Akka for Java Devs

Bridging the Imagination Divide

Duncan K. DeVore
Typesafe
@ironfish

Outline

1. Distributed Now?
2. The Challenges
3. Akka :: Java
4. Conclusion

Outline

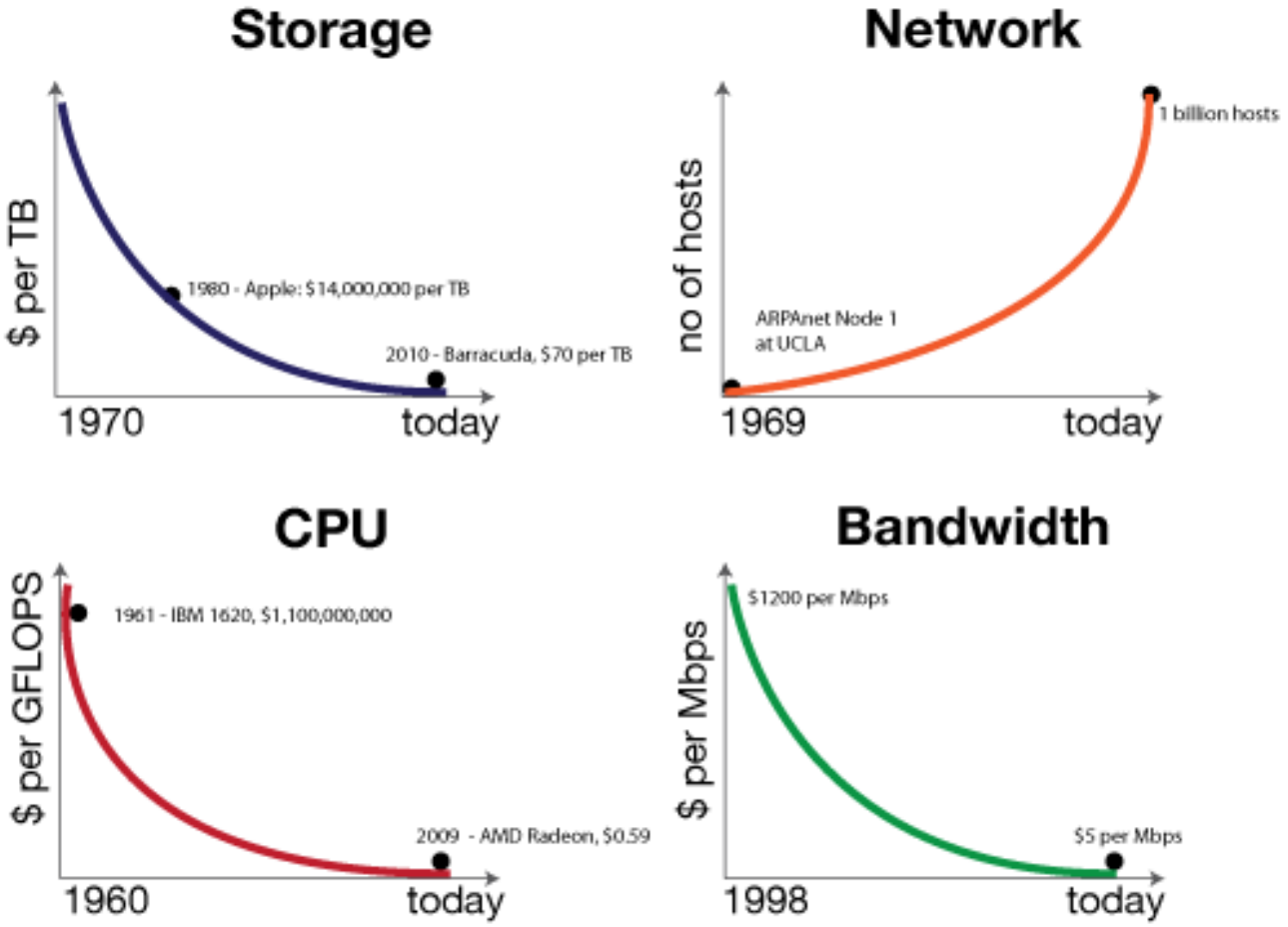
1. Distributed Now?
2. The Challenges
3. Akka :: Java
4. Conclusion

