Introducing Reactive Programming

Start Coding with Practical Examples in Java and Scala

Duncan K. DeVore
Typesafe
@ironfish
github/ironfish

Outline

1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

Outline

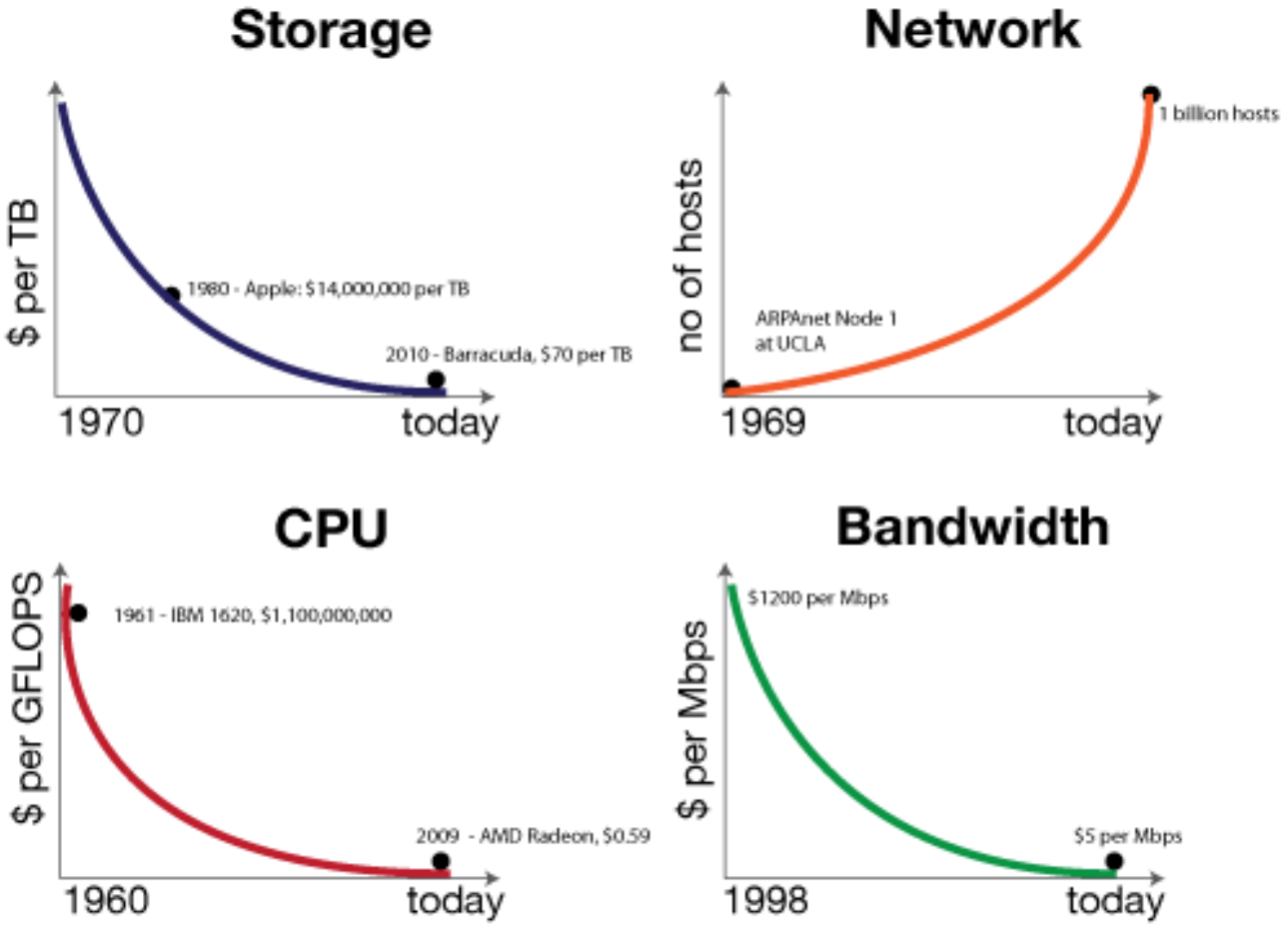
1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

different world

the problem

Yesterday	Today
Single machines	Clusters of machines
Single core processors	Multicore processors
Expensive RAM	Cheap RAM
Expensive disk	Cheap disk
Slow networks	Fast networks
Few concurrent users	Lots of concurrent users
Small data sets	Large data sets
Latency in seconds	Latency in milliseconds

