



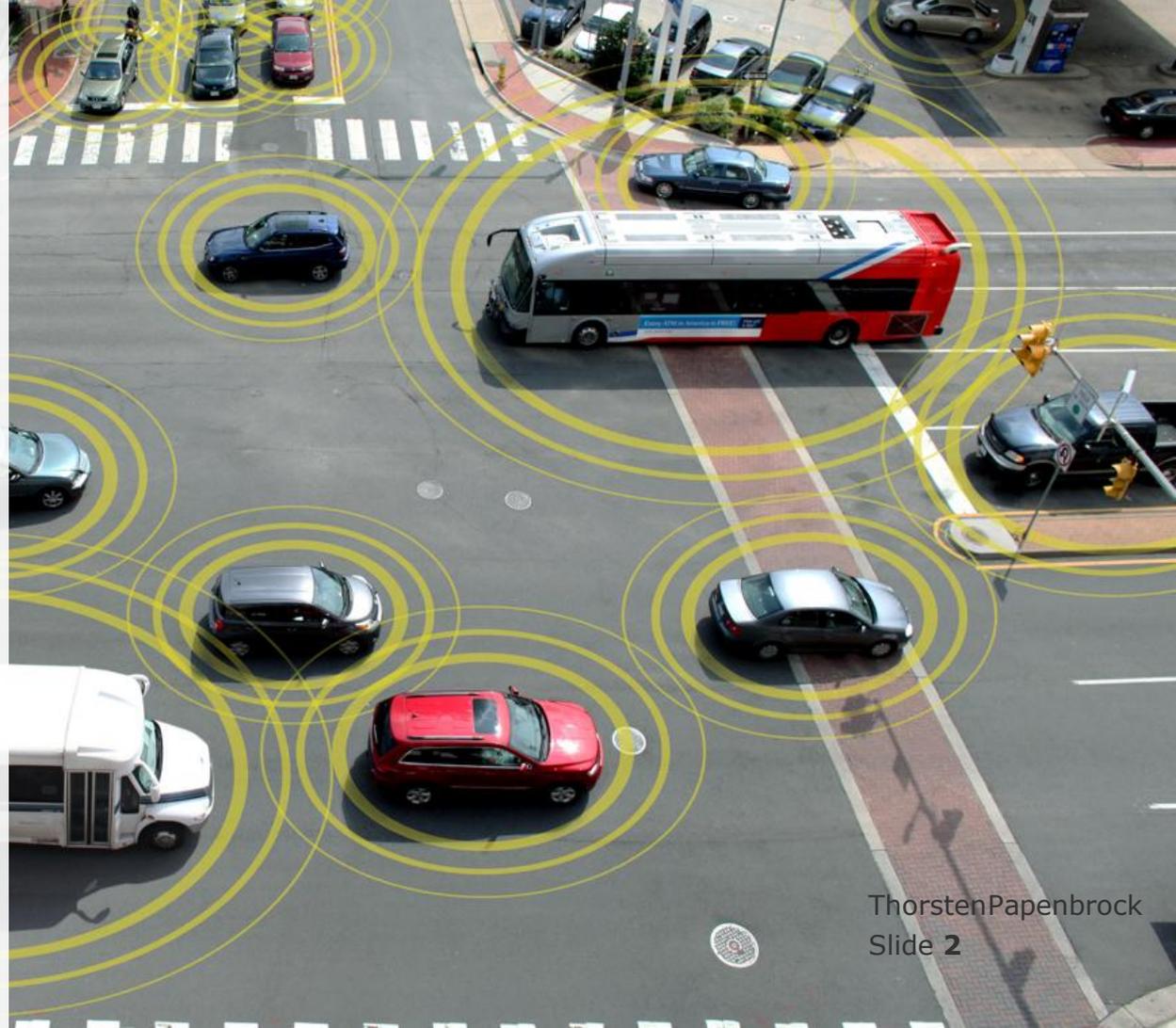
Distributed Data Management
Akka Actor Programming

Thorsten Papenbrock

F-2.04, Campus II
Hasso Plattner Institut

Akka Actor Programming Hands-on

- **Actor Model (Recap)**
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



Actor Programming

Object-oriented programming

- Objects encapsulate state and behavior.
- Objects communicate with each other.
- Separation of concerns makes applications easier to build and maintain.

Actor programming

- Actors encapsulate state and behavior.
- Actors communicate with each other.
- Actor activities are scheduled and executed transparently.
- Combines the advantages of object- and task-oriented programming.

Task-oriented programming

- Application split down into task graph.
- Tasks are scheduled and executed transparently.
- Decoupling of tasks and resources allows for asynchronous and parallel programming.

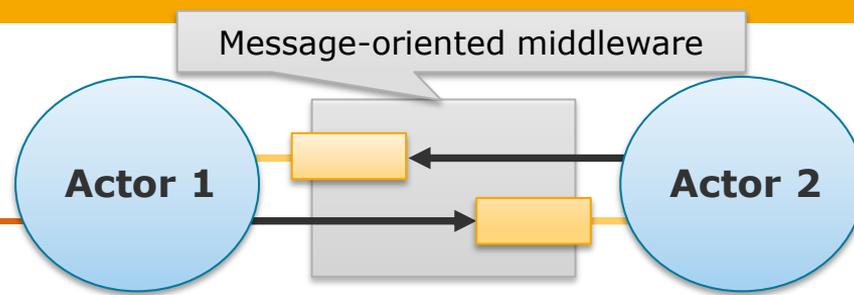
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 3

Actor Model (Recap)

Actor Model



Actor Model

- A stricter message-passing model that treats actors as the universal primitives of concurrent computation.
- **Actor:**
 - Computational entity (private state/behavior)
 - Owns exactly one mailbox (cannot subscribe to more or less queues)
 - Reacts on messages it receives (one message at a time)
- **Actor reactions:**
 - Send a finite number of messages to other actors
 - Create a finite number of new actors
 - Change own state, i.e., behavior for next message
- Actor model helps to prevent many parallel programming issues (concurrent memory access, race conditions, locking, deadlocks, ...)

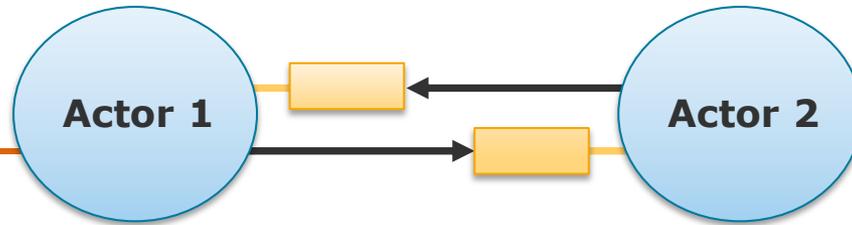


“The actor model retained more of what I thought were good features of the object idea”

Alan Kay, pioneer of object orientation

Actor Model (Recap)

Actor Model

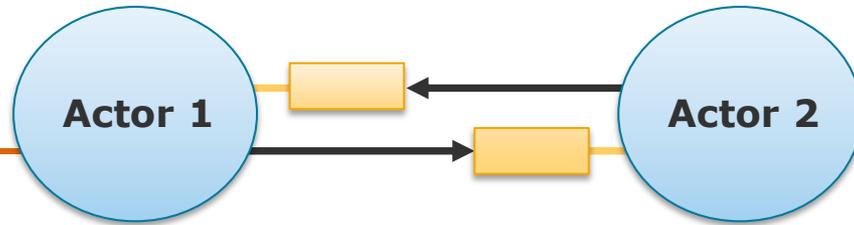


Advantages over RPC

- **Fault-tolerance:**
 - “Let it crash!” philosophy to heal from unexpected errors
 - Lifecycle management with automatic restarts of failed actors
 - Dead-letters mailbox to capture messaging faults
 - Errors are expected to happen and implemented into the model.
- **Fault-prevention:**
 - Asynchronous messaging helps to prevent deadlocks.
 - Private states and immutability prevent concurrent state modifications.
- **Parallelization:**
 - Implicit parallelization: Actors process one message at a time but different actors always operate independently, i.e., in parallel.
 - Dynamic parallelization: Actors may spawn new actors if needed.

Actor Model (Recap)

Actor Model



▪ Erlang:



- Actor library natively included in the Erlang language
- First popular actor implementation
- Special: **Native language support** and strong actor isolation

▪ Akka:



- Actor library for the JVM (Java und Scala)
- Most popular actor implementation (a.t.m.)
- Special: **Actor hierarchies** and typed actors

▪ Orleans:



- Actor library for Microsoft .NET/C#
- Very popular in research and industry (due to Microsoft)
- Special: **Virtual actors** (persistent state and transparent location)





- Implements the **actor model**
- A free and open-source **toolkit and runtime** for building concurrent and distributed applications on the JVM (<https://akka.io/>)
- Written in Scala (<https://scala-lang.org/>)
- Offers APIs for **Java und Scala**

- Invented by Jonas Bonér
- Maintained by Lightbend (<https://lightbend.com/>)

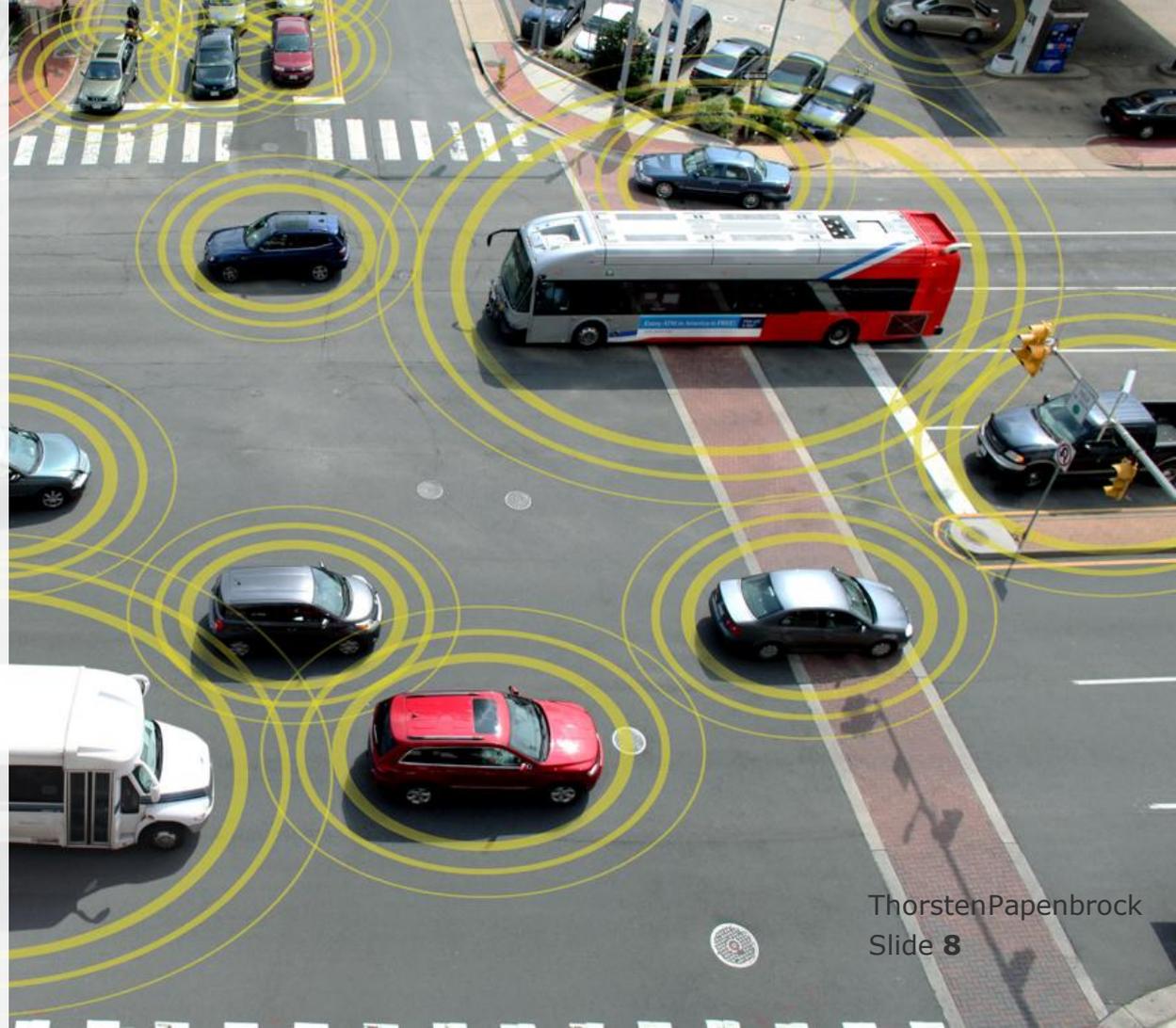
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 7

Akka Actor Programming Hands-on

- Actor Model (Recap)
- **Basic Concepts**
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



<https://akka.io/docs/>

Akka Documentation

Version 2.6.13

Java

Search

- Security Announcements
- Getting Started Guide
- General Concepts
- Actors
- Cluster
- Persistence
- Streams
- Discovery
- Utilities
- Other Akka modules
- Package, Deploy and Run
- Project Information
- Akka Classic**
 - Classic Actors
 - Classic Clustering
 - Classic Networking
 - Classic Utilities

Akka Classic

We will focus on **Java** in this Hands-on

Akka Classic pertains to the original Actor APIs, which have been improved by more type safe and guided Actor APIs. Akka Classic is still fully supported and existing applications can continue to use the classic APIs. It is also possible to use the new Actor APIs together with classic actors in the same ActorSystem, see [coexistence](#). For new projects we recommend using [the new Actor API](#).

Classic Actors

- Dependency
- Classic Actors**
- Classic Supervision
- Classic Fault Tolerance
- Classic Dispatchers
- Classic Mailboxes

Classic Clustering

- Classic Cluster Usage
- Classic Cluster Aware Routers
- Classic Cluster Singleton
- Classic Distributed Publish Subscribe in Cluster
- Classic Cluster Client

Classic Networking

- I/O
- Using TCP
- Using UDP
- DNS Extension

We will focus on **Akka Classic** in this Hands-on

- Classic Persistence
- Classic Persistent FSM
- Testing Classic Actors

- Classic Distributed Data
- Classic Multi-DC Cluster
- Classic Serialization

Basic Concepts

Akka Modules

Akka Actors

Core actor
model classes
for concurrency
and distribution

Akka Cluster

Classes for the
resilient and
elastic
distribution over
multiple nodes

Akka Streams

Asynchronous,
non-blocking,
backpressured,
reactive stream
classes

Akka Http

Asynchronous,
streaming-first
HTTP server and
client classes

Cluster Sharding

Classes to
decouple actors
from their
locations
referencing
them by identity

Akka Persistence

Classes to
persist actor
state for fault
tolerance and
state restore
after restarts

Distributed Data

Classes for an
eventually
consistent,
distributed,
replicated key-
value store

Alpakka

Stream
connector
classes to other
technologies

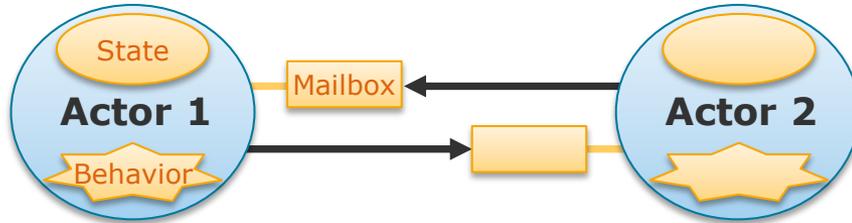
Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide 10

Basic Concepts

Akka Actors



- Actor = State + Behavior + Mailbox
- Communication:
 - Sending messages to mailboxes
 - Non-blocking, fire-and-forget
- Messages:
 - Immutable, serializable objects
 - Object classes are known to both sender and receiver
 - Receiver interprets a message via pattern matching

Mutable messages are possible,
but don't use them!

