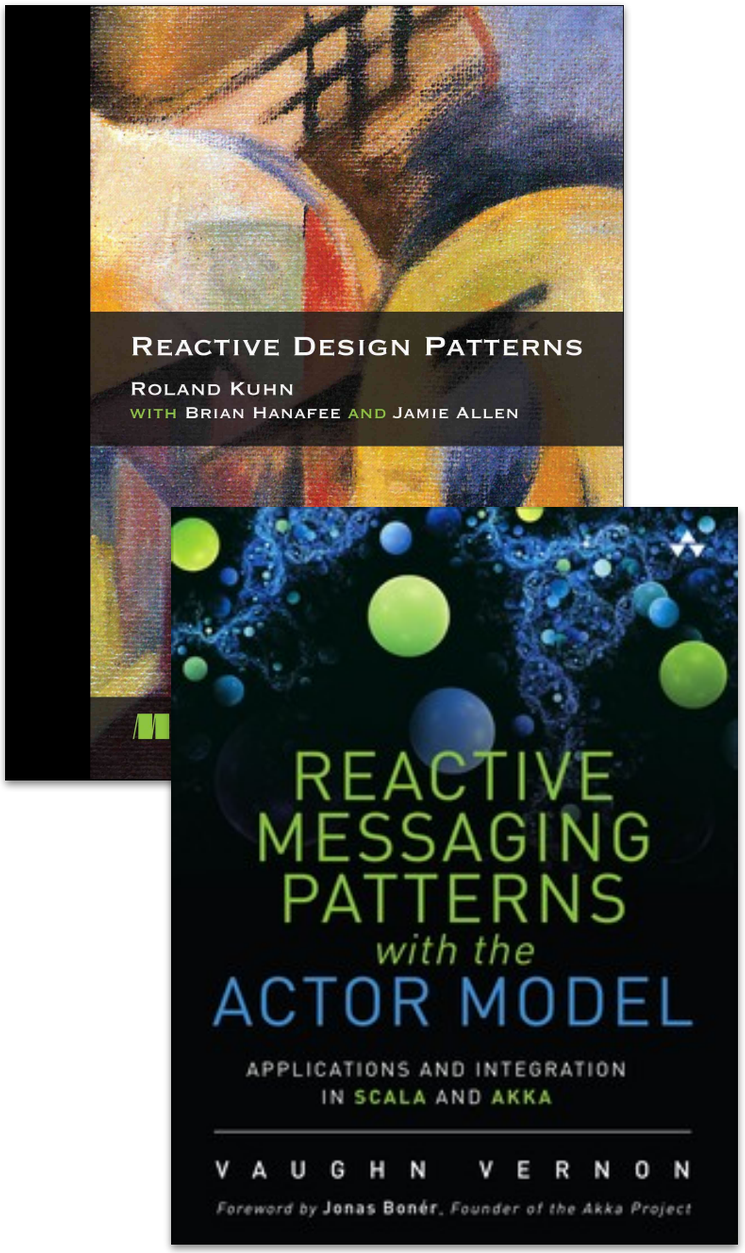


[Roestenburg et al. 2016]