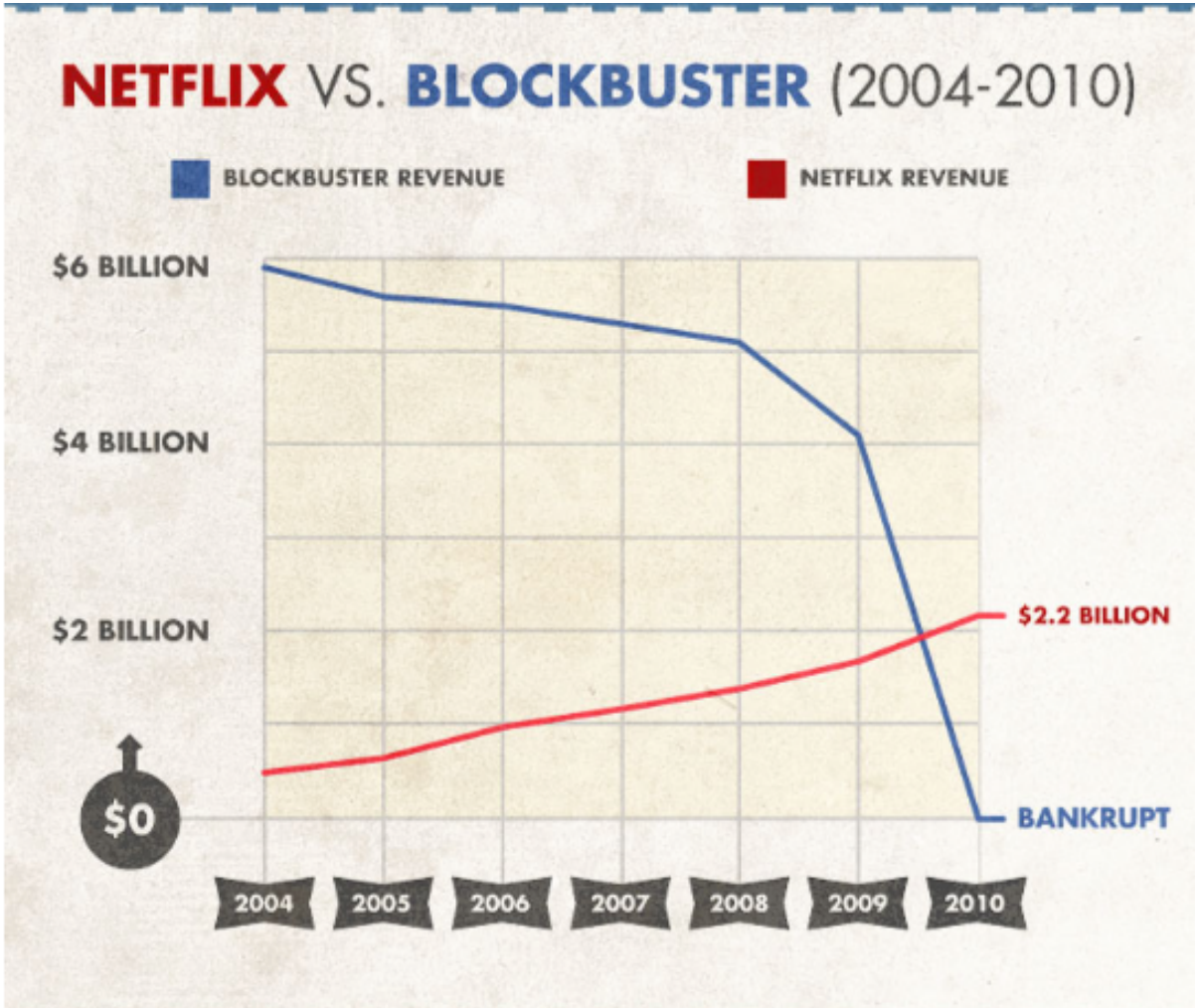
Yesterday vs Today



A study by MIT Sloan Management Review and Capgemini Consulting finds that companies now face a digital imperative: adopt new technologies effectively or face competitive obsolescence.