Distributed Data Management

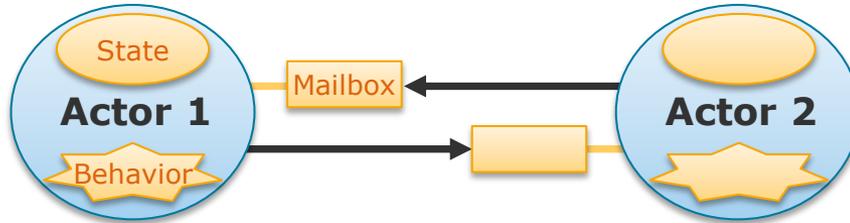
Akka Actor Programming

ThorstenPapenbrock
Slide 11

Basic Concepts

Akka Actors

Called in default actor constructor and set as the actor's behavior



```
public class Word extends AbstractActor {  
    @Override  
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(String.class, this::respondTo)  
            .matchAny(object -> System.out.println("Could not understand received message"))  
            .build();  
    }  
    private void respondTo(String message) {  
        System.out.println(message);  
        this.sender().tell("Received your message, thank you!", this.self());  
    }  
}
```

Inherit default actor behavior, state and mailbox implementation

The **Receive** class performs pattern matching and de-serialization

A **builder pattern** for constructing a **Receive** object with otherwise many constructor arguments

Send a response to the sender of the last message (asynchronously, non-blocking)

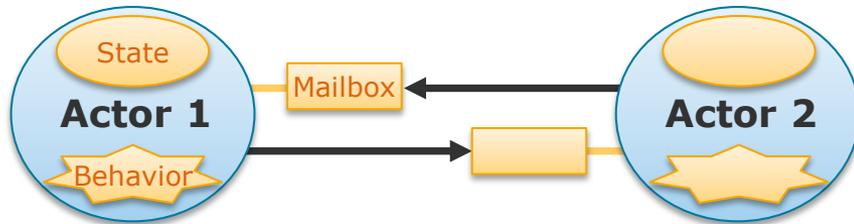
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 12

Basic Concepts

Akka Actors



```
public class Worker extends AbstractActor {  
  @Override  
  public Receive createReceive() {  
    return receiveBuilder()  
      .match(String.class, s -> this.sender().tell("Hello!", this.self()))  
      .match(Integer.class, i -> this.sender().tell(i * i, this.self()))  
      .match(Double.class, d -> this.sender().tell(d > 0 ? d : 0, this.self()))  
      .match(MyMessage.class, s -> this.sender().tell(new YourMessage(), this.self()))  
      .matchAny(object -> System.out.println("Could not understand received message"))  
      .build();  
  }  
}
```

The message types (= classes) define how the actor reacts

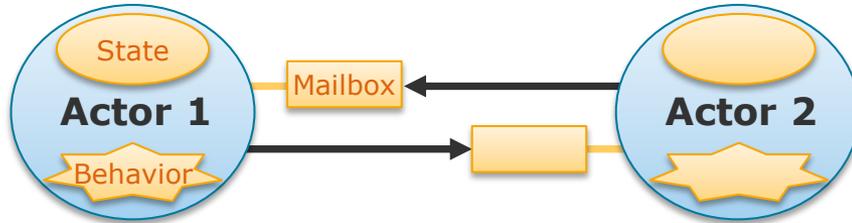
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 13

Basic Concepts

Akka Actors



```
public class Worker extends AbstractLoggingActor {
  @Override
  public Receive createReceive() {
    return receiveBuilder()
      .match(String.class, s -> this.sender().tell("Hello!", this.self()))
      .match(Integer.class, i -> this.sender().tell(i * i, this.self()))
      .match(Double.class, d -> this.sender().tell(d > 0 ? d : 0, this.self()))
      .match(MyMessage.class, s -> this.sender().tell(new YourMessage(), this.self()))
      .matchAny(object -> this.log().error("Could not understand received message"))
      .build();
  }
}
```

AbstractLoggingActor
provides proper logging

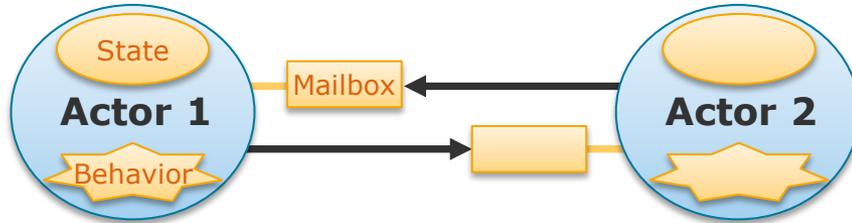
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 14

Basic Concepts

Akka Actors



```
public class Worker extends AbstractLoggingActor {
```

```
    public static class MyMessage implements Serializable {}
```

```
    @Override
```

```
    public Receive createReceive() {
```

```
        return receiveBuilder()
```

```
            .match(MyMessage.class, s -> this.sender().tell(new OtherActor.YourMessage(), this.self()))
```

```
            .matchAny(object -> this.log().error("Could not understand received message"))
```

```
            .build();
```

```
    }
```

```
}
```

Good practice:
Actors define their messages
(provides kind of an interface description)

**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide 15

Basic Concepts

Some Further Nodes

Redundant API calls

- Due to Java-Scala interface mix
 - `this.getContext() = this.context()`
 - `this.getSender() = this.sender()`
 - ...

Non-blocking, asynchronous

- Tell messaging
 - Java: `someActor.tell(message)`
 - Scala: `someActor ! message`

Blocking, synchronous

- Ask pattern
 - Java: `someActor.ask(message)`
 - Scala: `someActor ? message`

More on this pattern later!

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 16



Akka Scala Interface

A Master-Worker Example

```
case class Calculate(items: List[String])
case class Work(data: String)
case class Result(value: Int)
```

```
class Worker extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case Work(data) => sender ! Result(handle(data))
    case _ => log.info("received unknown message")
  }

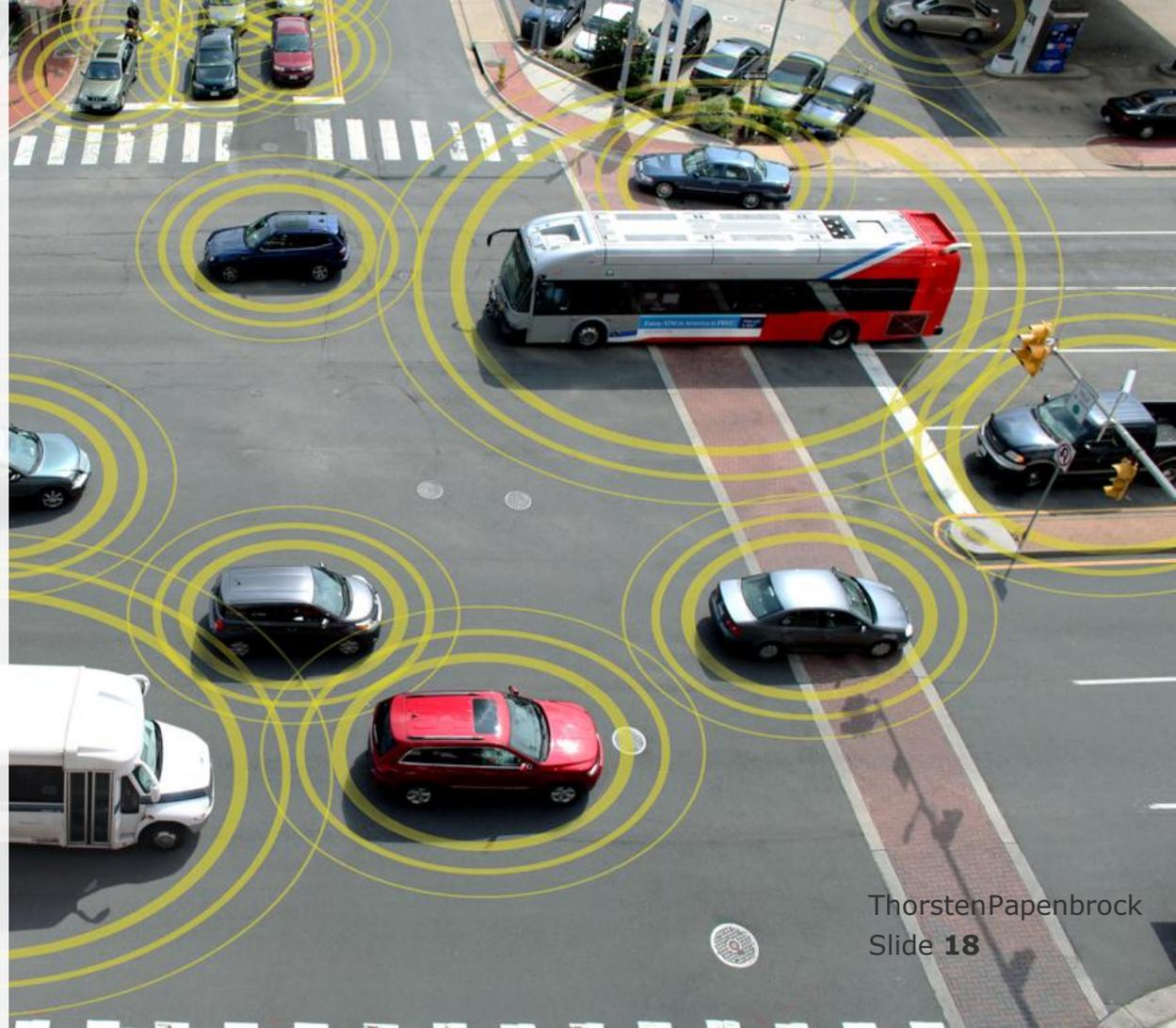
  def handle(data: String): Int = {
    data.hashCode
  }
}
```

```
class Master(numWorkers: Int) extends Actor {
  val worker = context.actorOf(Props[Worker], name = "worker")

  def receive = {
    case "Hello master" => sender ! "Hello sender"
    case Calculate(items) => for (i <- 0 until items.size) worker ! Work(item.get(i))
    case Result(value) => log.info(value)
    case _ => log.info("received unknown message")
  }
}
```

Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- **Runtime Architecture**
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



Task- and data-parallelism

Delegate work!

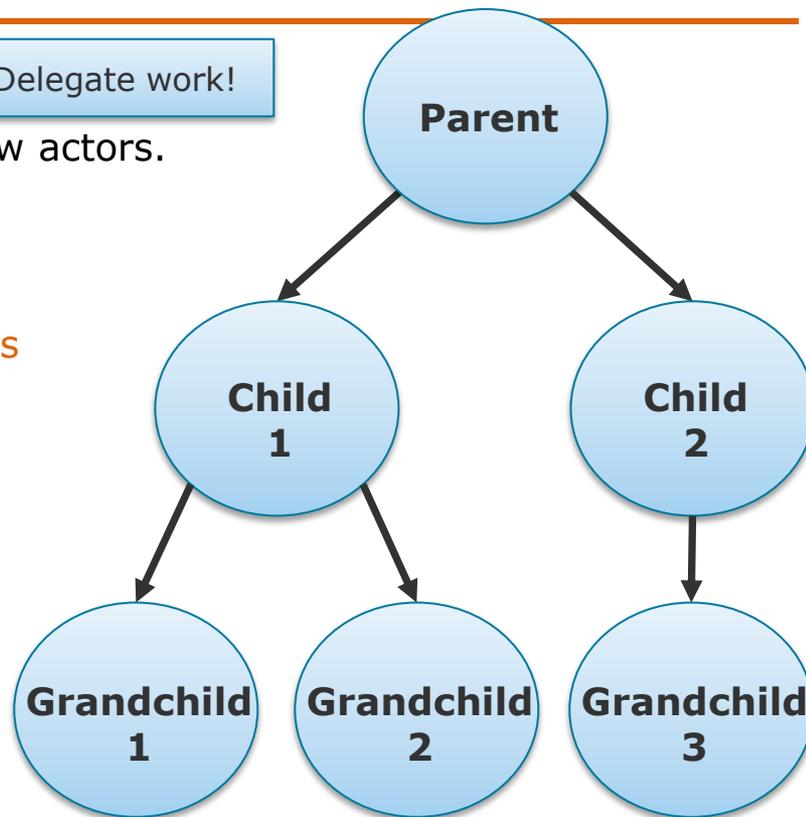
- Actors can dynamically create new actors.

Supervision hierarchy

- Creating actor (parent) **supervises** created actor (child).

Fault-tolerance

- If child fails, parent can choose:
 - restart**, **resume**, **stop**, or **escalate**



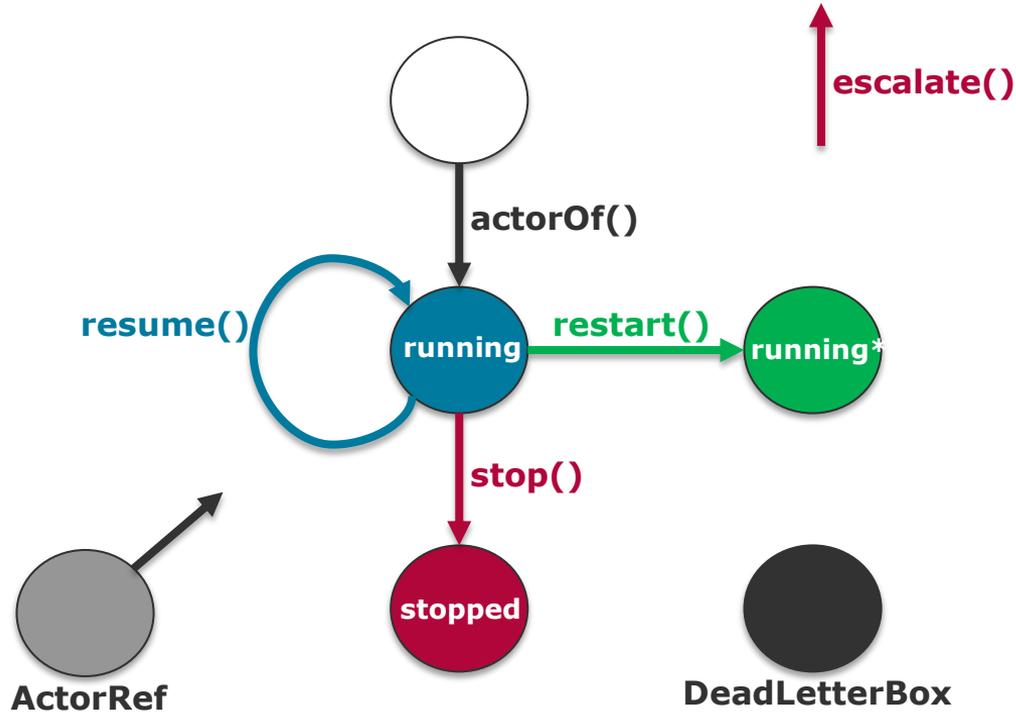
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 19

Runtime Architecture

Actor Lifecycles



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 20



- An **immutable and serializable handle** to an actor (kind of a symbolic link)
- **Decouples actors** from each other still allowing them to identify certain targets
- Can be **used to send message** to the referenced actor
- Referenced actor may reside in **the same or a remote ActorSystem**.

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **21**

Runtime Architecture

Actor Lifecycles

Actor Lifecycle

- **PreStart()**
 - Called before actor is started
 - Initialization
- **PreRestart()**
 - Called before actor is restarted
 - Free resources (keeping resources that can be re-used)
- **PostRestart()**
 - Called after actor is restarted
 - Re-initialization (re-using resources if possible)
- **PostStop()**
 - Called after actor was stopped
 - Free resources

Listen to
DisassociatedEvents

```
public class MyActor extends AbstractLoggingActor {  
  
    @Override  
    public void preStart() throws Exception {  
        super.preStart();  
        this.context().system().eventStream()  
            .subscribe(this.self(), DisassociatedEvent.class);  
    }  
  
    @Override  
    public void postStop() throws Exception {  
        super.postStop();  
        this.log().info("Stopped {}", this.self());  
    }  
}
```

Log that **MyActor**
was stopped

**Distributed Data
Management**

Akka Actor
Programming

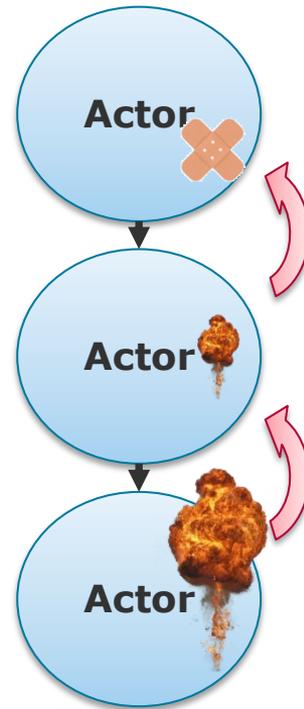
ThorstenPapenbrock
Slide **22**

Runtime Architecture

Let It Crash

“Let it crash” philosophy

- Distributed systems are inherently prone to errors (because there is simply more to go wrong/break).
 - Message loss, unreachable mailboxes, crashing actors ...
- Make sure that critical code is supervised by some entity that knows how errors can be handled.
- Then, if an error occurs, do not (desperately) try to fix it: let it crash!
 - Errors are propagated to supervisors that can deal better with them
- Example: Actor loses a database connection due to a DB restart.
 - It decides to crash.
 - Its supervisor restarts the actor, which re-creates the DB connection.