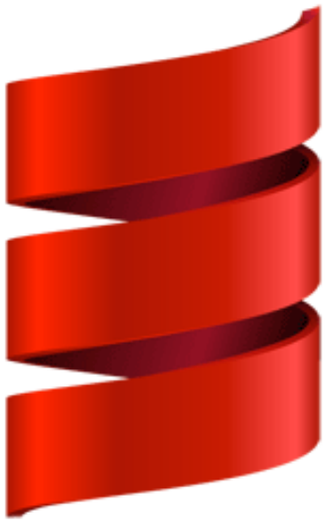
+



=



Take-away 1: programming language matters



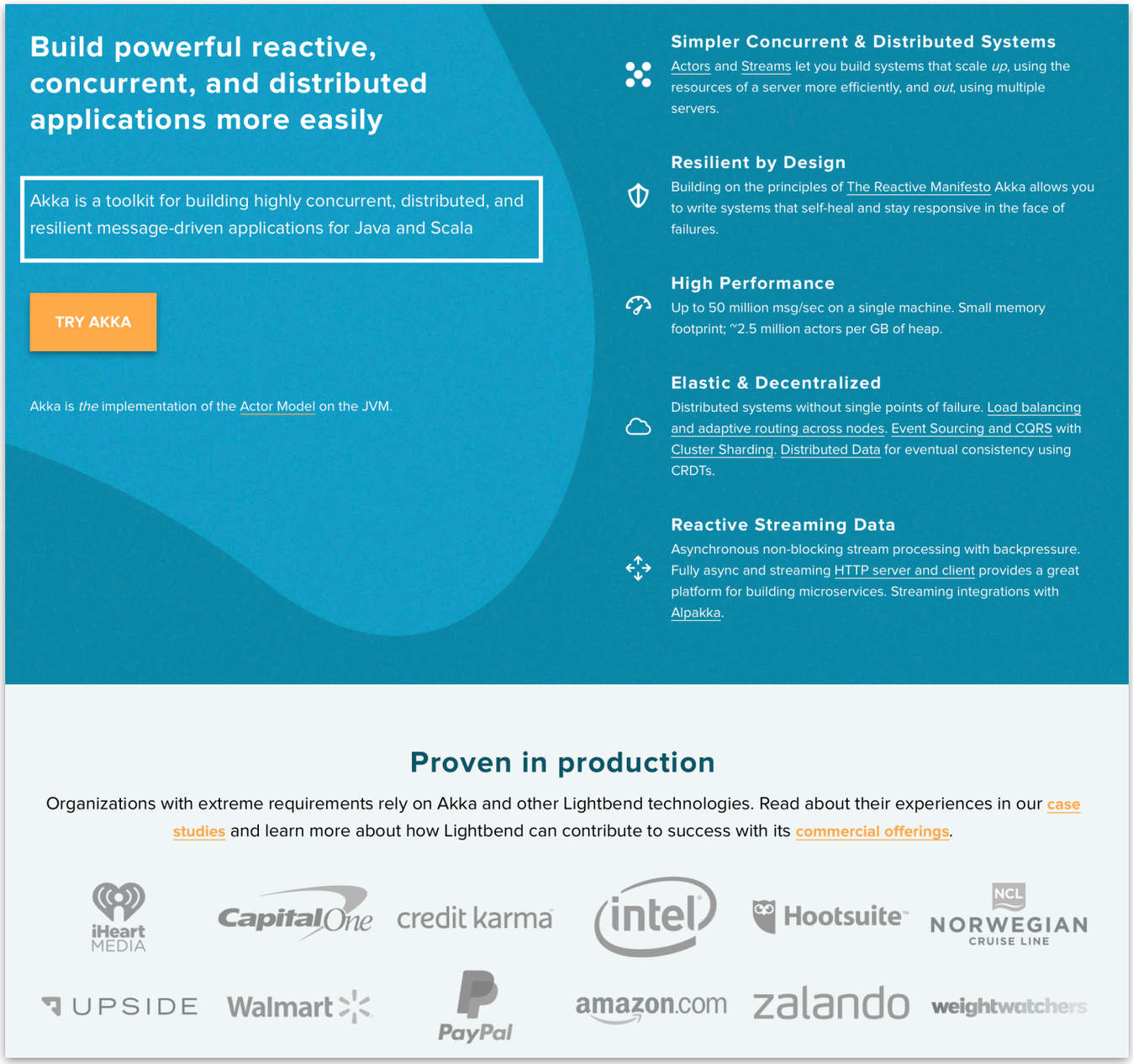
Scala

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

released in 2003 by Martin Odersky
professor at EPFL

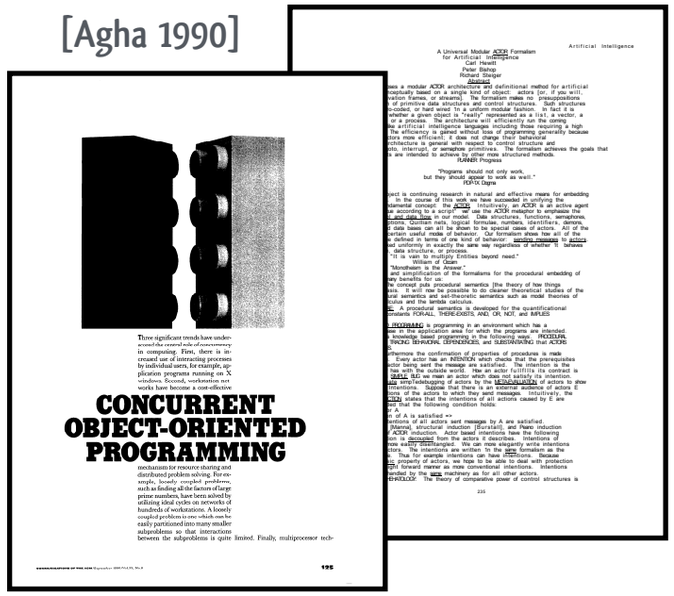
- Unifies and generalizes functional and object-oriented programming
- Features a strong static type system for safety
- Hosts multiple domain-specific languages
- Offers a read-eval-print loop for interactive prototyping
- Compatible with existing languages for the JVM

Take-away 2: programming model matters



[Hewitt et al., 1973]

[Agha 1990]



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

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.

[Zimmermann et al., IEEE Software 2016]

Take-away 4: application-level + 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

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

1990's

SPAGHETTI-ORIENTED
ARCHITECTURE
(aka Copy & Paste)

2000's

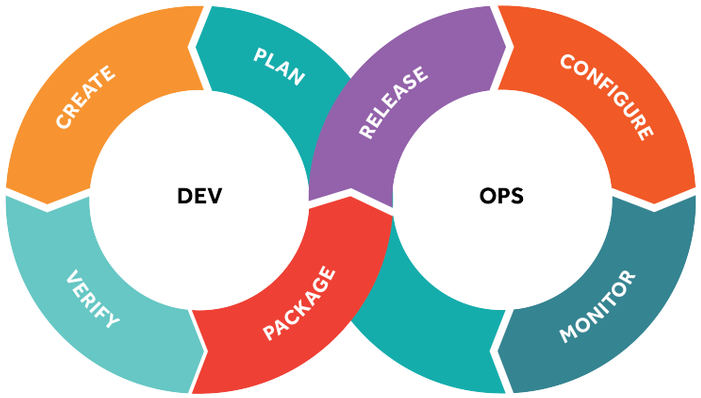
LASAGNA-ORIENTED
ARCHITECTURE
(aka Layered Monolith)

2010's

RAVIOLI-ORIENTED
ARCHITECTURE
(aka Microservices)

architecture

+



process

+

infrastructure