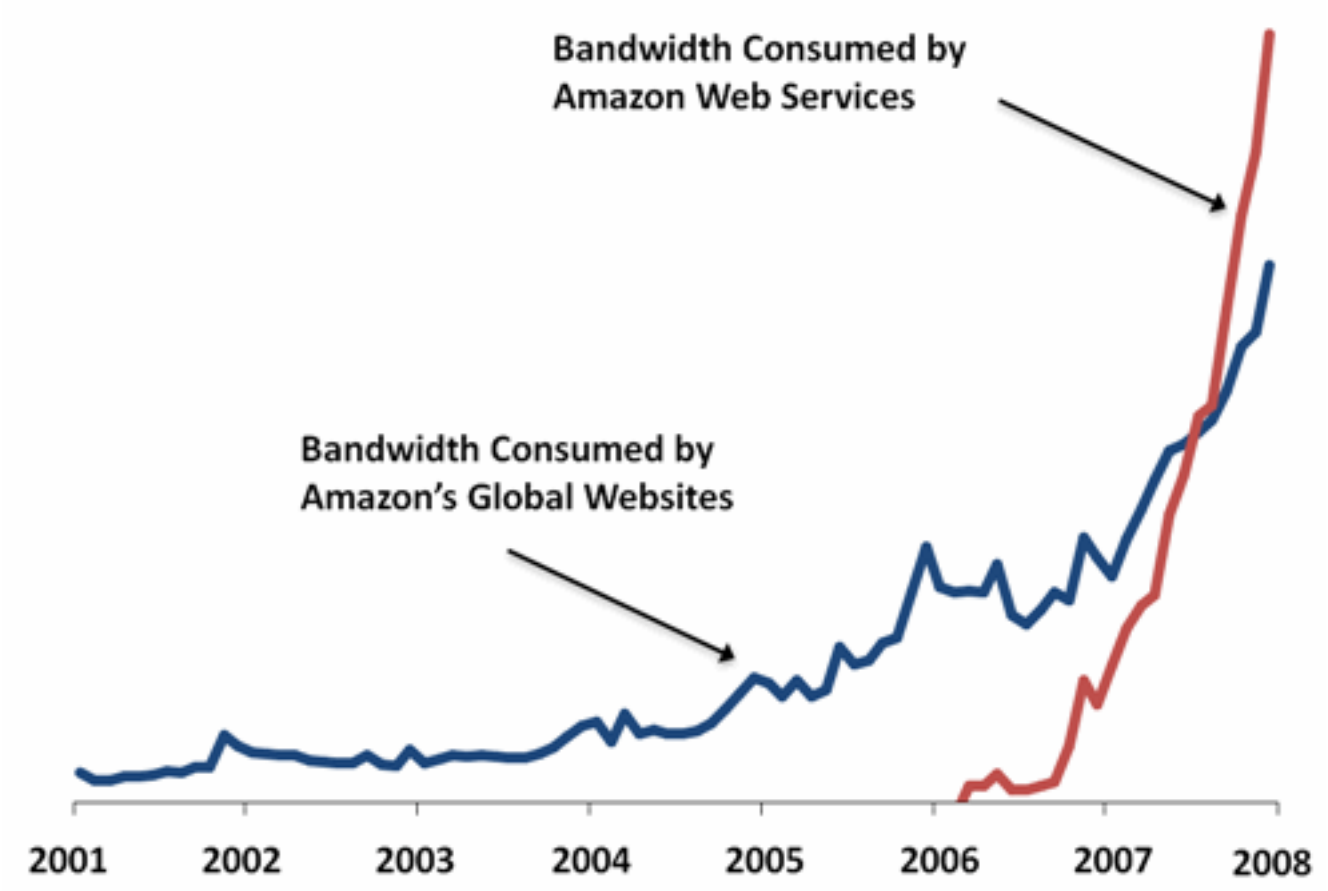
- October 2013

Case and Point



Case and Point

Amazon = Bookstore?



today's

golden rule!

“It's Not the Big That Eat the Small...It's the Fast That Eat the Slow”

- Jason Jennings and Laurence Haughten

what can

we do?

Outline

1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

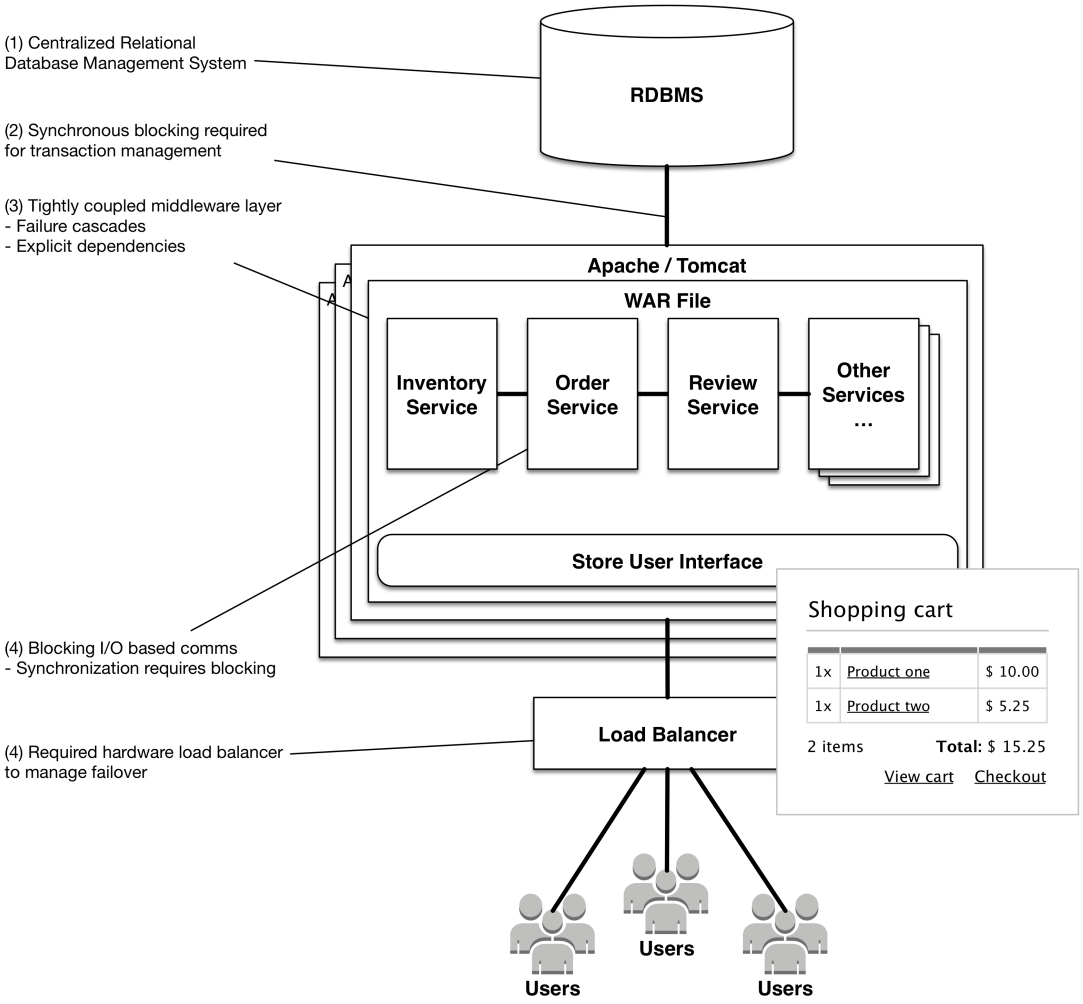
how about the

monolith?

*"Historically, since the dawn of web development, the majority of web applications have been designed based on a monolithic architecture. A monolithic architecture is an architecture where **functionally discernible aspects** of the system **are not** architecturally separate."*

- Reactive Application Development (Manning)

Monolithic Systems



Monolithic Systems

- Historically relies on ACID
- Blocks for transaction mgmt.
- Tightly coupled middleware
- Blocking based I/O comms
- Proactive approach to failure
- Relies on scaling up for growth
- Load balancer for distribution

Universal Scalability Law

Neil J. Gunther proved that the **blocking** (of any kind, anywhere) would actually **reduce concurrency** as a system is **scaled**. This holds true today and is called Gunther's Law but is widely known as The Universal Scalability Law.