Runtime Architecture

Actor Hierarchies

```
public class Master extends AbstractLoggingActor {
    public Master() {
        ActorRef worker = this.context().actorOf(Worker.props());
        this.context().watch(worker);
    }
    @Override
    public SupervisorStrategy supervisorStrategy() {
        return new OneForOneStrategy(3,
            Duration.create(10, TimeUnit.SECONDS),
            DeciderBuilder.match(IOException.class, e -> restart())
                .matchAny(e -> escalate())
                .build());
    }
}
```

Receive **Terminated**-messages for watched actors

Try 3 restarts in 10 seconds for **IOExceptions**; otherwise escalate

```
public class Worker extends AbstractLoggingActor {
    public static Props props() {
        return Props.create(Worker.class);
    }
}
```

Create the **Props** telling the context how to instantiate you

Master

Worker

A
factory pattern

Distributed Data Management

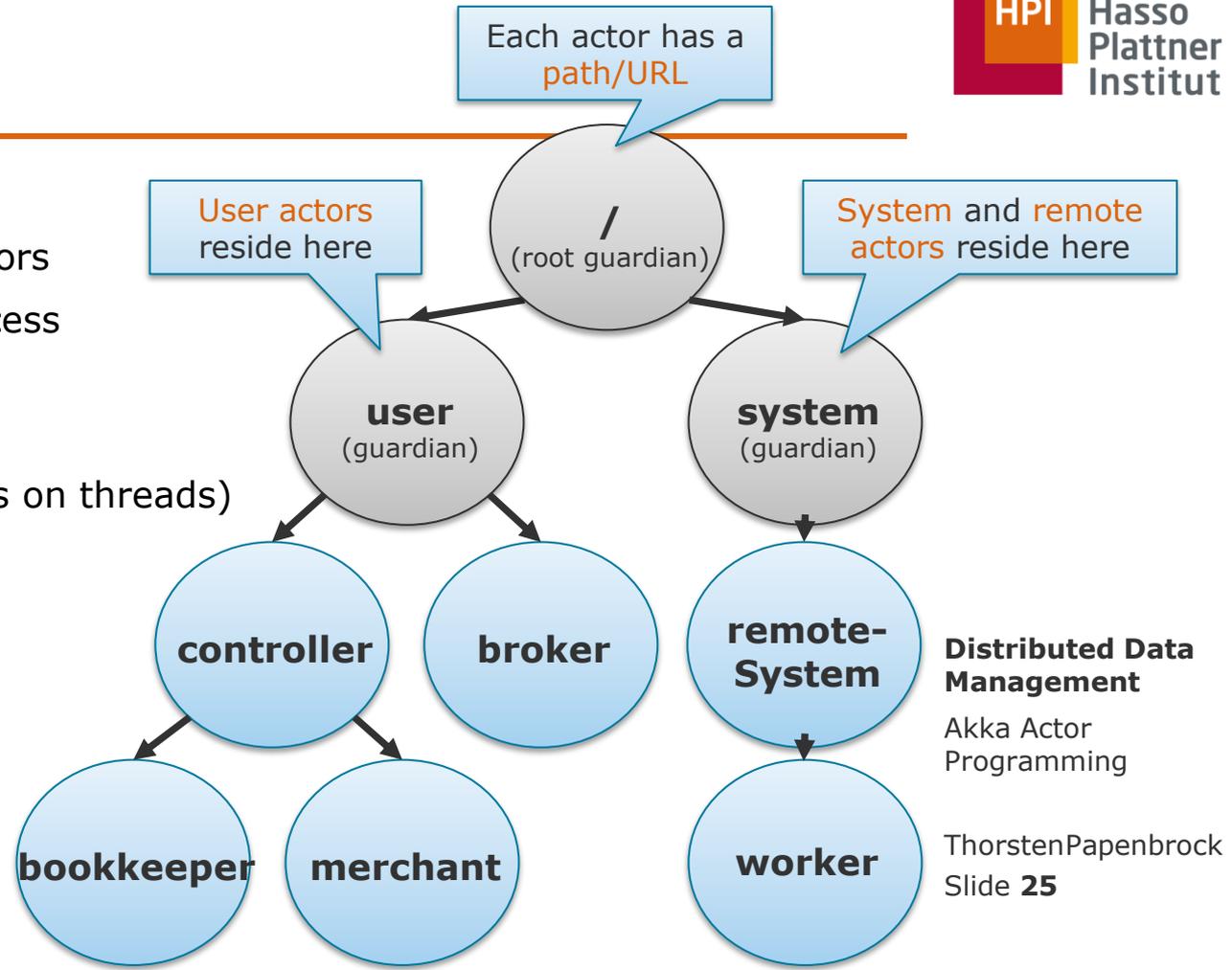
Akka Actor Programming

ThorstenPapenbrock
Slide 24

Runtime Architecture Actor Systems

ActorSystem

- A named hierarchy of actors
- Runs within one JVM process
- Configures:
 - **Actor dispatchers** (that schedule actors on threads)
 - **Global actor settings** (e.g. mailbox types)
 - **Remote actor access** (e.g. addresses)
 - ...



Runtime Architecture Actor Systems

Event stream

- Reacts on errors, new nodes, message sends, message loss, ...

Dispatcher

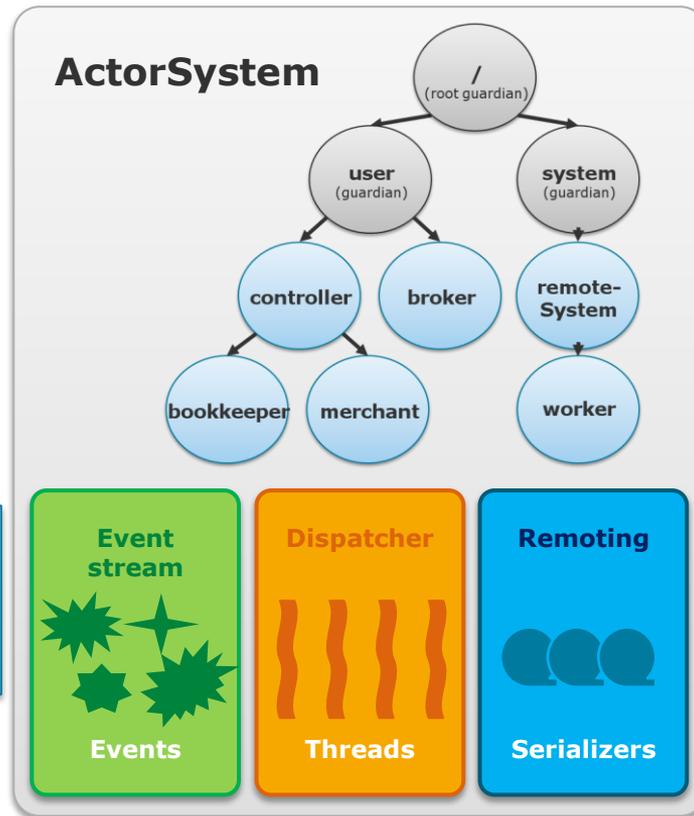
- Assigns threads dynamically to actors.
- Transparent multi-threading
 - # Threads \approx # CPU cores
 - # Actors $>$ # CPU cores (usually many hundreds)

- Over-provisioning!

Idle actors bind ~400 bytes of heap space but no CPU resources

Remoting

- Resolves remote actor addresses.
- Sends messages over network.
 - serialization + de-serialization



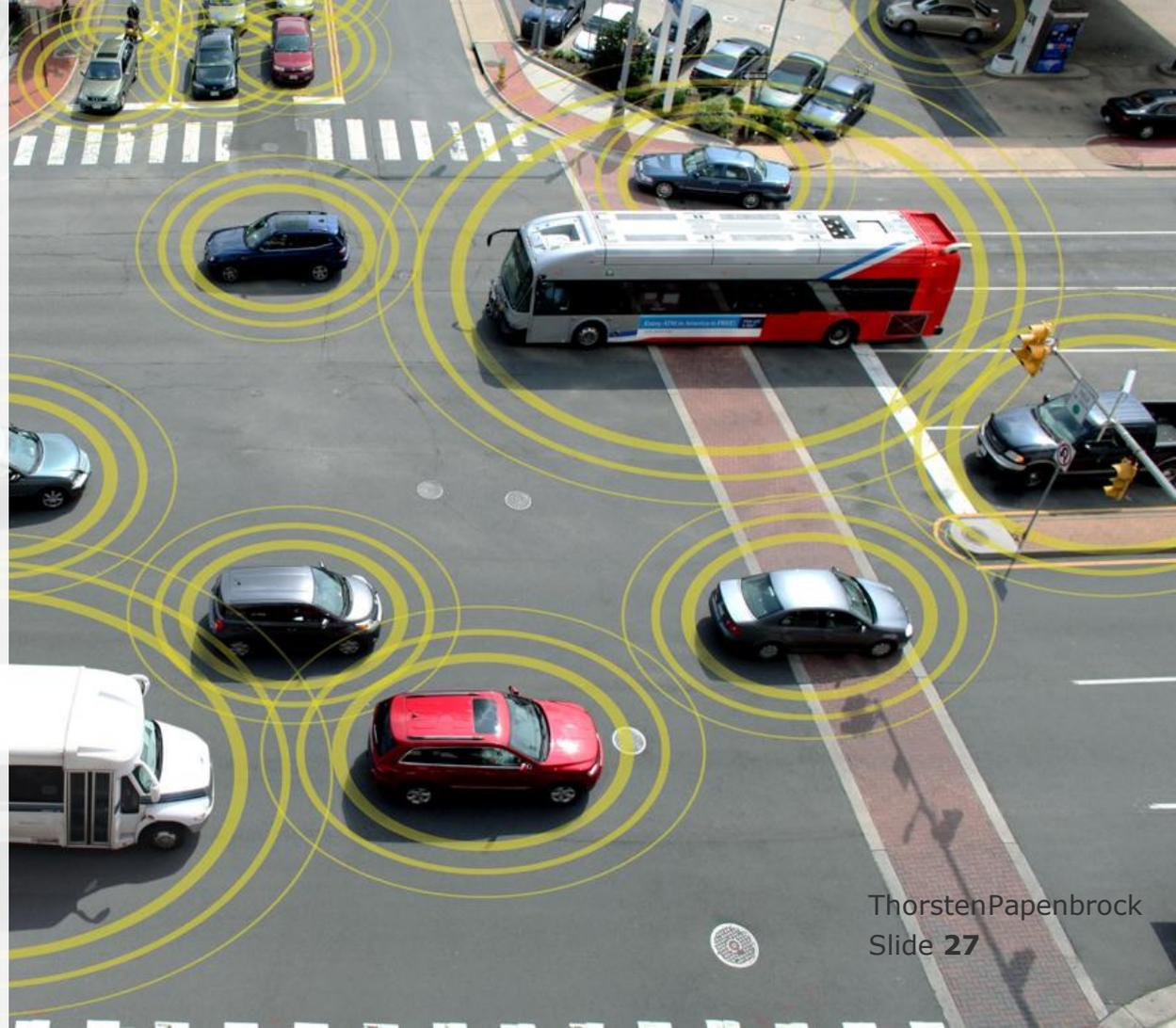
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 26

Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- **Demo**
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



<https://github.com/HPI-Information-Systems/akka-tutorial>

HPI-Information-Systems / akka-tutorial

Notifications Star 12 Fork 21

Code Issues Pull requests 3 Actions Projects Wiki Security Insights

master 4 branches 0 tags

Go to file Code

	thorsten-papenbrock Updated dependency versions for akka-tutorial and o... 904c8b1 3 minutes ago 62 commits
akka-tutorial	Updated dependency versions for akka-tutorial and octopus 3 minutes ago
ddm-exercise	Merge commit 12 minutes ago
octopus	Updated dependency versions for akka-tutorial and octopus 3 minutes ago
.gitignore	Merged the Imp and pc tasks into one project, updated the librari... 23 minutes ago
LICENSE	Added the octopus project to the repository. 3 years ago
README.md	Merged the Imp and pc tasks into one project, updated the librari... 23 minutes ago