It’s a different world

out there

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

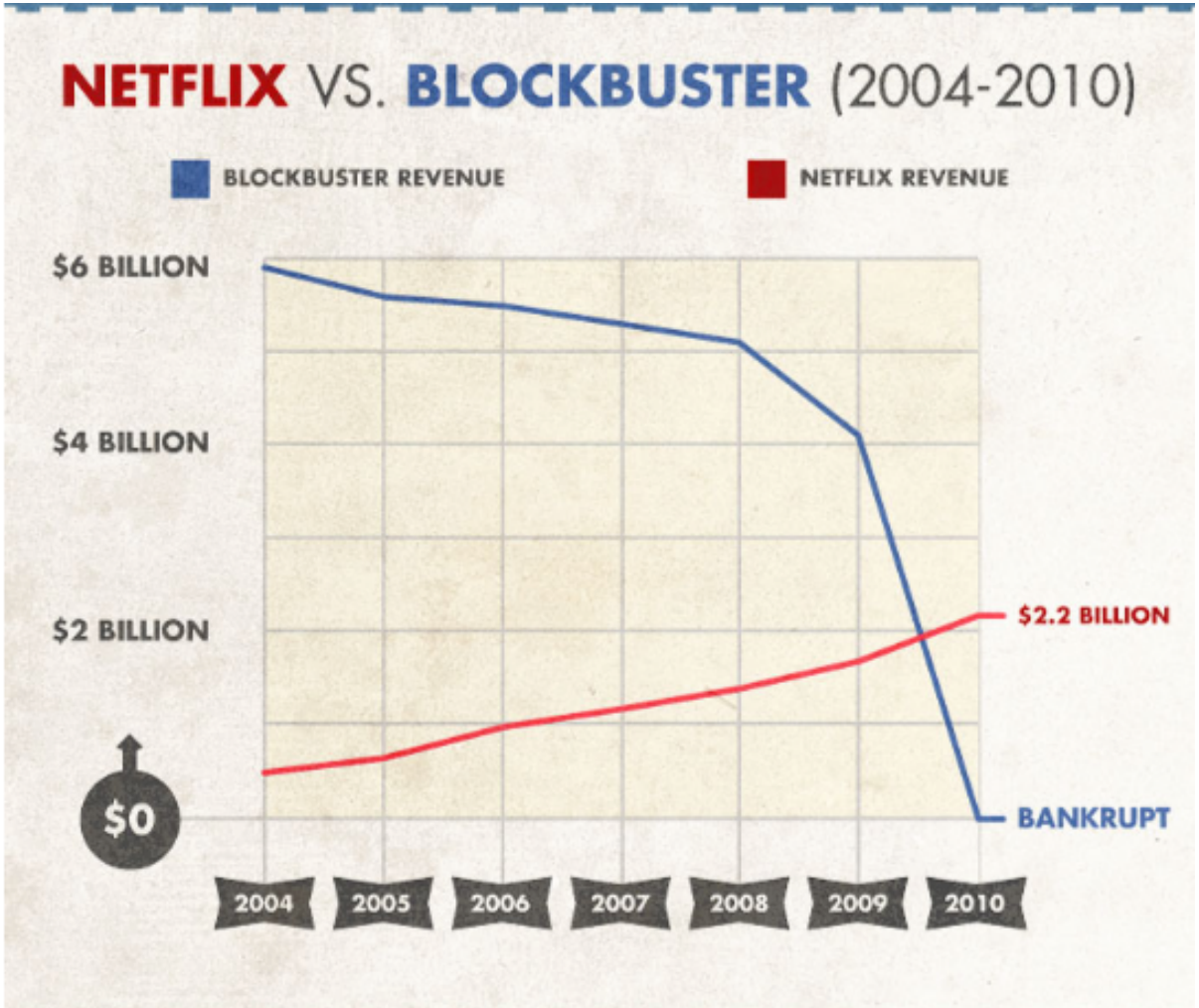
Yesterday vs Today



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.