This is due to two reasons:

1. ***Contention***; waiting for queues or shared resources
2. ***Coherency***; the delay for data to become consistent

how about

concurrency?

*"In computer science, **concurrency** is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other."*

-- Google

The Process

- > one program could run at once (not concurrently)
- Isolated independent execution of programs
- OS allocates resources (memory, file handles, etc.)
- Comms (sockets, shared memory, semaphores, etc.)
- Process schedulers
- Multi-tasking, time sharing

The Thread

- Multiple program control flow
- Coexist within the same process
- Path to hardware parallelism
- Simultaneous scheduling
- Run on multiple CPU's
- Non-sequential comp-model
- Multiple things @ once!
- But there are challenges...

Not Easy!

- Non-determinism
- Shared Mutable State
- Amdahl's Law
- Exponential growth of problem
- Very difficult to debug
- Complex locking system
- Just not fun!

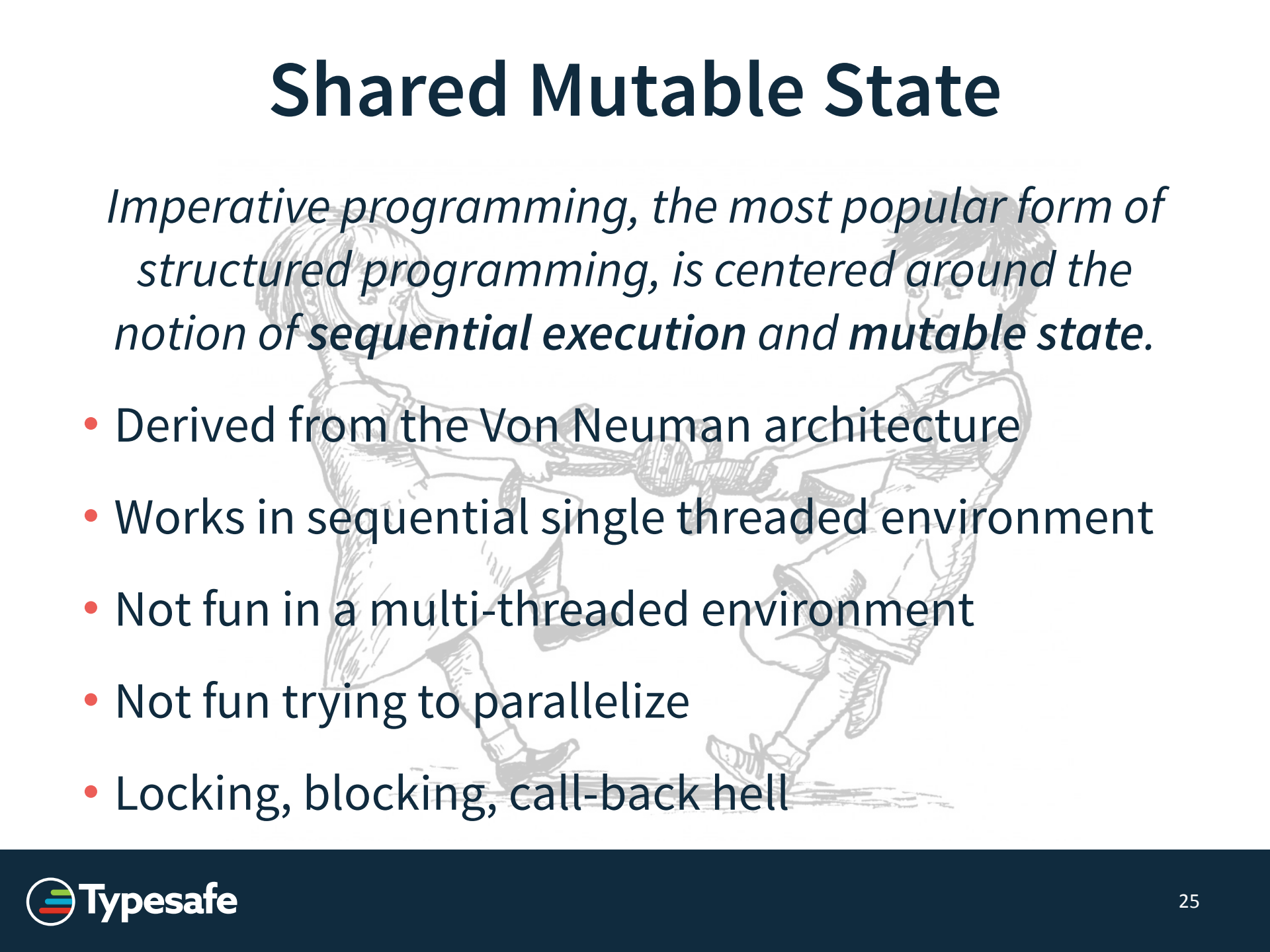
*"Although threads seem to be a small step from sequential computation, in fact, they represent a **huge step**. They discard the most essential and appealing properties of sequential computation: **understandability**, **predictability**, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of **pruning** that **nondeterminism**."*

-- The Problem with Threads, Edward A. Lee, Berkeley 2006

Shared Mutable State

*Imperative programming, the most popular form of structured programming, is centered around the notion of **sequential execution** and **mutable state**.*

- Derived from the Von Neuman architecture
- Works in sequential single threaded environment
- Not fun in a multi-threaded environment
- Not fun trying to parallelize
- Locking, blocking, call-back hell



Concurrency Definition (Real One)

Madness, mayhem, **heisenbug**, bohrbug, mandelbug and general all around pain an suffering.