README.md

Akka tutorial

About

Code for the Akka tutorial

Readme

Apache-2.0 License

Releases

No releases published

Distributed Data Management

Akka Actor Programming

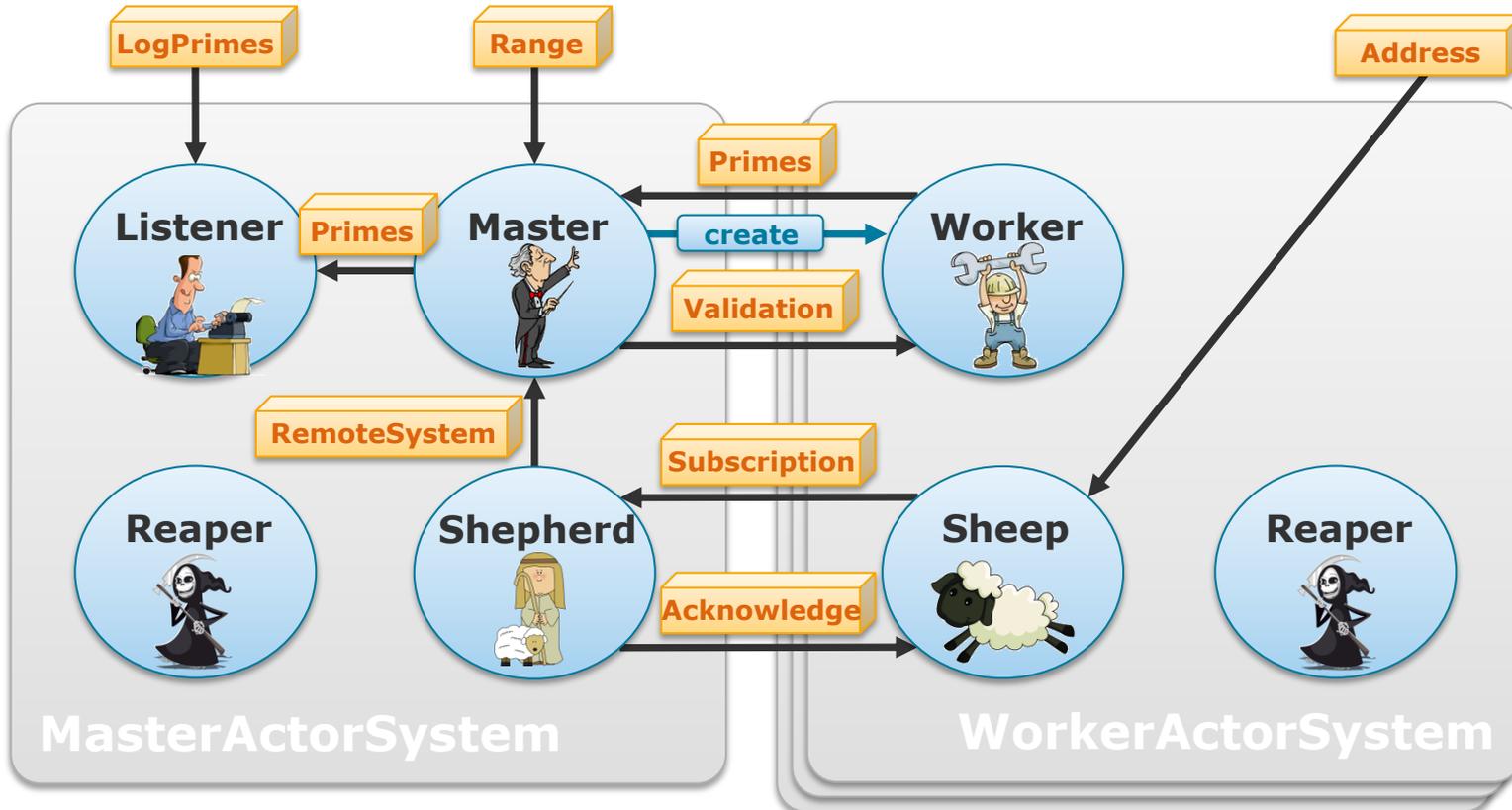
Packages

No packages published

ThorstenPapenbrock Slide 28

Contributors 6

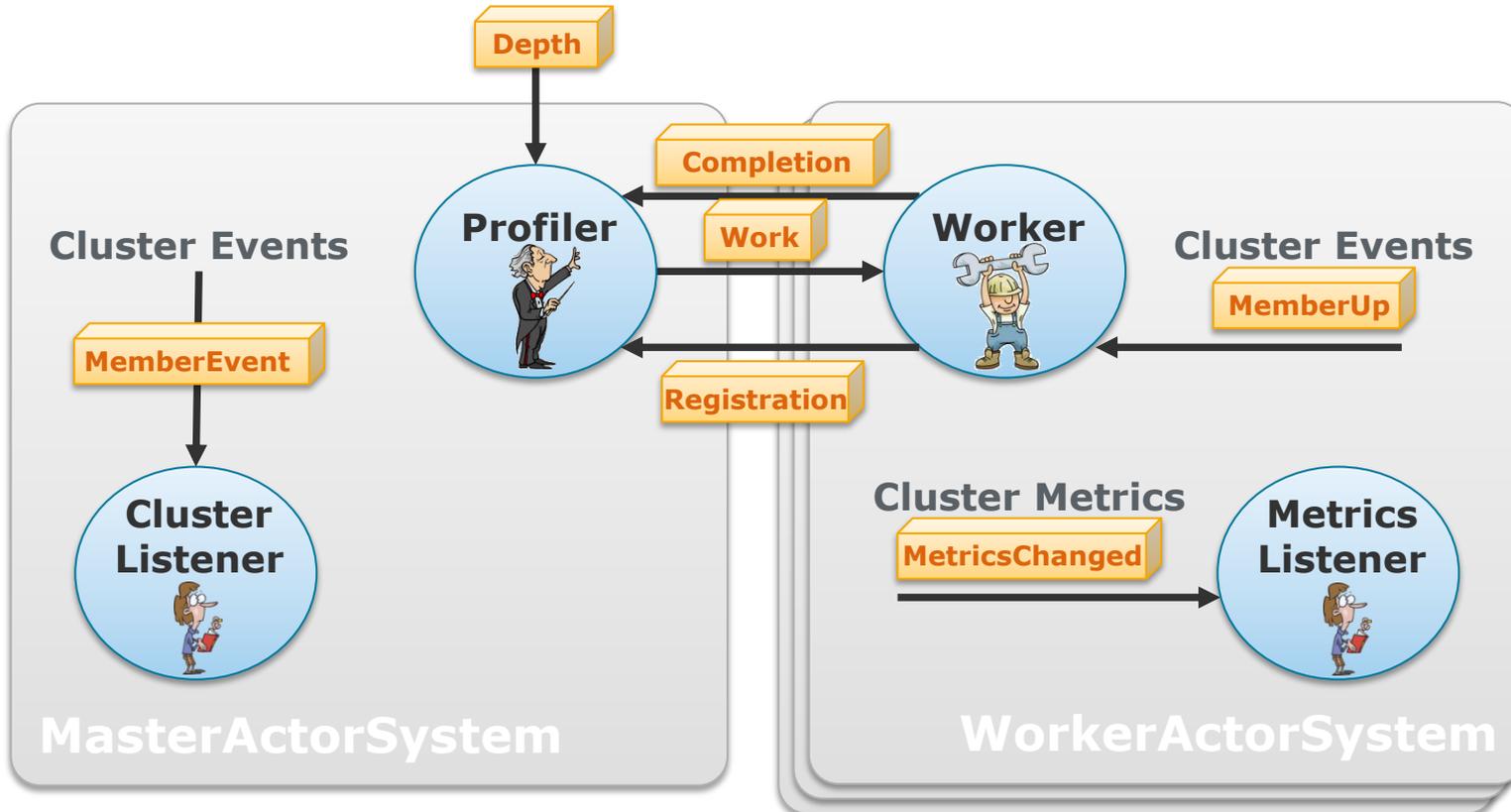
Demo akka-tutorial



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 29

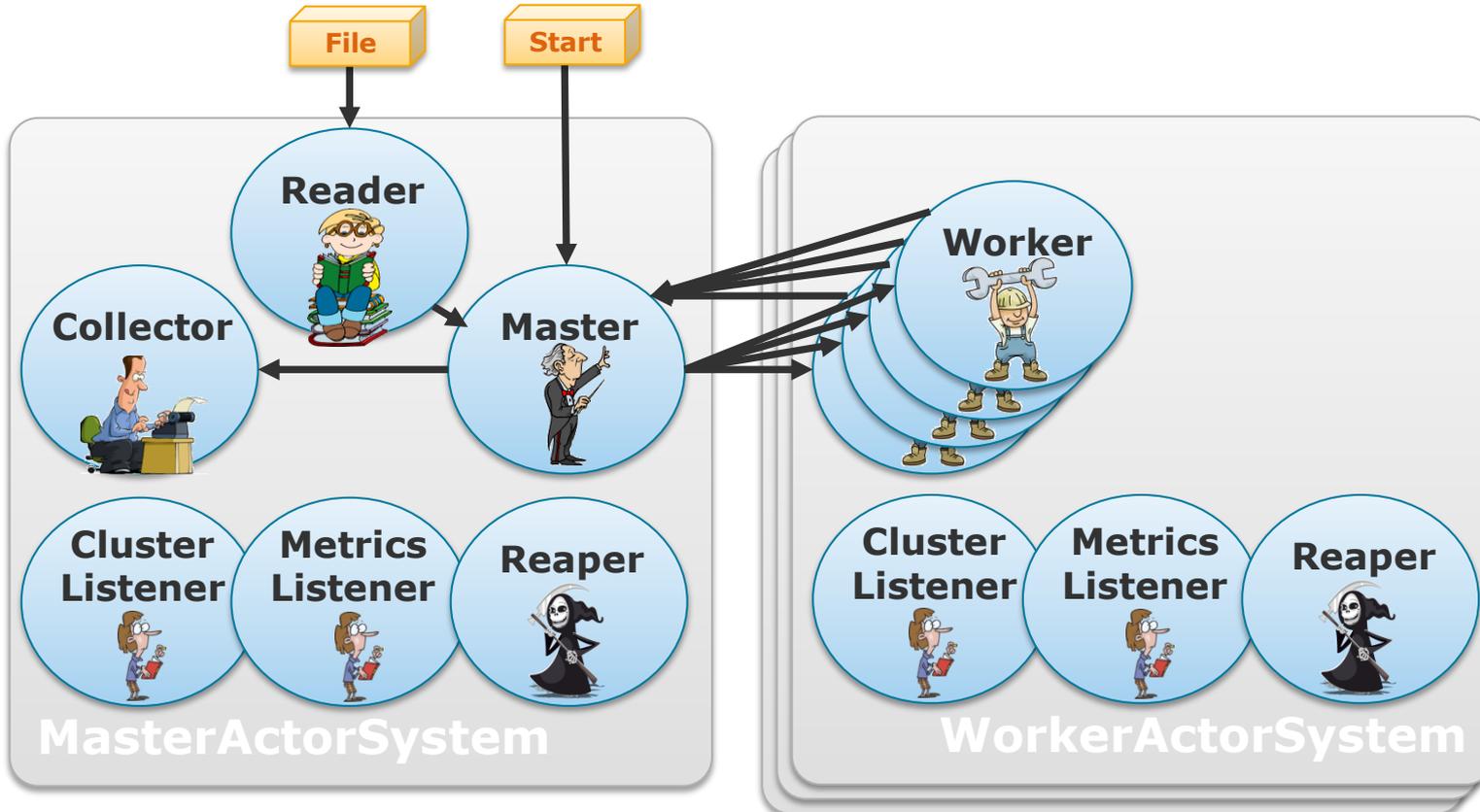


**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **30**

Demo ddm-exercise



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 31

Demo

Minimal Setup

pom.xml



Maven, sbt or Gradle

Base actor library

actors, supervision, scheduling, ...

Remoting library

remote actors, heartbeats ...

Logger library

logging event bus for akka actors

Testing library

TestKit class, expecting messages, ...

Kryo library

Custom serialization with Kryo

```
<dependencies>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-actor_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-slf4j_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-testkit_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.twitter</groupId>
    <artifactId>chill-akka_${scala.version}</artifactId>
    <version>0.9.2</version>
  </dependency>
</dependencies>
```

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 32

Demo

Minimal Setup

application.conf

remote.conf, akka.conf, ... however you call it

```
akka {  
  actor {  
    provider = remote  
  }  
  remote {  
    artery {  
      enabled = on  
      transport = tcp  
      canonical.hostname = "192.168.0.5"  
      canonical.port = 7787  
    }  
  }  
}
```

or udp

hostname or IP

... of this actor system

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 33

Demo

Minimal Setup

Main.java

AlgorithmXY.java, Homework1.java, ...

```
public static void start(String appName, String host, int port) {  
  
    final Config config = ConfigFactory.parseString(  
        "akka.remote.artery.canonical.hostname = \" + host + "\"\n" +  
        "akka.remote.artery.canonical.port = \" + port")  
        .withFallback(ConfigFactory.load("application"));  
  
    final ActorSystem system = ActorSystem.create(appName, config);  
  
    final ActorRef master = system.actorOf(Master.props(), Master.DEFAULT_NAME);  
    final ActorRef worker1 = system.actorOf(Worker.props(master), Worker.DEFAULT_NAME + "1");  
    final ActorRef worker2 = system.actorOf(Worker.props(master), Worker.DEFAULT_NAME + "2");  
    final ActorRef worker3 = system.actorOf(Worker.props(master), Worker.DEFAULT_NAME + "3");  
  
    master.tell("Hello World", ActorRef.noSender());  
}
```

Create the ActorSystem

Populate the ActorSystem

Cause some action

Ending the main thread does not end the ActorSystem!
We will discuss termination in a few slides.

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 34

```
public class WorkerTest {  
  
    private ActorSystem actorSystem;  
  
    @Before  
    public void setUp() {  
        this.actorSystem = ActorSystem.create();  
    }  
  
    @Test  
    public void shouldWorkAsExpected() {  
        new TestKit(this.actorSystem) {  
            ActorRef worker = this.actorSystem.actorOf(Worker.props());  
            worker.tell(new Worker.WorkMessage(73), this.getRef());  
  
            Master.ResultMessage expectedMsg = new Master.ResultMessage(42);  
            this.expectMsg(Duration.create(3, "secs"), expectedMsg);  
        }  
  
        @After  
        public void tearDown() {  
            TestKit.shutdownActorSystem(this.actorSystem);  
        }  
    }  
}
```

TestKit offers an ActorRef over which it can expect responses

Distributed Data Management

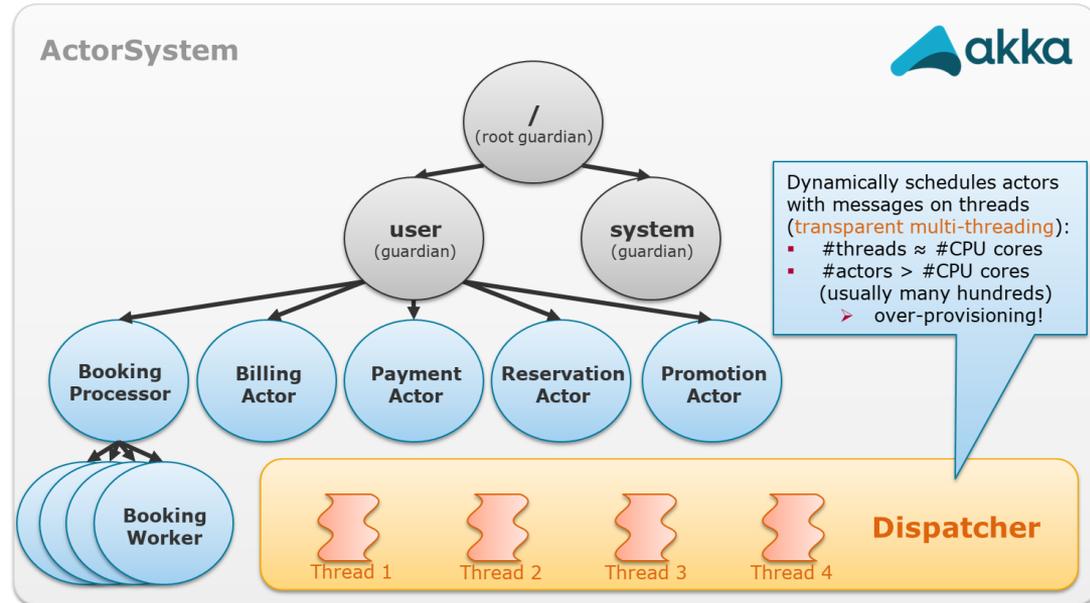
Akka Actor
Programming

ThorstenPapenbrock
Slide 35

Demo

Dispatching

- The ActorSystem schedules its actors on threads that live inside a dispatcher.
 - With no further care, all actors are scheduled on the same **default dispatcher**.
 - But we might want to ...
 - control actor prioritization.
 - configure dispatching logic.
 - avoid starvation.
 - separate worker actors from system actors.
 - pin actors to threads.
- Use custom dispatchers!



Demo

Dispatching

- Configure a custom dispatcher in the `application.conf`:

```
my-dispatcher {  
  type = Dispatcher  
  executor = "fork-join-executor"  
  fork-join-executor {  
    # Parallelism (threads) ... ceil(available processors * factor)  
    parallelism-factor = 2.0  
    # Min number of threads to cap factor-based parallelism number to  
    parallelism-min = 2  
    # Max number of threads to cap factor-based parallelism number to  
    parallelism-max = 10  
  }  
  # Throughput defines the maximum number of messages  
  # processed per actor before the thread jumps to the next actor  
  # Set to 1 for as fair as possible.  
  throughput = 100  
}
```

alternatives:

- Dispatcher
- PinnedDispatcher

alternatives:

- fork-join-executor
- thread-pool-executor

= number of threads per core

```
my-other-dispatcher {  
  type = Dispatcher  
  executor = "thread-pool-executor"  
  thread-pool-executor {  
    fixed-pool-size = 32  
  }  
  throughput = 1}
```

Demo

Dispatching

- Dispatch actors on your custom dispatcher:
 - Option A: via code

```
final ActorRef worker = system.actorOf(  
  Props.create(Worker.class).withDispatcher("my-dispatcher"),  
  "worker");
```

- Option B: via configuration

```
akka.actor.deployment {  
  /worker {  
    dispatcher = my-dispatcher  
  }  
}
```

```
final ActorRef worker = system.actorOf(  
  Props.create(Worker.class),  
  "worker");
```

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide 38

Create dedicated Dispatchers!

- A dedicated `PinnedDispatcher` with **one thread** may be an optimal choice for a central `Master/Leader/Scheduler/...` actor to optimize cache lines and availability for that special role.
- A `Dispatcher` on a `ThreadPool` with `#CPU_Cores - n` threads is a good execution model for `Worker` actors with long running jobs, because they will block at most `#CPU_Cores - n` cores and leave `n` cores for other system activities to keep the system reactive.
- Why use `Dispatchers` for parallelization control and not actor instances? (i.e. we could also limit the number of `Worker` actors to `#CPU_Cores - n`)
 - Answer: **Overprovisioning**
 - If we have more than one type of actor for long running jobs, e.g., `FileReadingWorkers` and `DatabaseReadingWorkers`, there should be in total only `#CPU_Cores - n` workers but we might want to max out either of the two types at times.

Demo

Best Practices

Not everything is an actor!

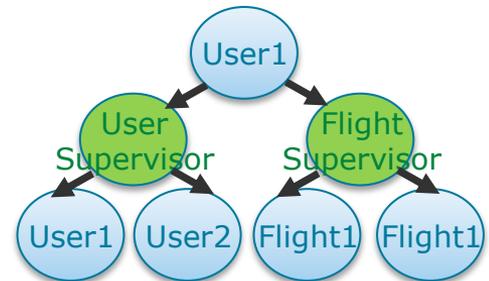
- Sending a message to an actor is more expensive than calling a method.
- Use actors where concurrency is expected.
- You can still use objects to encapsulate functionality and keep actor implementations clean.

Actors should do one thing!

- Although actors can dynamically change their behaviors and react to all kinds of messages, overly complex interfaces and unclear semantics make actors unmaintainable.
- Define a clear role and a semantically meaningful task for every actor.

Actor hierarchies rock!

- Flat hierarchies (e.g. putting everything below the user guardian) prohibit proper failure handling.
- Use supervisor actors for focused failure handling.



Demo

Best Practices

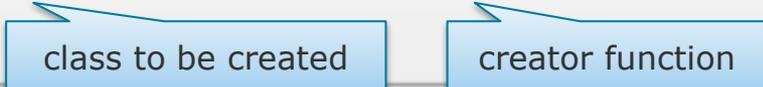
Keep messages organized!

- Actors do (in general) not make their accepted messages visible in interfaces.
- Good practice: Define messages inside the classes of actors that consume them.

Let actors decide about their instantiation!

- We create many actors, so concise instantiation code matters.
- Good practice: Define the creation of actor props inside their own classes.

```
// Somewhere:  
ActorRef worker = this.context().actorOf(Worker.props(resultCollector, filePath));  
  
// Worker:  
public abstract class Worker extends AbstractLoggingActor {  
  public static Props props(final ActorRef resultCollector, final String filePath) {  
    if (SystemUtils.hasGPU())  
      return Props.create(GPUWorker.class, () -> new GPUWorker(resultCollector, filePath));  
    return Props.create(CPUWorker.class, () -> new CPUWorker(resultCollector, filePath));  
  }  
}
```



class to be created

creator function

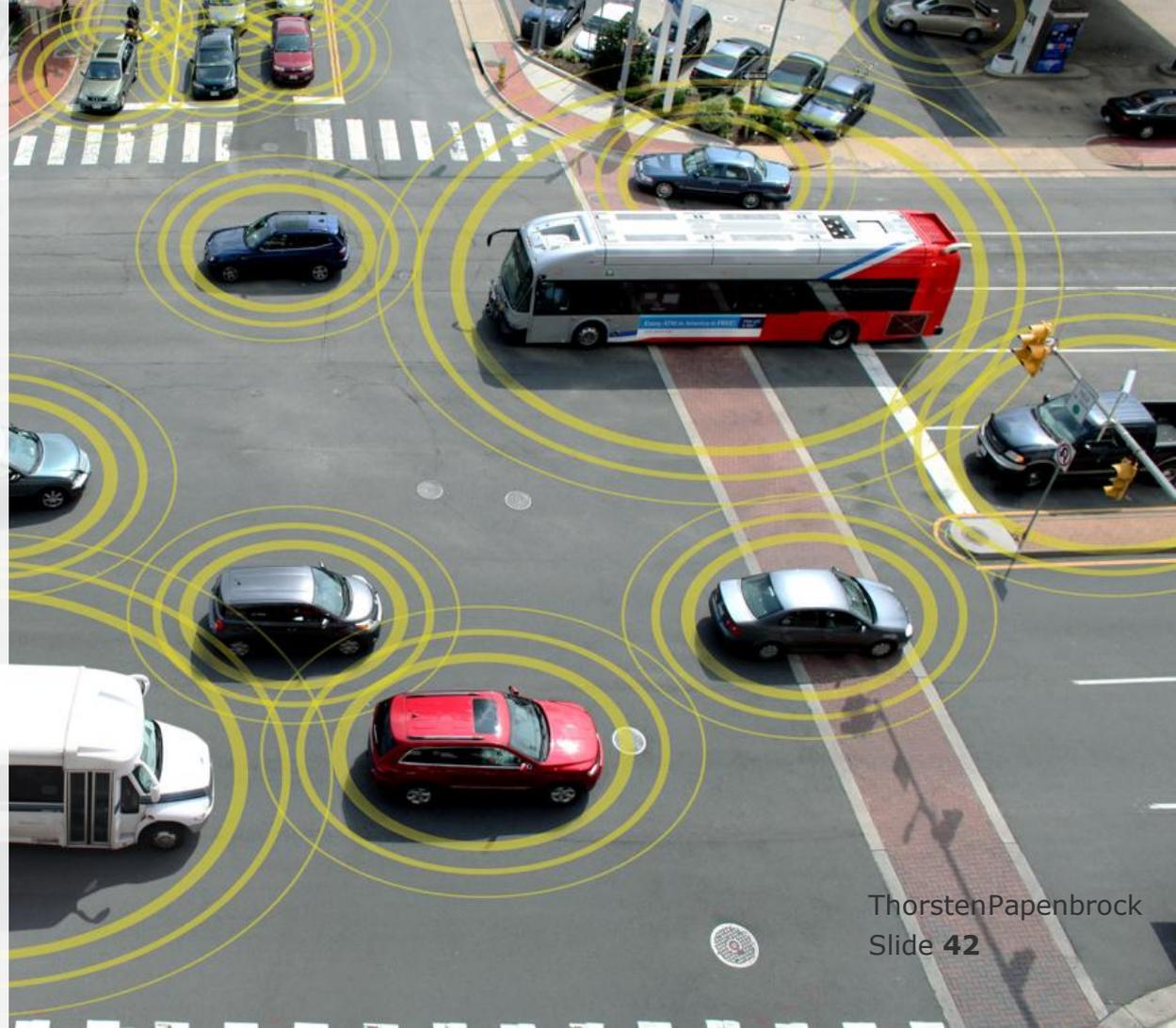
Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide 41

Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- **Messaging**
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



Messaging

Message Delivery Guarantees

Message delivery

- **at-most-once**: each message is delivered zero or one times.