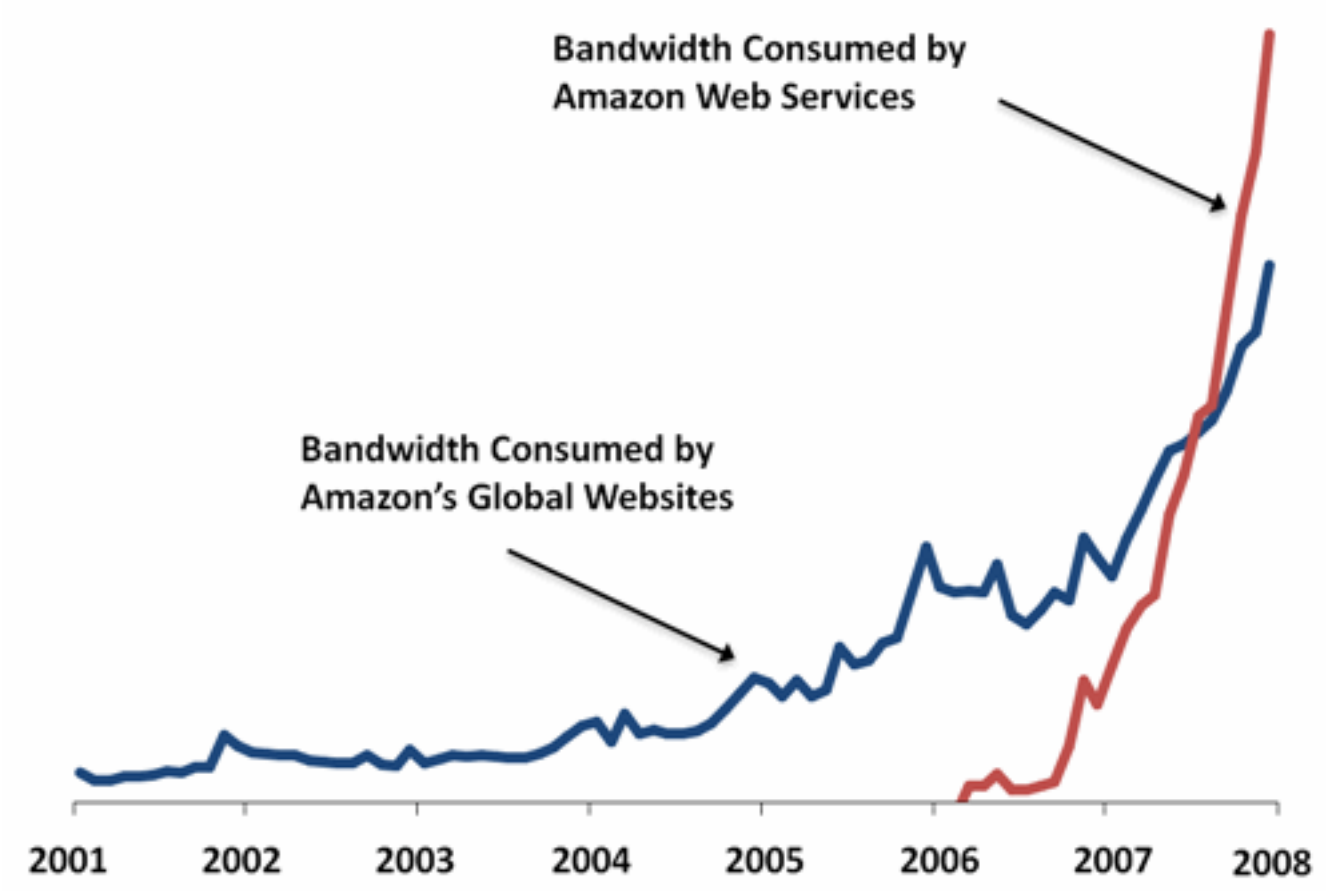
- October 2013

Case and Point



Case and Point

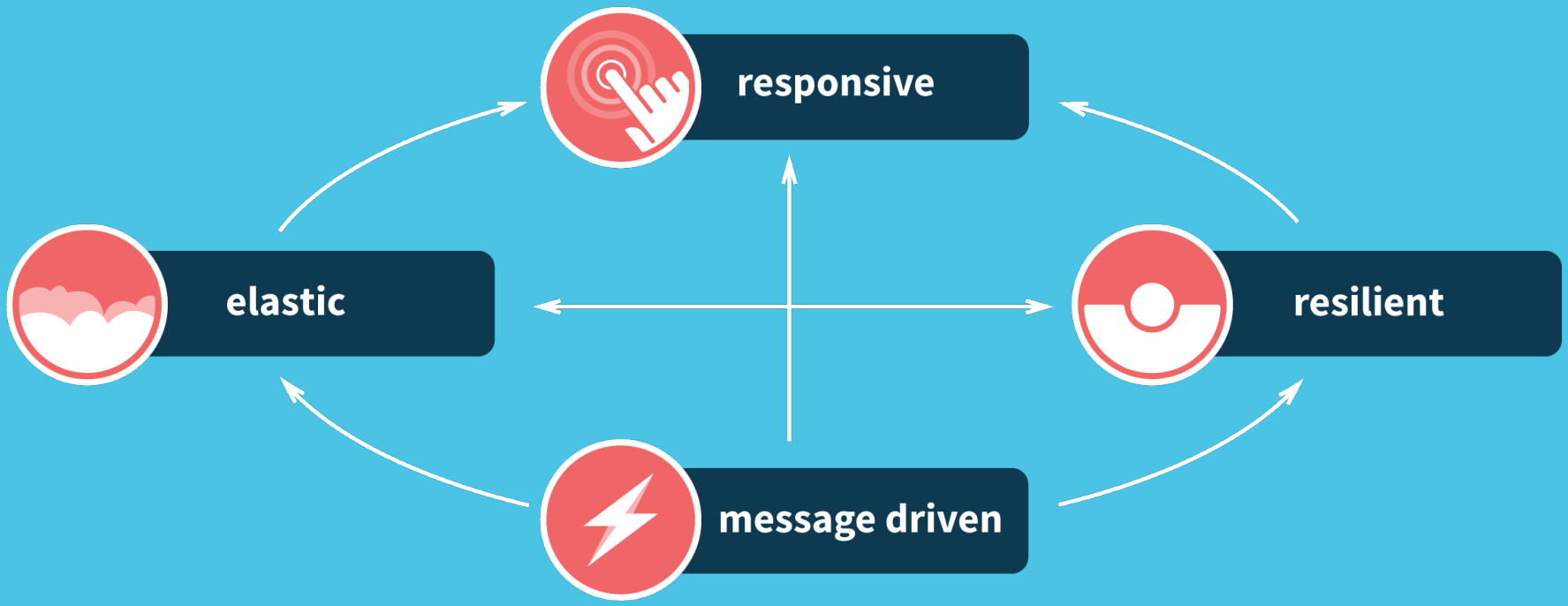
Amazon = Bookstore?



“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



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

- Reactive Application Development (Manning)

Outline

1. Distributed Now?
2. The Challenges
3. Akka :: Java
4. Conclusion

what is **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

- More than one program could run at once (not concurrently)
- Isolated independent execution of programs
- OS would allocate resources (memory, file handles, etc.)
- Communication (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