-- me

how about

distributed computing?

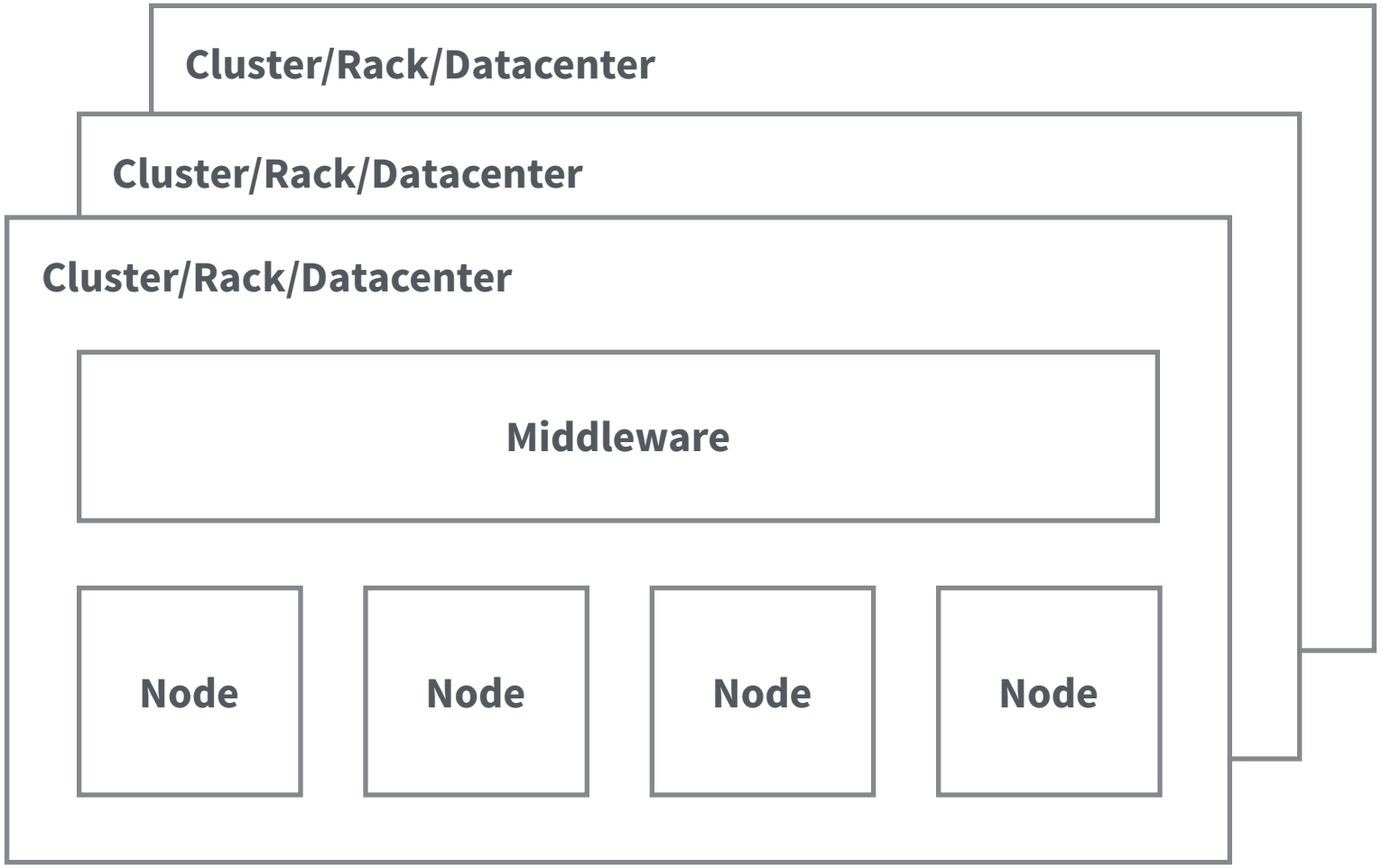
*"A **distributed system** is a software **system** in which components located on networked computers communicate and coordinate their actions by passing messages."*

-- Wikipedia

Distributed Computing

- The “new” normal
- Mobile computing
- Cloud services, REST etc.
- Clustering, multi-data denter
- NOSQL, NewSQL DBs
- Big Data

Distributed Systems



yes, it is the

solution!

but there

are problems!

The network is

Inherently Unreliable

http://aphyr.com/posts/288-the-network-is-reliable

Fallacies of Distributed Computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn’t change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

The Graveyard of Distributed Systems

- Distributed Shared Mutable State
  - $\Rightarrow$ EVIL^{N} (where N is number of nodes)
- Serializable Distributed Transactions
- Synchronous RPC
- Guaranteed Delivery
- Distributed Objects
  - *"Sucks like an inverted hurricane"* - Martin Fowler

what can

we do?