- no guaranteed delivery; no message duplication
- highest performance; no implementation overhead
- fire-and-forget

You can implement at-least-once and exactly-once, with at-most-once!

- **at-least-once**: each message is delivered one or more times.

- guaranteed delivery; possibly message duplication
- ok-ish performance; state in sender
- send-and-acknowledge

With TCP Akka basically guarantees exactly-once, but node failures can still cause message loss!

- **exactly-once**: each message is delivered once.

- guaranteed delivery; no message duplication
- bad performance; state in sender and receiver
- send-and-acknowledge-and-deduplicate

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 43

Messaging

Message Delivery Guarantees

Message ordering

- **no ordering**: all messages can be arbitrarily out of order
 - no guaranteed ordering
 - highest performance; no implementation overhead
- **sender-receiver ordering**: all messages between specific sender-receiver pairs are ordered (by send order)
 - ordered individual communications
 - good performance; message broker simply sustains received order
- **total ordering**: all messages are ordered (by send timestamps)
 - serialized communication (see *total-order-broadcast* later in lecture)
 - bad performance; global ordering



Only with TCP!

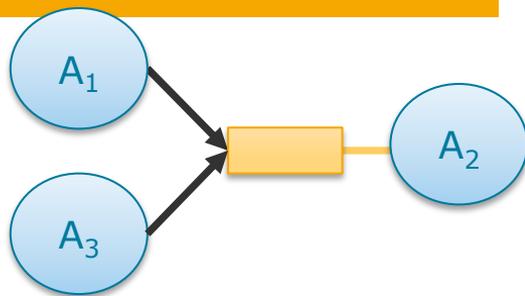
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 44

Messaging

Message Delivery Guarantees



Message ordering

- **sender-receiver ordering**: all messages between specific sender-receiver pairs are ordered (by send order)
- Example:
 - Actor A_1 sends messages M_1, M_2, M_3 to A_2
 - Actor A_3 sends messages M_4, M_5, M_6 to A_2

“If X is delivered...”
➤ No guaranteed delivery, i.e., messages may get lost and not arrive at A_2 !

- If M_1 is delivered it must be delivered before M_2 and M_3
- If M_2 is delivered it must be delivered before M_3
- If M_4 is delivered it must be delivered before M_5 and M_6
- If M_5 is delivered it must be delivered before M_6
- A_2 can see messages from A_1 interleaved with messages from A_3

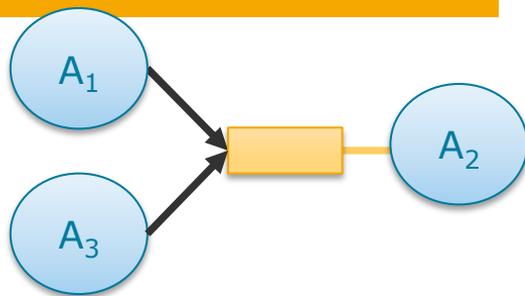
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 45

Messaging

Message Delivery Guarantees



Message ordering

- **sender-receiver ordering**: all messages between specific sender-receiver pairs are ordered (by send order)

- Send order is not transitive:

A_1 sends M_1 to A_2

A_1 sends M_2 to A_3

A_2 forwards M_1 to A_3

- A_3 may receive M_1 and M_2 in any order!

- Failure communication uses different channel:

A_1 has child A_2

A_2 sends M_1 to A_1

A_2 fails causing failure message M_2 being send to A_1

- A_1 may receive M_1 and M_2 in any order!

Framework does not know that M_1 was forwarded, i.e., that M_1 is "younger" than M_2



In this way, A_1 might receive a message from a dead actor A_2

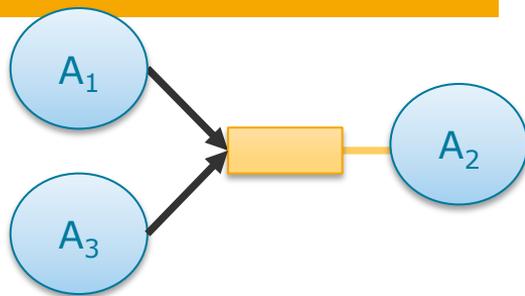
Although A_2 causes M_2 , it is technically not the sender of M_2

Akka Actor Programming

ThorstenPapenbrock
Slide 46

Messaging

Message Delivery Guarantees



Message ordering

- **sender-receiver ordering**: all messages between specific sender-receiver pairs are ordered (by send order)
- General notes:
 - Ordering guarantee holds only for TCP-based messaging.
 - The ordering guarantee can be violated by various factors, such as node failures.
 - If ordering is important, add and check custom sequence numbers!

Distributed Data Management

Akka Actor Programming

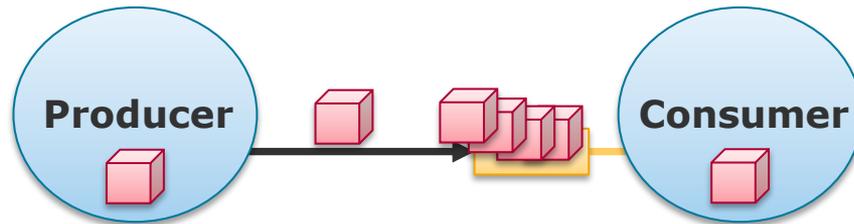
ThorstenPapenbrock
Slide 47

Messaging

Pull vs. Push

Work Propagation

- **Producer** actors generate work for other **consumer** actors
- **Push propagation:**
 - Producers send work packages to their consumers immediately (in particular, data is copied over the network proactively).
 - Work is queued in the inboxes of the consumers.
 - **Fast work propagation; risk for message congestion/drops**



You can have **back-pressured mail boxes**, but that kind of kills the non-blocking, fire-and-forget messaging

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **48**

Messaging

Pull vs. Push

Work Propagation

- **Producer** actors generate work for other **consumer** actors
- **Push propagation:**
 - Producers send work packages to their consumers immediately (in particular, data is copied over the network proactively).
 - Work is queued in the inboxes of the consumers.
 - **Fast work propagation; risk for message congestion/drops**
- **Pull propagation:**
 - Consumers ask producers for more work if they are ready.
 - Work is queued in the producers' states.
 - **Slower work propagation; no risk for message congestion**

```
public class PullProducer extends AbstractLoggingActor {
    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(NextMessage.class, this.sender().tell((this.workPackages.remove())))
            .matchAny(object -> this.log().info("Unknown message"))
            .build();
    }
}
```

Distributed Data Management

Akka Actor
Programming

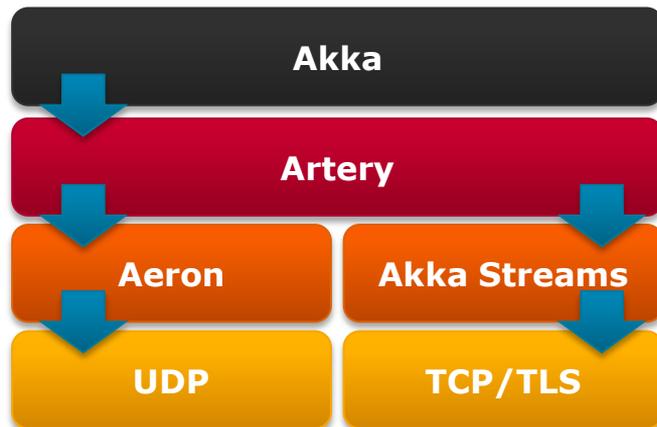
ThorstenPapenbrock
Slide 49

Messaging

Akka's Messaging System

Artery

- High-performance, streaming-based messaging system
- Part of the Akka toolkit
- Uses compression on actor path strings (in ActorRefs) to reduce the general message overhead
- Based on Aeron for UDP channels and Akka Streams for TCP/TLS channels



Distributed Data Management

Akka Actor Programming

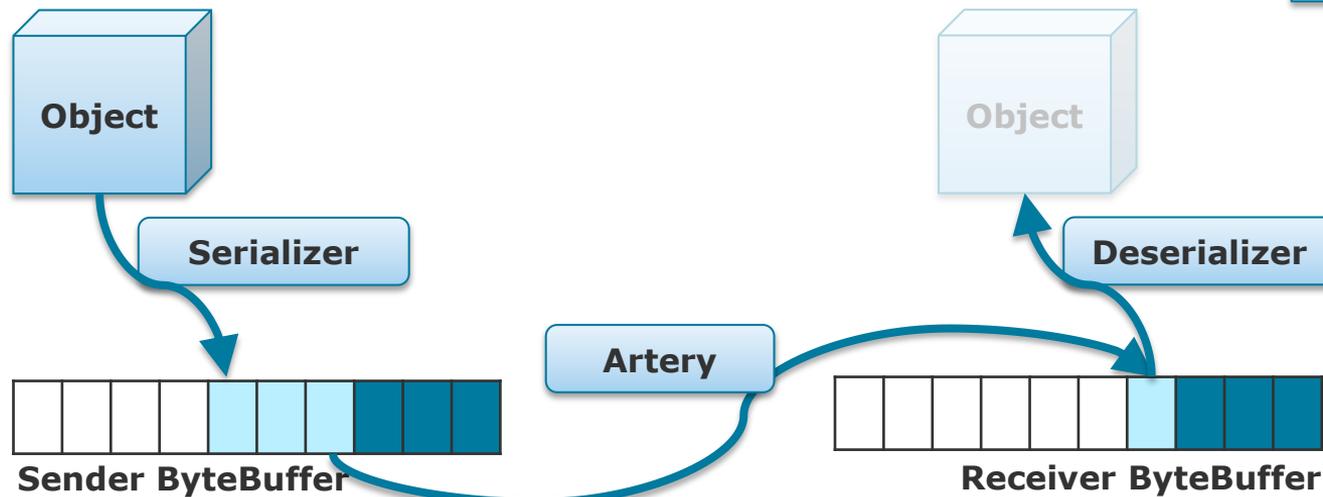
ThorstenPapenbrock
Slide 50

Messaging Akka's Messaging System

Artery

- Focused on high-throughput, low-latency communication
- Mostly allocation-free operations (serialize data directly into buffers)
- Support for faster serialization/deserialization using ByteBuffers

Streaming!



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 51

Messaging

Akka's Messaging System

Artery

- Focused on high-throughput, low-latency communication
- Mostly allocation-free operations (serialize data directly into buffers)
- Support for faster serialization/deserialization using ByteBuffers

➤ But: What if we need to send large amounts of data **over the network**?

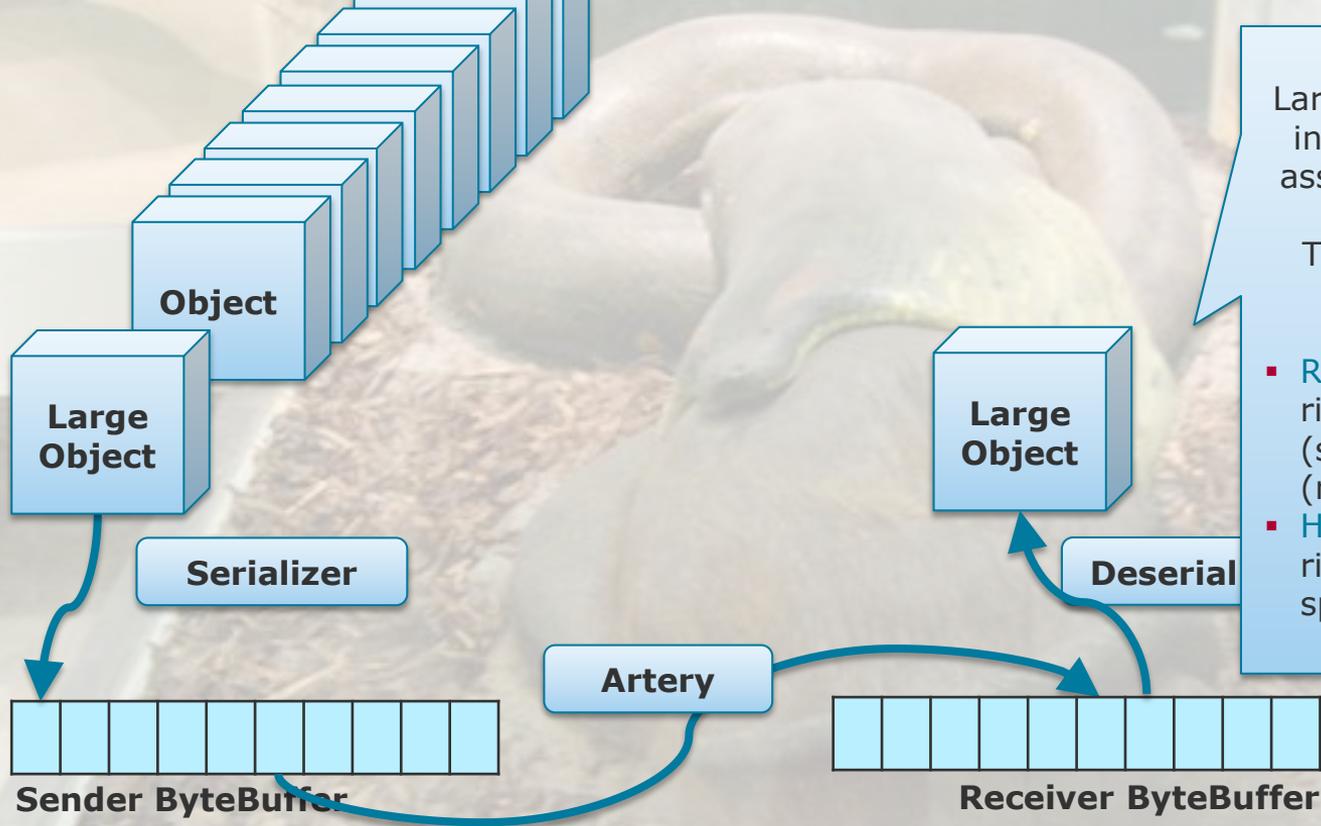
Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **52**



➤ But: What if we need to send large amounts of data **over the network**?



Large messages are broken down into frames that need to be re-assembled on the receiving side.

This blocks the TCP socket for other messages:

- **Regular messages:**
risk of message congestion (sender) and idle times (receiver)
- **Heartbeat messages:**
risk of cluster partitions and split-brain scenarios

➤ But: What if we need to send large amounts of data **over the network**?

- Use side channels for large data transfer
 - Different channel that does not block main channel messages
 - Transfer protocol that is optimized for large files (WebSockets, UDP, FTP, ...)
 - Side channel examples:
 - Artery's Large Message Channel
 - Akka's http client-server module
 - Netty, FTP or other file transfer protocols
 - Database or shared file system
1. Send data via side channel to memory/disk of remote host.
 2. Send data reference in an Akka message when data is transferred.

Messaging

Artery's Large Message Channel

application.conf

```
akka {  
  actor {  
    provider = remote  
  }  
  remote {  
    artery {  
      enabled = on  
      transport = tcp  
      canonical.hostname = "192.168.0.5"  
      canonical.port = 7787  
      large-message-destinations = [  
        "/user/*/largeMessageProxy",  
        "/user/**/largeMessageProxy"  
      ]  
    }  
  }  
}
```

Note that there is also a maximum message size for these side channels. We take a look at how to send messages of arbitrary size in our Akka exercise!