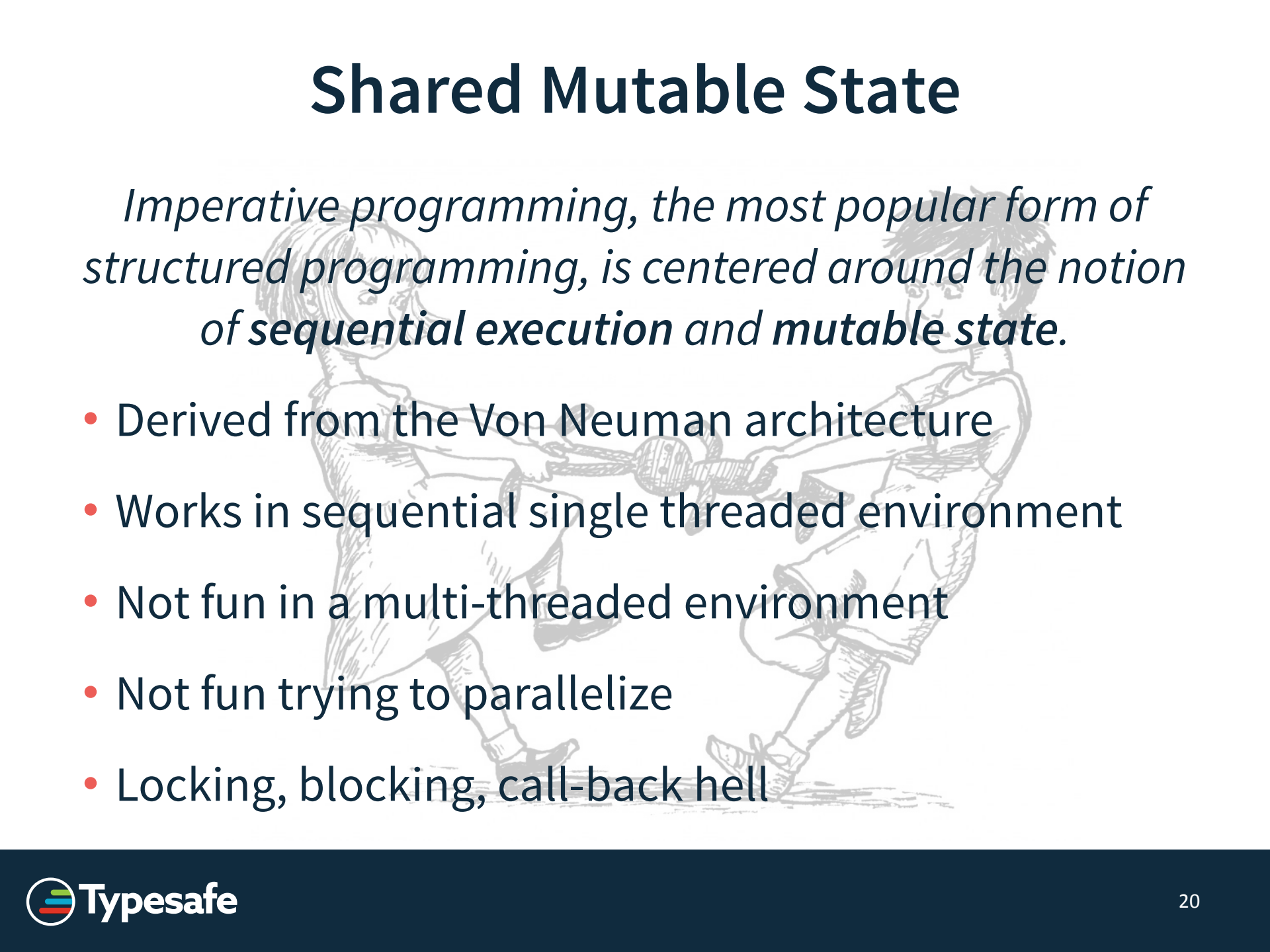
*"Although threads seem to be a small step from sequential computation, in fact, they represent a **huge step**. They discard the most essential and appealing properties of sequential computation: **understandability**, **predictability**, and determinism. Threads, as a model of computation, are wildly 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

what is **distribution?**

*"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

is the new normal

- Mobile
- Cloud Services, REST etc.
- Clustering, Multi-Data Center
- NOSQL DBs
- Big Data