Outline

1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

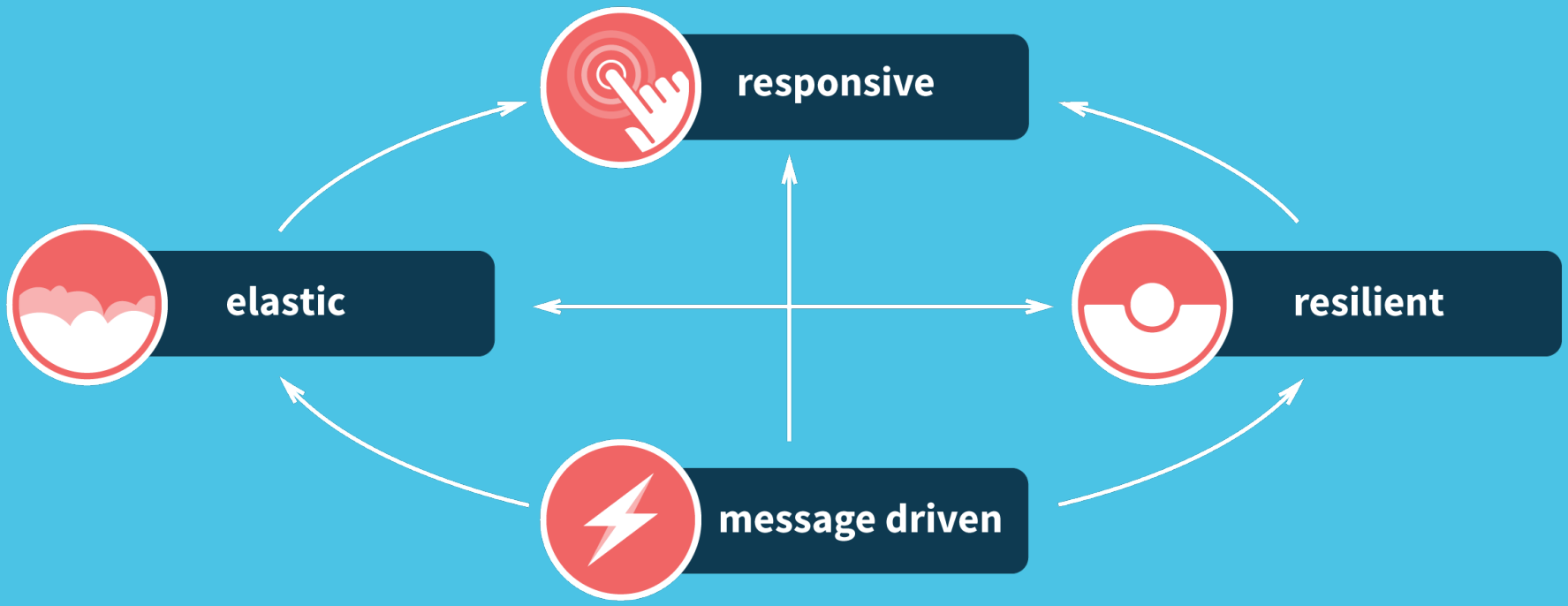
reactive systems

the solution

“In today's world, the demand for distributed systems has exploded. As customer expectations such as an immediate response, no failure, and access anywhere increase, companies have come to realize that distributed computing is the only viable solution.”

- Reactive Application Development (Manning)

Reactive Systems



Message Driven

message | ‘mesij |

a verbal, written or recorded communication sent to or left for a recipient who cannot be contacted directly

- Foundation for being elastic and resilient
- Based on asynchronous communication
- Sender design not affected by message propagation
- Recipient design not affected by message propagation
- You can design in isolation
- Leads to loosely coupled design

Elastic

elastic | i‘lasatik |

(an object or material) able to resume its normal shape spontaneously after contraction, dilatation, or distortion

- Key for being responsive
- Allocation / de-allocation of resources
- Reaction to increase / decrease in load
- Scale up or down by using 1..n cores
- Scala in or out by using 1..n machines
- Save money on unused computing power

Resilient

resilient | riˈzilyənt |

(an object or material) able to recoil or spring back into after bending, stretching, or being compressed

- Key for being responsive
- Embracing failure through expectation
- Supervisor hierarchy for strategic failure management
- Strategies can be implemented based on type of failure
- Self healing with multiple recovery paths
    - Start, stop, restart, escalate, custom…

Responsive

responsive | riˈspänsiv |

reacting quickly and positively; responding readily and with interest and enthusiasm

- Cornerstone of usability and utility
- Built on top of elastic resilient behavior
- Requires quick detection of problems
- Requires effective resolution of problems
- Result of effectively reacting to load
- Result of effectively reacting to failure

"Modern applications must embrace these changes by incorporating this behavior into their DNA".

- Reactive Application Development (Manning)

Outline

1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

what is

akka?

"Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM."

- Typesafe

The Toolkit

- Actor System
- Remoting
- Routing
- Clustering
- Cluster Sharding
- Akka Persistence
- Akka Streams
- Much more …

Actor System

The core Akka library akka-actor

- Hierarchical Structure
- Message passing
- Serialization
- Dispatching
- Routing
- Location transparency