Use side channel for all messages from and to actors named „largeMessageProxy“

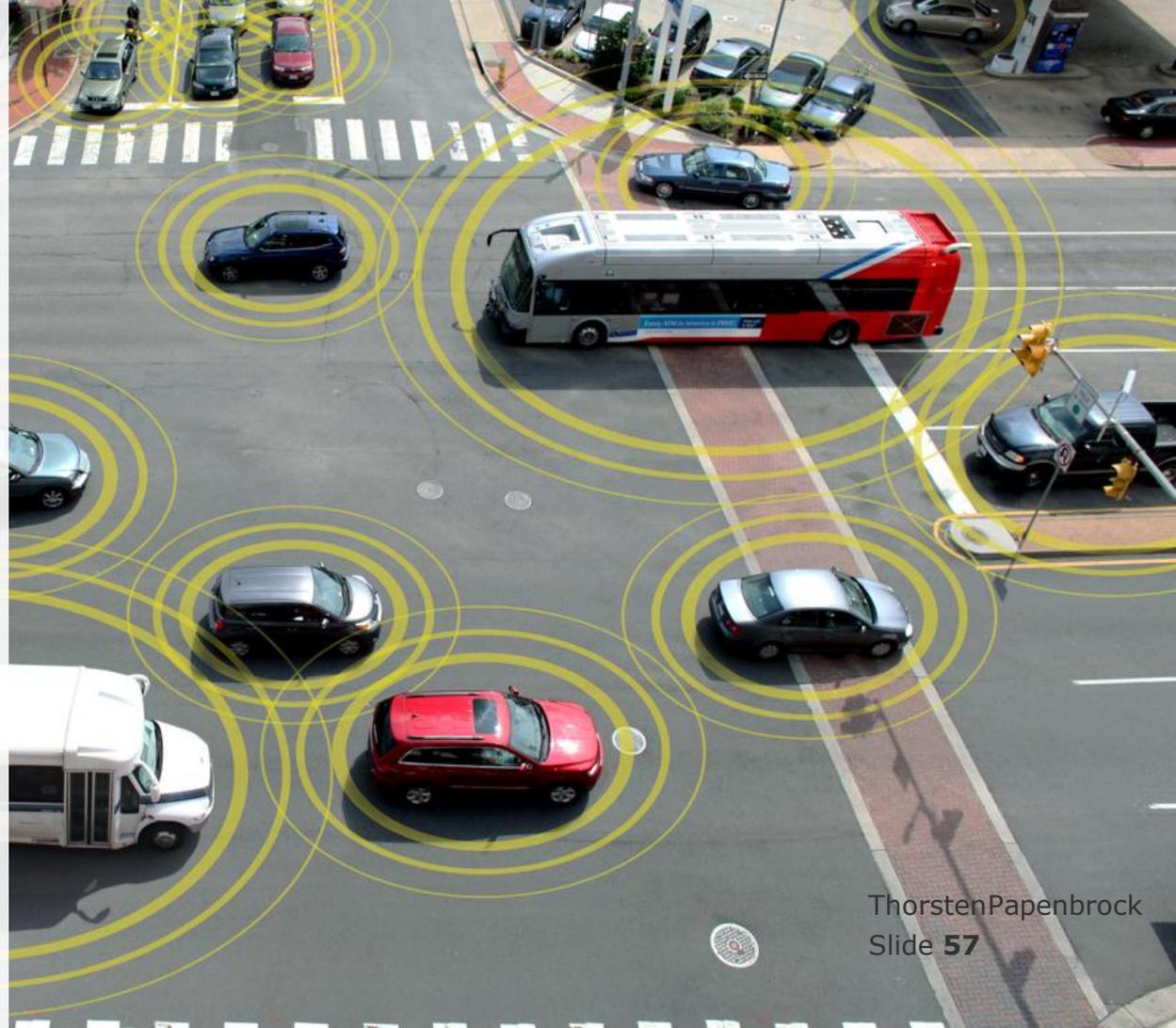
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 56

Akka Actor Programming Hands-on

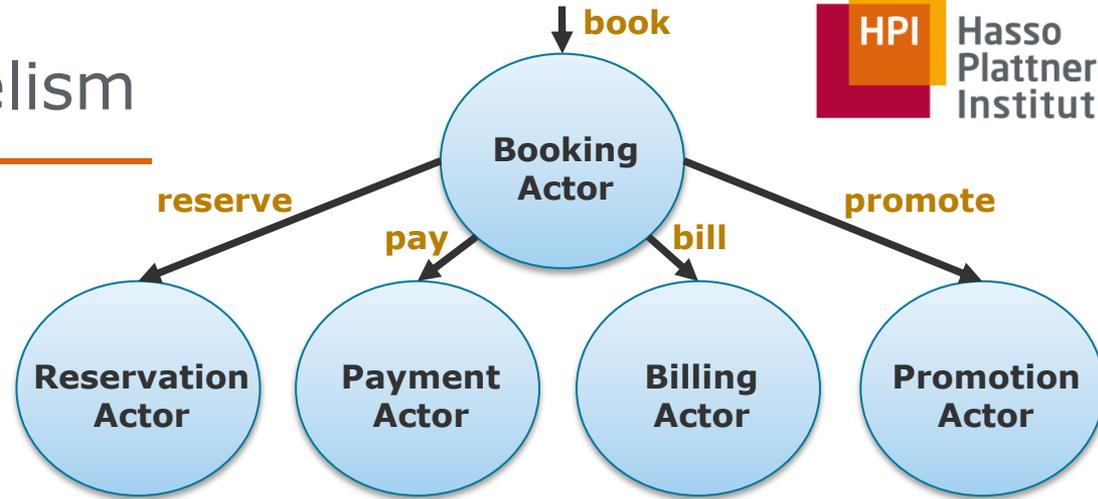
- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- **Parallelization**
- Remoting
- Clustering
- Patterns
- Homework



Task- and Data-Parallelism

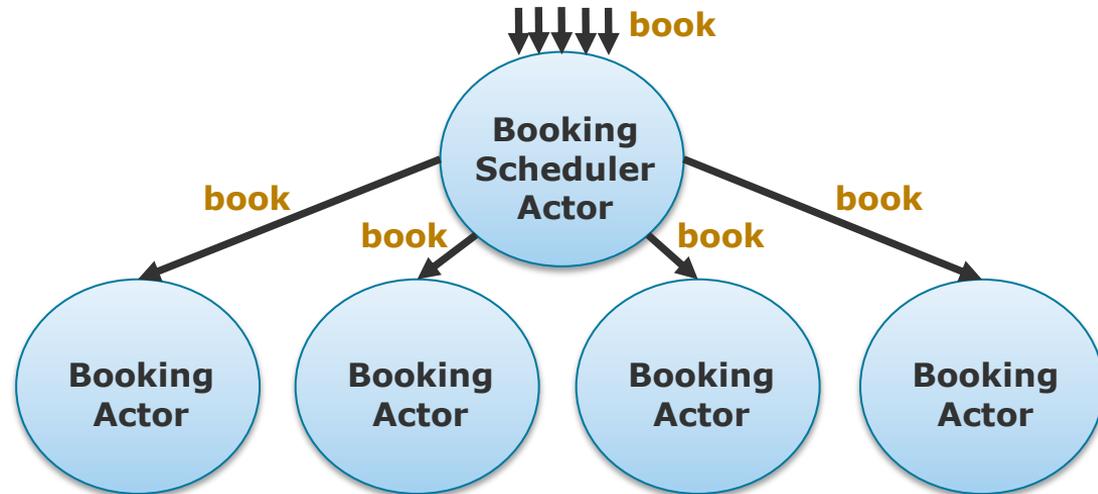
Task-parallelism

- Distribute sub-tasks to different actors



Data-parallelism

- Distribute chunks of data to different actors



Dynamic Parallelism

- Actors often delegate work if they are responsible for ...
 - many tasks.
 - compute-intensive tasks (with many subtasks).
 - data-intensive tasks (with independent partitions).
- Work can be delegated to a dynamically managed pool of worker actors.



Task-parallelism



Data-parallelism

Task Scheduling

- Strategies (see package `akka.routing`):
 - RoundRobinRoutingLogic
 - BroadcastRoutingLogic
 - RandomRoutingLogic
 - ...
 - SmallestMailboxRoutingLogic
 - ConsistentHashingRoutingLogic
 - BalancingRoutingLogic



Push Propagation!

**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **59**

```
Router workerRouter = new Router(new SmallestMailboxRoutingLogic());
```

```
for (int i = 0; i < this.numberOfWorkers; i++) {  
    workerRouter = workerRouter.addRoutee(this.context().actorOf(Worker.props()));  
}
```

Scala world: All objects are immutable!

```
for (WorkPackage workMessage : this.workPackages) {  
    workerRouter.route(workMessage, this.self());  
}
```

Logic defines the worker to be chosen.

Task Scheduling

- Strategies (see package `akka.routing`):
 - RoundRobinRoutingLogic
 - BroadcastRoutingLogic
 - RandomRoutingLogic
 - ...
 - SmallestMailboxRoutingLogic
 - ConsistentHashingRoutingLogic
 - BalancingRoutingLogic

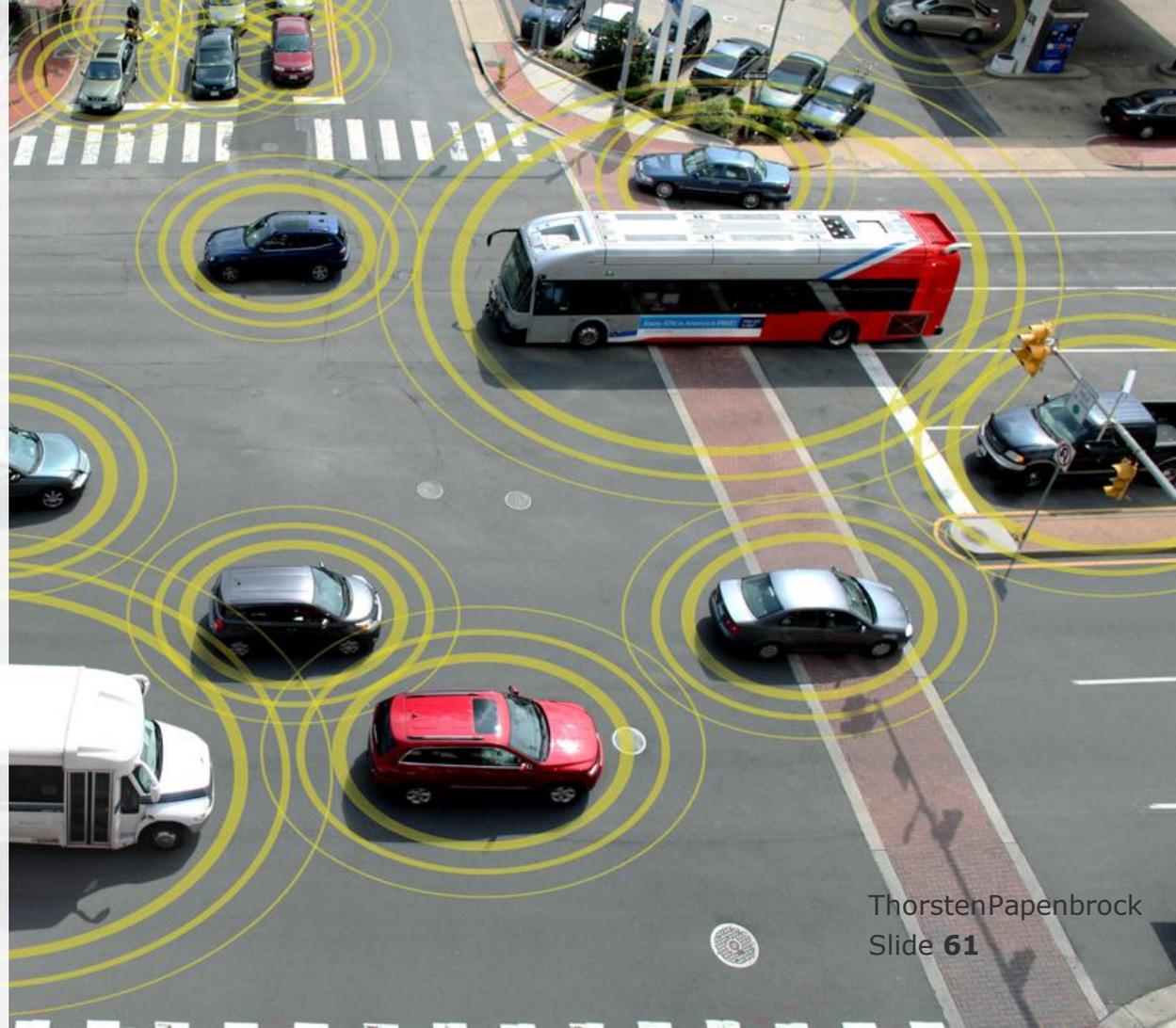
Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **60**

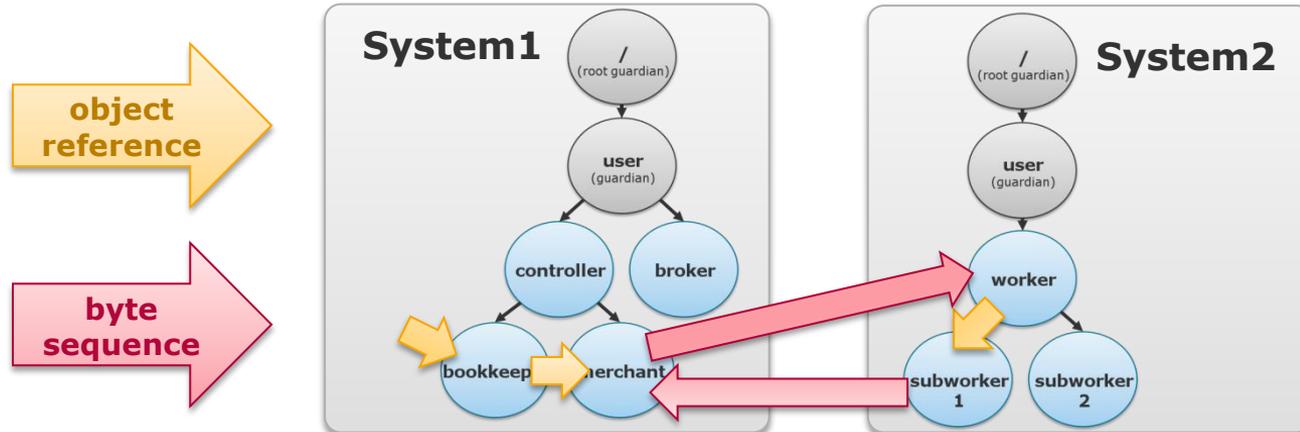
Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- **Remoting**
- Clustering
- Patterns
- Homework



Serialization

- Only messages to **remote actors** are serialized!
 - Communication within one system: **language-specific data types**
 - Pointers and primitive values
 - Communication via process boundaries: **transparent serialization**
 - Serializable, Kryo, Protocol Buffers, ... (configurable)



**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **62**

Remoting Serialization

application.conf

```
akka {  
  actor {  
    provider = remote  
    serializers {  
      java = "akka.serialization.JavaSerializer"  
      kryo = "com.twitter.chill.akka.ConfiguredAkkaSerializer"  
      proto = "akka.remote.serialization.ProtoBufSerializer"  
      my = "de.hpi.myalgo.serialization.MyMessageSerializer"  
    }  
    serialization-bindings {  
      "java.io.Serializable" = kryo  
      "de.hpi.myalgo.serialization.MyMessage" = my  
    }  
  }  
  remote {  
    artery {  
      [...]  
    }  
  }  
}
```

Known object serializers

kryo for all serializable messages

my for MyMessage messages

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **63**

Remoting Serialization

A Java **Serializable** class must do the following:

1. Implement the `java.io.Serializable` interface.
2. Identify the fields that should be serializable.
 - Means: Declare non-serializable fields as "transient".
3. Have access to the no-arg constructor of its first non-serializable superclass.
 - Means: Define no-arg constructors only if non-serializable superclasses exists.

Usually no no-arg constructor needed.

<https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html#a4539>

A Java **Kryo** class must do the following:

By default, if a class has a zero argument constructor, then it is invoked via `ReflectASM` or reflection; otherwise, an exception is thrown.

No-arg constructor needed!