The network is

Inherently Unreliable

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

Outline

1. Distributed Now?
2. The Challenges
3. Akka :: Java
4. 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

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

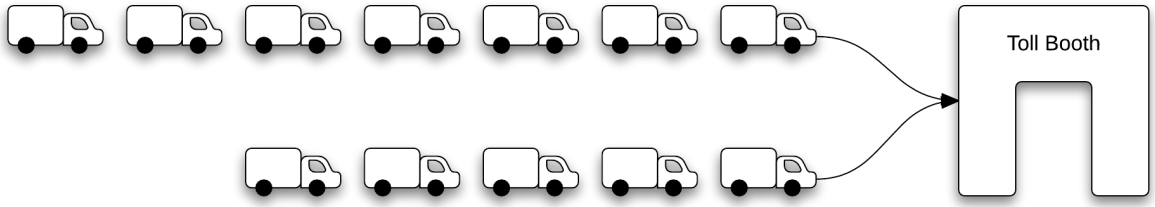
Remoting

Distributed by Default Mentality

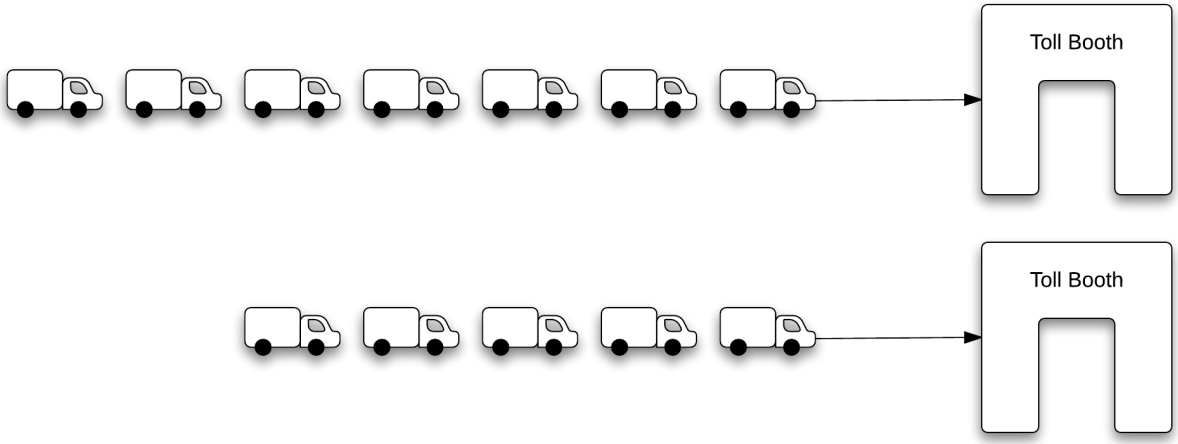
- Provides a **unified programming model**
- Employs referential or location transparency.
- Local and remote use the **same API**
- Distinguished by configuration, allowing a
- Succinct way to code Message-driven applications.

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 Domain Driven Design!
- System where Entities consume more resources than a single machine can provide.

Akka Persistence

Persistent State Management

- Staple for most applications.
- Provides not only this, but also automatic recovery
- When a system restarts due to failure or migration.
- Provides a foundation for building CQRS and Event Sourced 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 ::
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.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
public class StudentActor extends UntypedActor {
    public static Props props(final ActorRef teacher) {
        return Props.create(Student.class, () -> new Student(teacher));
    }
    public final ActorRef teacher;
    public final LoggingAdapter log =
        Logging.getLogger(context().system, getClass.getName());
    teacher.tell(new Question("What is the square root of pie?"), self());
    @Override
    public void onRecieve(final Object message) {
        log.info(message);
    }
}
```

The Teacher Actor

```
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
public class TeacherActor extends UntypedActor {
    public final LoggingAdapter log =
        Logging.getLogger(context().system, getClass.getName());
    @Override
    public void onRecieve(final Object message) {
        log.info(message);
        sender().tell(getAnswer(message), self());
    }
    private Answer getAnswer(final Object message) {
        // . . . some processing logic to derive answer, should handle errors
    }
}
```

Creating the Actors

```
public class School {

    . . . 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), “teacher”);

    }

}
```

Outline

1. Distributed Now?
2. The Challenges
3. Akka :: Java
4. Conclusion

Questions?

Akka for Java Devs

Bridging the Imagination Divide

Duncan K. DeVore
Typesafe
@ironfish