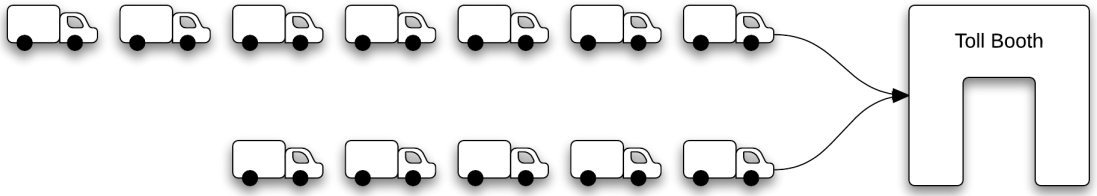
Remoting

Distributed by Default Mentality

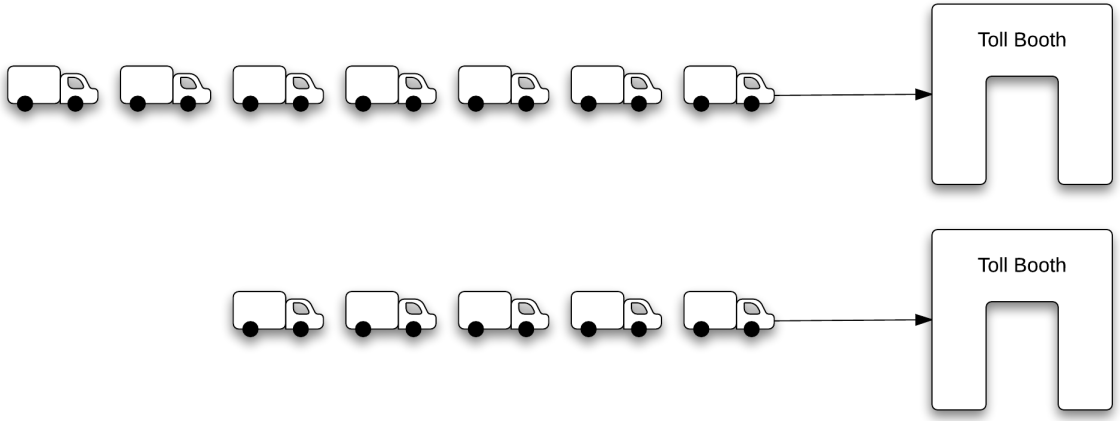
- Provides a **unified programming model**
- Employs **referential** or **location** transparency
- Local and remote use the **same API**
- Distinguished by **configuration**
- **Succinct** way to code **message-driven** apps

Routing

Concurrent Toll - Two Lanes, One Toll Booth



Parallel Toll - Two Lanes, Two Toll Booths



Clustering

Distributed by Default Mentality

- **Loosely coupled** group of systems
- Present themselves as a **single system**
- Resilient **decentralized peer-to-peer** based cluster
- No **single point** of failure
- No **single point** of bottleneck
- **Automatic** failure detection

Cluster Sharding

Distributed by Default Mentality

- Interact with actors that are **distributed** across several **nodes** in a cluster by a **logical identifier**.
- This access by logical identifier is **important** when your actors in **aggregate** consume **more resources** than can fit on **one** machine.
- Think Distributed Domain Driven Design!
- Automatic **activation** and **passivation**

Akka Persistence

Persistent State Management

- **Staple** for most applications
- Not only this, but also **automatic recovery**
- System **restarts** due to **failure** or **migration**
- Foundation for **Event Sourced** based systems
- Provides a foundation for **CQRS** based system

Akka Streams

- **Govern** the **exchange** of stream data
- Across **asynchronous** boundaries
- Manages **back-pressure**
- The **source** and **destination** work together
- **Eliminates** bottlenecks
- Provides a **traceable path**

akka with

java!

The Question Message

```
public final class Question extends Serializable {

    public static final long serialVersionUID = 1;
    public final String msg;

    public Question(String msg) {
        // quava preconditions can be used here
        this.msg = msg;
    }

    . . .
    // toString, equals and hashCode need to be implemented!
}
```

The Answer Message

```
public final class Answer extends Serializable {

    public static final long serialVersionUID = 1;
    public final String msg;

    public Answer(String msg) {
        // quava preconditions can be used here
        this.msg = msg;
    }

    . . .
    // toString, equals and hashCode need to be implemented!
}
```

The Student Actor

```
import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Props;
import akka.japi.pf.ReceiveBuilder;

public class Student extends AbstractLoggingAdapter {

    public final ActorRef teacher;
    public Student(final ActorRef teacher) {

        this.teacher = teacher;
        teacher.tell(new Question("What is the square root of pie?"), self());

        receive(ReceiveBuilder.matchAny(log().info(message)).build());
    }

    public static Props props(final ActorRef teacher) {
        return Props.create(Student.class, () -> new Student(teacher));
    }
}
```

The Teacher Actor

```
import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Props;
import akka.japi.pf.ReceiveBuilder;

public class Teacher extends AbstractLoggingAdapter {

    public Teacher() {
        receive(ReceiveBuilder.
            match(Question.class, question -> {
               sender().tell(getAnswer(question.msg), self());
            }).
            matchAny(this::unhandled).build()
        );
    }

    public static Props props(){
        return Props.create(Teacher.class, Teacher::new);
    }

    private Answer getAnswer(final Object message) {
        // some processing logic to derive answer, should handle errors
    }
}
```

Creating the Actors

```
public class SchoolApp {
    . . . main(final String[] args) . . .

    private final ActorSystem system;
    private final LoggingAdapter log;
    private final ActorRef teacher;
    private final ActorRef student;

    public School(final ActorSystem system) {
        this.system = system;
        log = Logging.getLogger(system, getClass().getName());
        teacher = createTeacher();
        student = createStudent();
    }

    protected ActorRef createTeacher() {
        return system.actorOf(Teacher.props(), “teacher”);
    }

    protected ActorRef createStudent() {
        return system.actorOf(Student.props(teacher), “student”);
    }
}
```

akka with

scala!