<https://github.com/EsotericSoftware/kryo/blob/master/README.md>

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **64**

```
public class Worker extends AbstractLoggingActor {
```

```
    public static class MyMessage implements Serializable {  
        private static final long serialVersionUID = -6823011111281387872L;  
        private MyMessage() {}  
        private MyMessage(String myText, int myValue) {  
            this.myText = myText;  
            this.myValue = myValue;  
        }  
        private String myText;  
        private int myValue;  
        public String getMyText() {  
            return this.myText;  
        }  
        public int getMyValue() {  
            return this.myValue;  
        }  
    }  
}
```

Default constructor for serialization

All-args constructor for instantiation

The "real" message attributes

Getters for private message data;
no need for setters due to message immutability

So much shorter in Scala...

```
case class MyMessage(myText: String, myValue: Int)
```

```
@Override
```

```
public Receive createReceive() {  
    return receiveBuilder()  
        .match(MyMessage.class, s -> this.sender().tell(new OtherActor.YourMessage(), this.self()))  
        .matchAny(object -> this.log().error("Could not understand received message"))  
        .build();  
}
```

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 65

```
import lombok.AllArgsConstructor;  
import lombok.Getter;
```

Lombok provides aspects for generating most Java boilerplate code

```
public class Worker extends AbstractLoggingActor {
```

```
    @Getter @AllArgsConstructor
```

@Data for getters and setters

```
    public static class MyMessage implements Serializable {  
        private static final long serialVersionUID = -6823011111281387872L;  
        private MyMessage() {}  
        private String myText;  
        private int myValue;  
    }  
}
```

No final due to serialization

```
    @Override
```

```
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(MyMessage.class, s -> this.sender().tell(new OtherActor.YourMessage(), this.self()))  
            .matchAny(object -> this.log().error("Could not understand received message"))  
            .build();  
    }  
}
```

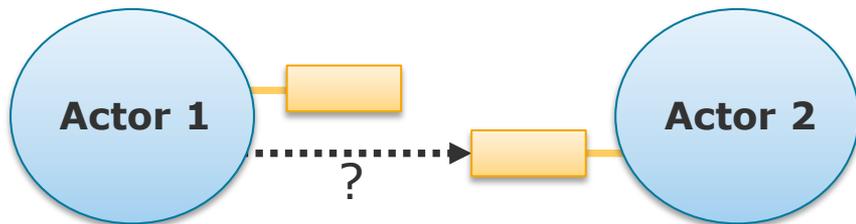
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 66

Remoting Actor Lookup

ActorRefs serve as pointers to local/remote actors.



1. By **construction**:
 - Create a child actor.
2. By **application**:
 - Ask for a reference in your constructor or provide a setter.
3. By **message**:
 - Ask a known actor to send you a reference to another actor.

```
public class Master extends AbstractLoggingActor {  
    private ActorRef worker;
```

1.

A factory pattern

```
    public Master() {  
        this.worker = this.context().actorOf(Worker.props());  
    }
```

```
@Override
```

```
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(ActorRef.class, worker -> this.worker = worker)  
            .matchAny(object -> this.log().error("Invalid message"))  
            .build();  
    }  
}
```

3.

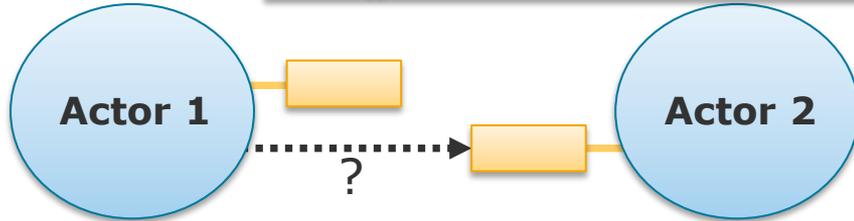
**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide 67

Remoting Actor Lookup

ActorSelection is a logical view to a subtree of an ActorSystem; *tell()* broadcasts to that subtree



1. By **construction**:
 - Create a child actor.
2. By **application**:
 - Ask for a reference in your constructor or provide a setter.
3. By **message**:
 - Ask a known actor to send you a reference to another actor.
4. By **name** (path/URL):
 - Ask the context to create a reference to an actor with a certain URL.

```
public class Master extends AbstractLoggingActor {  
    private ActorRef worker;  
  
    public Master() {  
        Address address = new Address("akka.tcp",  
            "MyActorSystem", "localhost", 7877);  
  
        ActorSelection selection = this.context().system()  
            .actorSelection(String.format(  
                "%s/user/%s", address, "worker"));  
        selection.tell(new HelloMessage(), this.self());  
    }  
}
```

4.

URL:
"akka.tcp://MyActorSystem@localhost:7877/user/worker"

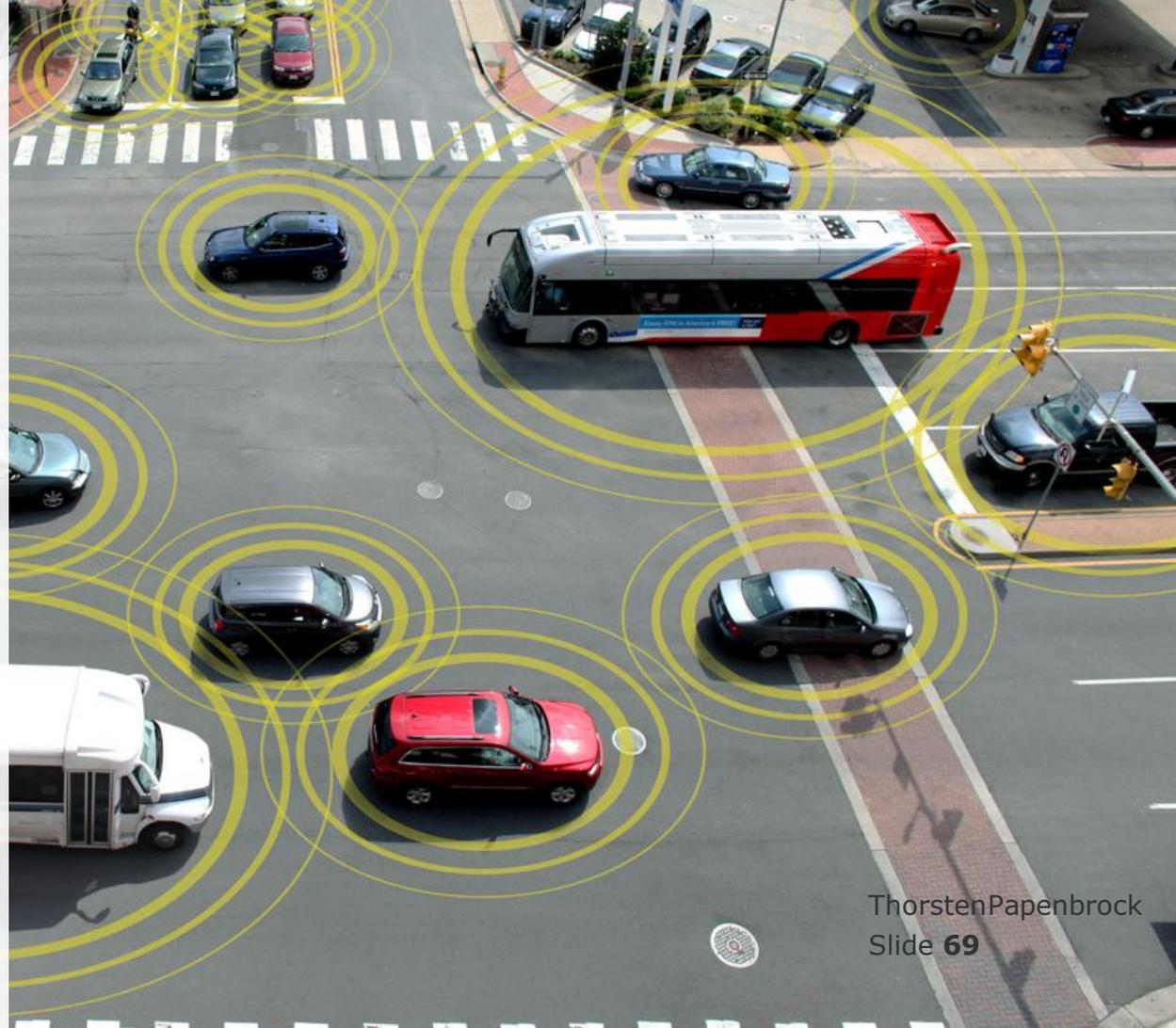
**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **68**

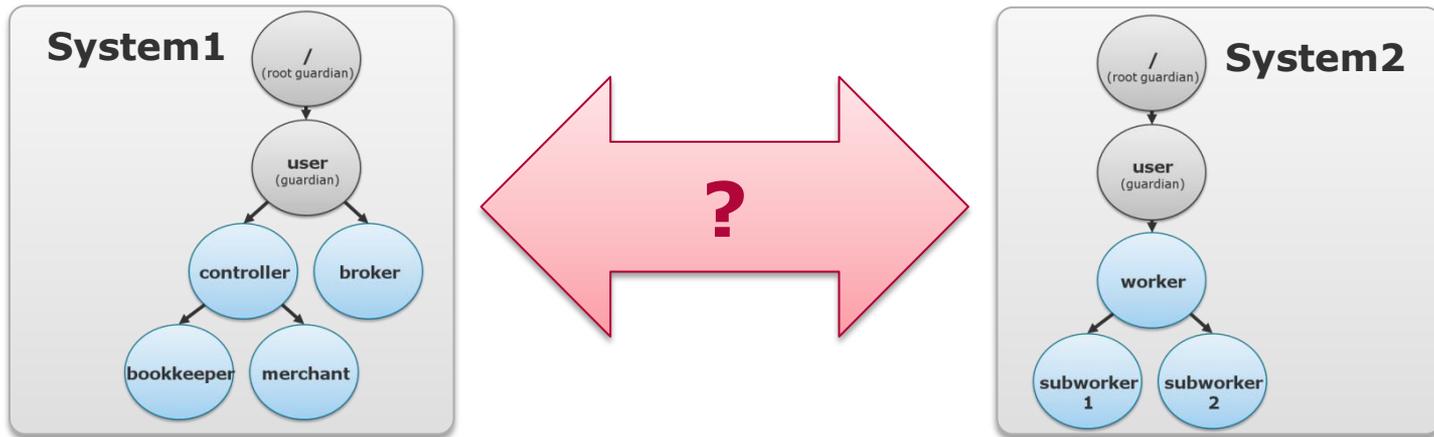
Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- **Clustering**
- Patterns
- Homework



Clustering

Cluster-Awareness



How does System1 know ...

- which other ActorSystems are available?
(the number might even change at runtime!)
- what failures occurred in other ActorSystems?
(single actors but also entire nodes might become unavailable!)
- what roles other ActorSystems take?
(e.g. a master or worker or metrics collector or entirely different application!)

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 70

Clustering

Cluster-Awareness

Akka Actors

Core Actor model classes for concurrency and distribution

Akka Cluster

Classes for the resilient and elastic distribution over multiple nodes

Akka Streams

Asynchronous, non-blocking, backpressured, reactive stream classes

Akka Http

Asynchronous, streaming-first HTTP server and client classes

Cluster Sharding

Classes to decouple actors from their locations referencing them by identity

Akka Persistence

Classes to persist actor state for fault tolerance and state restore after restarts

Distributed Data

Classes for an eventually consistent, distributed, replicated key-value store

Alpakka

Stream connector classes to other technologies

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 71

Clustering Dependency

Clustering capabilities
cluster membership, singletons,
publish/subscribe, cluster client, ...

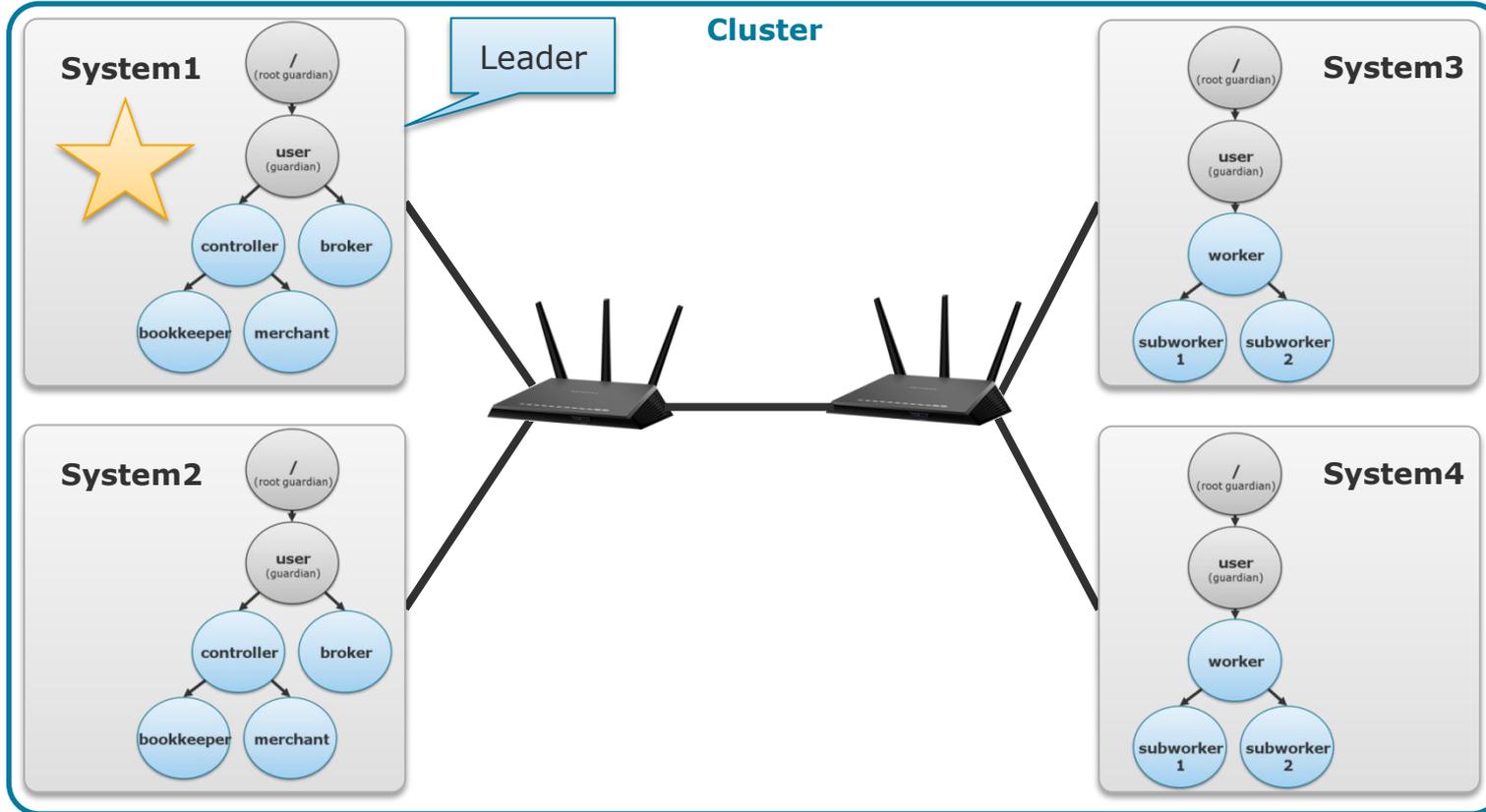
Metrics collection
CPU load, memory consumption, ...

Transparent actors
logical references,
distributed/persisted state, ...

```
<dependencies>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-actor_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-tools_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-metrics_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-cluster-sharding_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  ...
</dependencies>
```

Maven – pom.xml

Clustering Cluster



Distributed Data Management

Akka Actor Programming

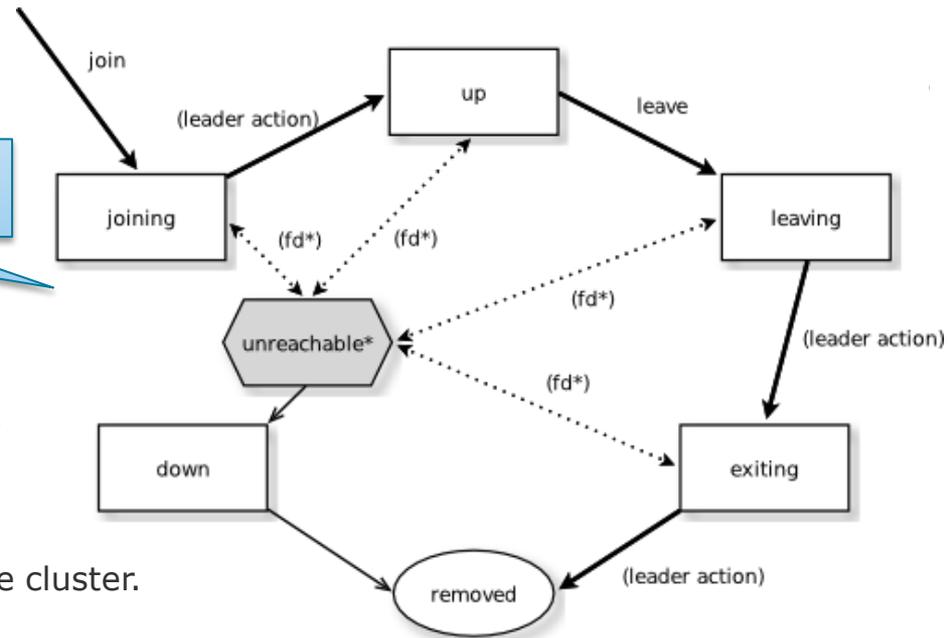
ThorstenPapenbrock
Slide 73

Clustering

Cluster

fd* = failure detector
(see ϕ accrual failure detector later)

- A distributed membership service.
- Runs in all Akka cluster-managed ActorSystems.
- Stores all membership information in a distributed, fully replicated key-value store (= Riak).
- Uses gossiping¹ to propagate cluster changes.
- Elects the eldest node in a cluster as "leader" of the cluster.
- Leader decisions²:
 - state changes to "up", "exiting", and "removed" for members of the cluster
- Member decisions²:
 - state changes to "joining", "leaving", and (via local failure detector) "unreachable"



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 74

[1] <https://doc.akka.io/docs/akka/2.5/common/cluster.html#gossip>

[2] <https://doc.akka.io/docs/akka/2.5/common/cluster.html#membership-lifecycle>

Clustering Configuration

application.conf

```
akka {  
  actor {  
    provider = cluster  
    [...]  
  }  
  remote {  
    artery {  
      [...]  
    }  
  }  
  cluster {  
    min-nr-of-members = 3  
    downing-provider-class = "akka.cluster.sbr.SplitBrainResolverProvider"  
  }  
  coordinated-shutdown {  
    run-by-actor-system-terminate = on  
    terminate-actor-system = on  
  }  
}
```

instead of `local` or `remote`

the cluster membership waits for these numbers of member before it sets their status to up

If cluster nodes are abruptly terminated, the downing-provider resolves the nodes' cluster state changes

Ties the coordinated shutdown to the termination of the ActorSystem

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **75**

```
public static void start(String appName, String host, int port, String seedhost, int seedport) {  
  
    final Config config = ConfigFactory.parseString(  
        "akka.remote.artery.canonical.hostname = \" + host + "\"\n" +  
        "akka.remote.artery.canonical.port = \" + port + "\"\n" +  
        "akka.cluster.roles = [worker]\n" +  
        "akka.cluster.seed-nodes = [\"akka://\" + appName + "@\" + seedhost + \":\" + seedport + "\"]")  
        .withFallback(ConfigFactory.load("application"));  
  
    final ActorSystem system = ActorSystem.create(appName, config);  
  
    system.registerOnTermination(...);  
    Cluster.get(system).registerOnMemberUp(...);  
    Cluster.get(system).registerOnMemberRemoved(...);  
  
    [...]  
}
```

Sets ActorSystem-specific configuration parameters dynamically.