The Question Message

```
final case class Question(msg: String)
```

The Answer Message

```
final case class Answer(msg: String)
```

The Student Actor

```
import akka.actor.{ Actor, ActorLogging, ActorRef, Props }

object Student {
  def props(teacher: ActorRef): Props =
    Props(new Student(teacher))
}

class Student(teacher: ActorRef) extends Actor with ActorLogging {
  import Student._

  teacher ! Question("What is the square root of pie?")

  override def receive: Receive = {
    case msg => log.info(msg)
  }
}
```

The Teacher Actor

```
import akka.actor.{ Actor, ActorLogging, Props }

object Teacher {
  def props: Props =
    Props(new Teacher)
}

class Teacher extends Actor with ActorLogging {

  override def receive: Receive = {
    case q:Question => sender() ! getAnswer(q.msg)
    case _          => log.info("received unexpected message")
  }

  private def getAnswer(question: String): Answer = {
    // some processing logic to derive answer, should handle errors
  }
}
```

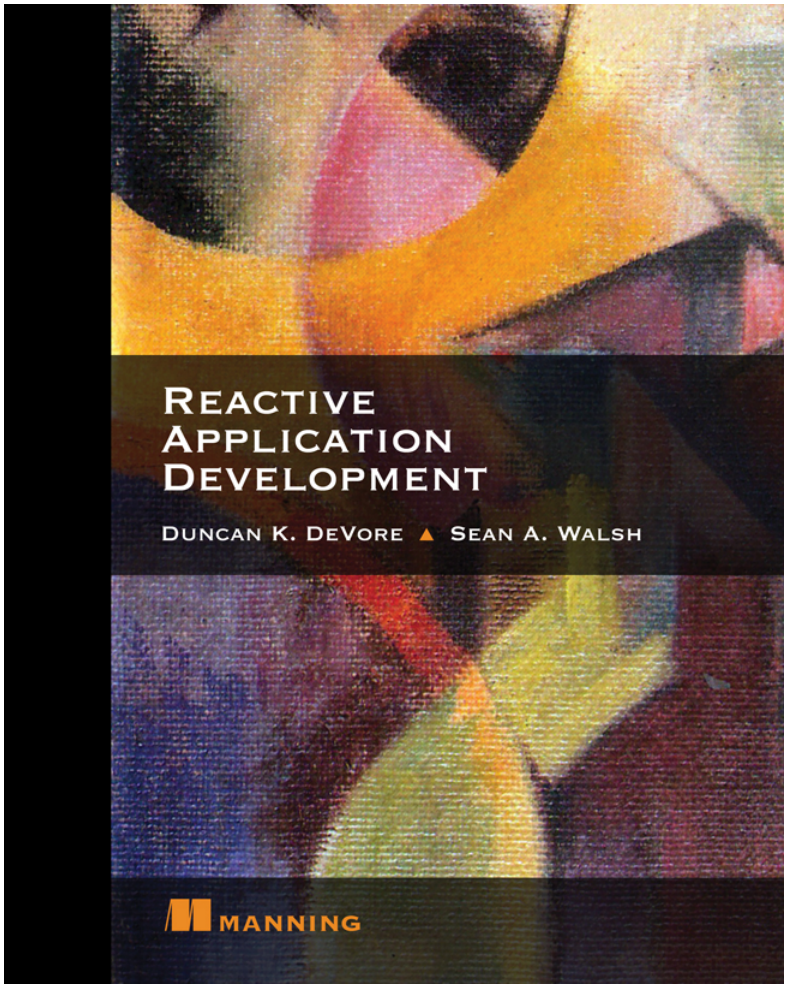
Creating the Actors

```
object SchoolApp {
  def main(args: Array[String]): Unit = {
    val system = ActorSystem(“school-system”)
    val schoolApp: SchoolApp = new SchoolApp(system)
 }
}

class SchoolApp(system: ActorSystem) {
  private val final log = Logging(system, getClass.name)
  private val teacher: ActorRef createTeacher()
  private val student: ActorRef createStudent()

  protected def createTeacher(): ActorRef {
    system.actorOf(Teacher.props, “teacher”);
  }

  protected def createStudent(): ActorRef {
    system.actorOf(Student.props(teacher), “student”);
  }
}
```

Outline

1. Different World
2. The Challenges
3. Reactive Systems
4. Akka :: Java :: Scala
5. Conclusion

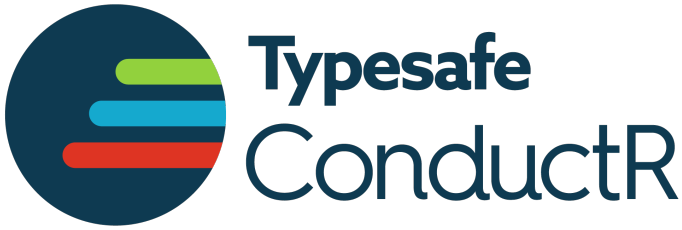
Reactive Application Development



Resources

- Reactive Application Development
  manning.com/devore/?a_aid=ironfish&a_bid=39e254aa
- Presentation Code Examples from Book
  https://github.com/ironfish/reactive-application-development-scala
- The Typesafe Reactive Platform
  www.typesafe.com/products/typesafe-reactive-platform
- Akka.io
  akka.io/
- Akka :: Java
  doc.akka.io/docs/akka/current/java.html?_ga=1.174827046.564944632.1421682545
- Akka :: Scala
  doc.akka.io/docs/akka/current/scala.html?_ga=1.140608438.564944632.1421682545

Resources



Today's demands on Operations are simply not met by yesterday's software architectures and technologies

ConductR is a management tool for Operations to deploy and control Reactive applications running in a cluster. Supplementing existing tools that focus on more traditional, monolithic application architectures, ConductR is built specifically for distributed Reactive applications, reflecting that Operations requires a different approach to both the tools and methodologies they use.

- Typesafe ConductR
  www.typesafe.com/products/conductr
- Typesafe ConductR Evaluation Webinar
  info.typesafe.com/WBN-2015-05-27-Evaluating-TS-ConductR_LP-Reg.html

questions?

Introducing Reactive Programming

Start Coding with Practical Examples in Java and Scala

Duncan K. DeVore
Typesafe
@ironfish
github/ironfish