Sets the seed node(s) for initial connect; any connected node can be a seed node; first node connects to itself, which creates the cluster.

Setting-up callbacks allows the application's main thread to end before the ActorSystem is actually up or ends.

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 76

```
public static void start(String appName, String host, int port, String seedhost, int seedport) {  
    [...]  
    Cluster.get(system).registerOnMemberUp(new Runnable() {  
        @Override  
        public void run() {  
            for (int i = 0; i < 10; i++)  
                system.actorOf(WorkerActor.props(), "worker" + i);  
        }  
    });  
}
```

Run when this node has been set to status "up".

Create local actors (and send initial messages) when "up".

Retrieve the cluster object when needed
(to access cluster events, failure detector, node status, ...).

```
Cluster cluster = Cluster.get(this.context().system());
```

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide 77

```
public static void start(String appName, String host, int port, String seedhost, int seedport) {  
    [...]  
    Cluster.get(system).registerOnMemberRemoved(new Runnable() {  
        @Override  
        public void run() {  
            system.terminate();  
            new Thread() {  
                @Override  
                public void run() {  
                    try {  
                        Await.ready(system.whenTerminated(), Duration.create(10, TimeUnit.SECONDS));  
                    } catch (Exception e) {  
                        System.exit(-1);  
                    }  
                }  
            }.start();  
        }  
    });  
}
```

Run when this node has been "removed".

Terminate the ActorSystem.

Let a dedicated thread
wait for the ActorSystem to terminate;
it kills the application
if the ActorSystem does not terminate in time.

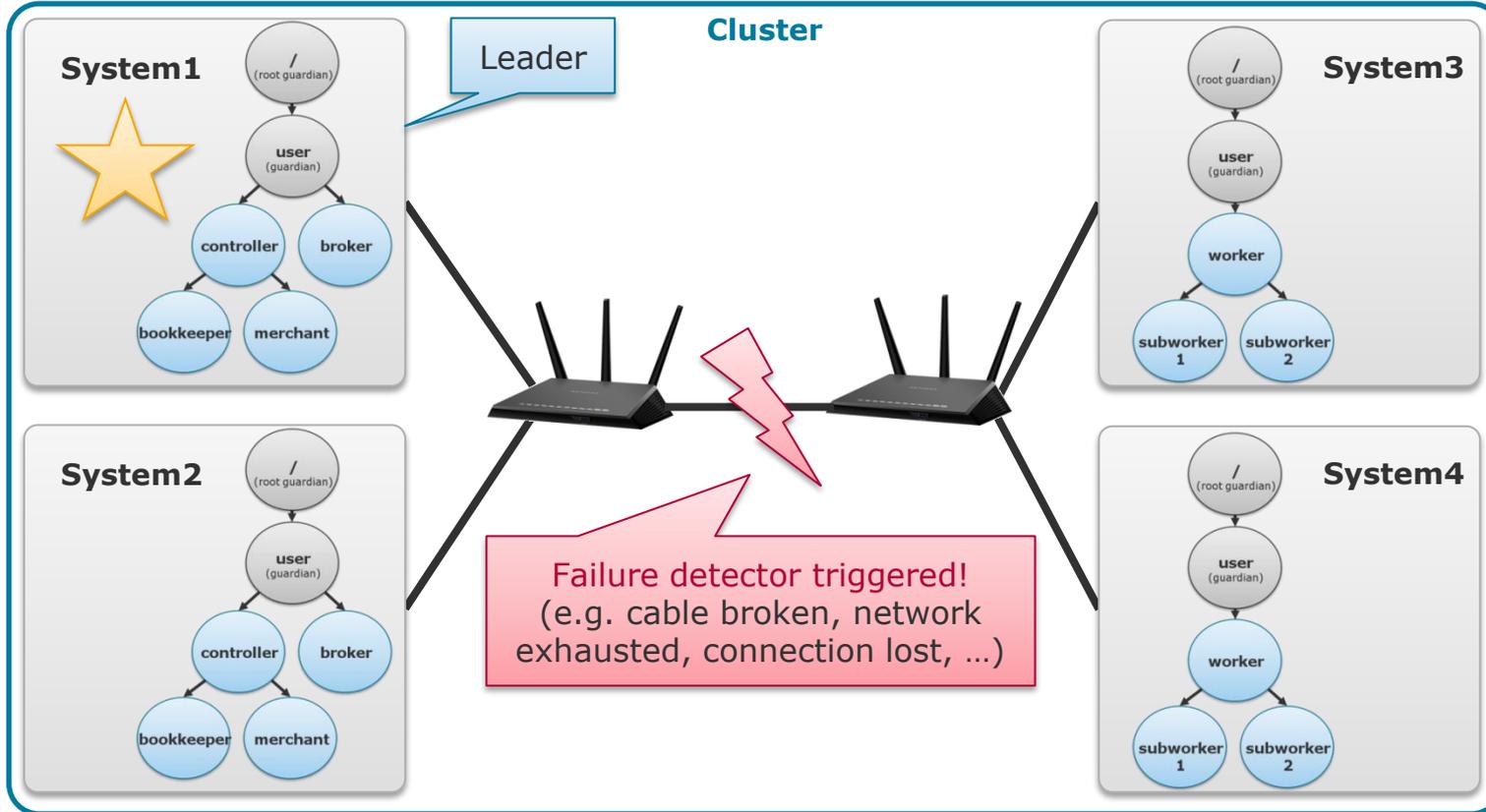
**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **78**

Clustering

Cluster Partitioning



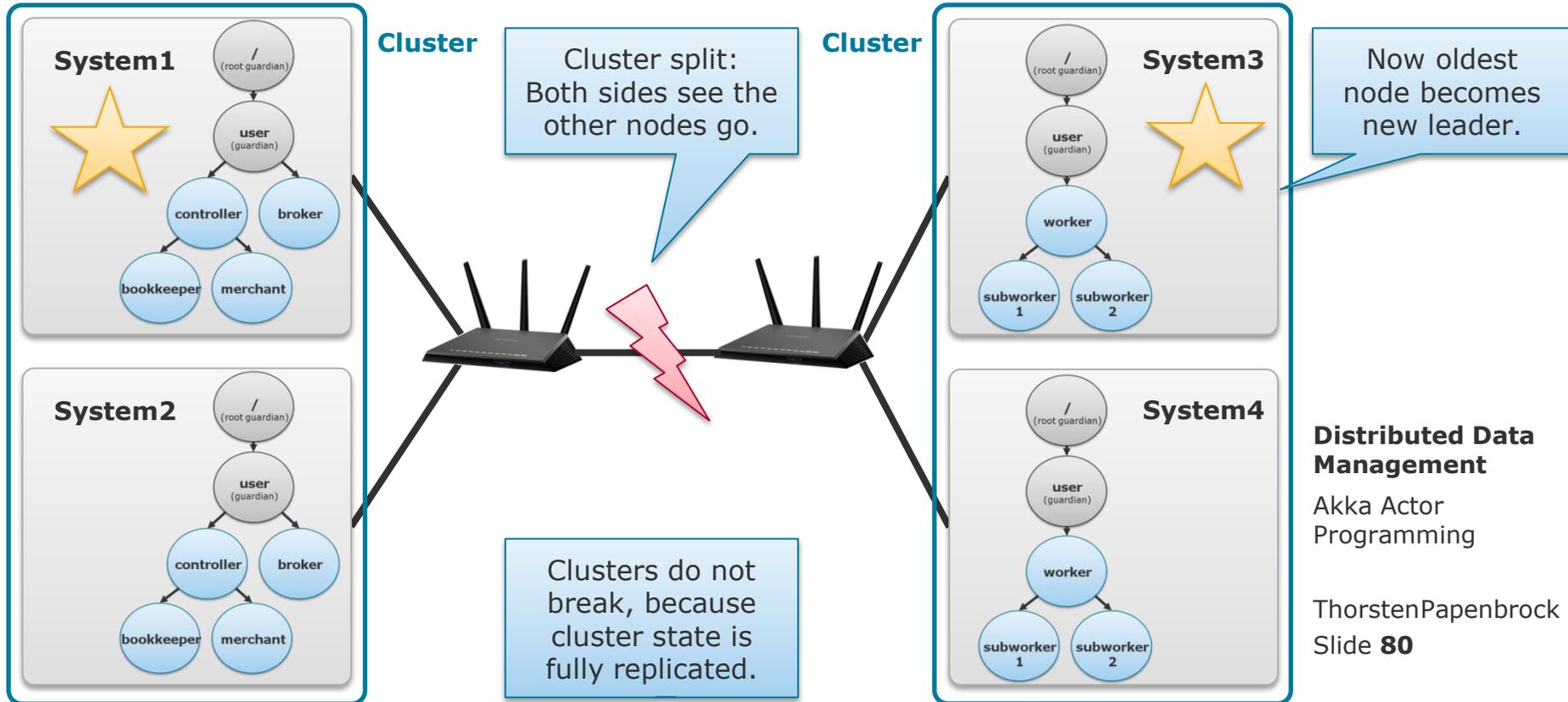
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 79

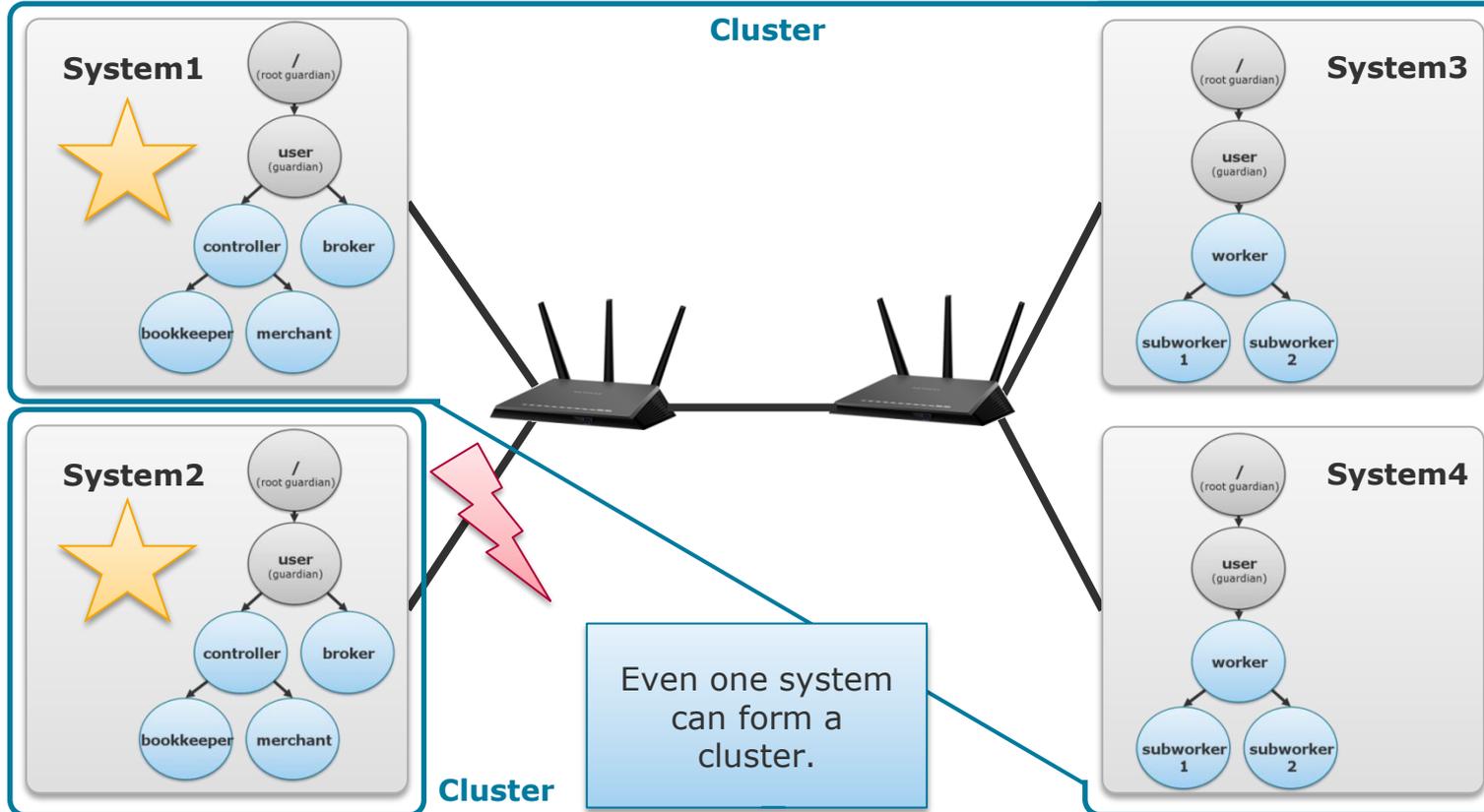
Clustering

Cluster Partitioning



Clustering

Cluster Partitioning



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **81**

Clustering

Cluster Partitioning

If the cluster partitions, ...

- each partition will form its own cluster.
- no *onMemberRemoved()* callback is triggered, because every node stays in some cluster.
- each cluster keeps track of all removed ActorSystems so that "[...] the same actor system can never join a cluster again once it's been removed from that cluster"¹.
(Otherwise, the cluster could run into split-brain situations (= two leaders))

To re-unite the nodes, ...

1. identify the "main" cluster
2. terminate all "non-main" clusters
3. and restart ActorSystems on all affected nodes.

[1] <https://doc.akka.io/docs/akka/2.5/common/cluster.html#cluster-specification>

Using the Cluster

```
public class WorkerActor extends AbstractActor {  
    @Override  
    public void preStart() {  
        Cluster.get(this.context().system()).subscribe(this.self(), MemberUp.class);  
    }  
  
    @Override  
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(CurrentClusterState.class, this::registerAll)  
            .match(MemberUp.class, message -> this.register(message.member()))  
            .match(WorkMessage.class, this::handle)  
            .build();  
    }  
  
    private void registerAll(CurrentClusterState message) {  
        message.getMembers().forEach(member -> {  
            if (member.status().equals(MemberStatus.up()))  
                this.register(member);  
        });  
    }  
  
    private void register(Member member) {  
        if (member.hasRole("master"))  
            this.getContext().actorSelection(member.address() + "/user/masteractor")  
                .tell(new RegistrationMessage(), this.self());  
    }  
}
```

Subscribe to cluster events

Register at all masters after creation

Register at any new master

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 83

Clustering

Member Events

```
public class MasterActor extends AbstractActor {  
    @Override  
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(RegistrationMessage.class, this::handle)  
            .match(Terminated.class, this::handle) .build();  
    }  
  
    private void handle(RegistrationMessage message) {  
        this.context().watch(this.sender());  
        this.workers.add(this.sender());  
        this.sender().tell(new WorkMessage());  
    }  
  
    private void handle(Terminated message) {  
        this.context().unwatch(message.getActor());  
        this.workers.remove(this.sender());  
    }  
}
```

Simply watch the remote workers

**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **84**

Clustering

Cluster-Aware Scheduler

```
int maxWorkersPerNode = 10;
int maxWorkersPerCluster = 1000000;
boolean allowLocalWorkers = true;
Set<String> roles = new HashSet<>(Arrays.asList("worker"));

ActorRef router = system.actorOf(
  new ClusterRouterPool(
    new AdaptiveLoadBalancingPool(SystemLoadAverageMetricsSelector.getInstance(), 0),
    new ClusterRouterPoolSettings(maxWorkersPerCluster, maxWorkersPerNode,
      allowLocalWorkers, roles)))
  .props(Props.create(WorkerActor.class), "router");
```

Automatically spawns 10 workers per node and not more than 1,000,000 per cluster

Scheduling strategy: average load

application.conf

```
akka {
  extensions = ["akka.cluster.metrics.ClusterMetricsExtension"]
  cluster.metrics.native-library-extract-folder=${user.dir}/target/native
}
```

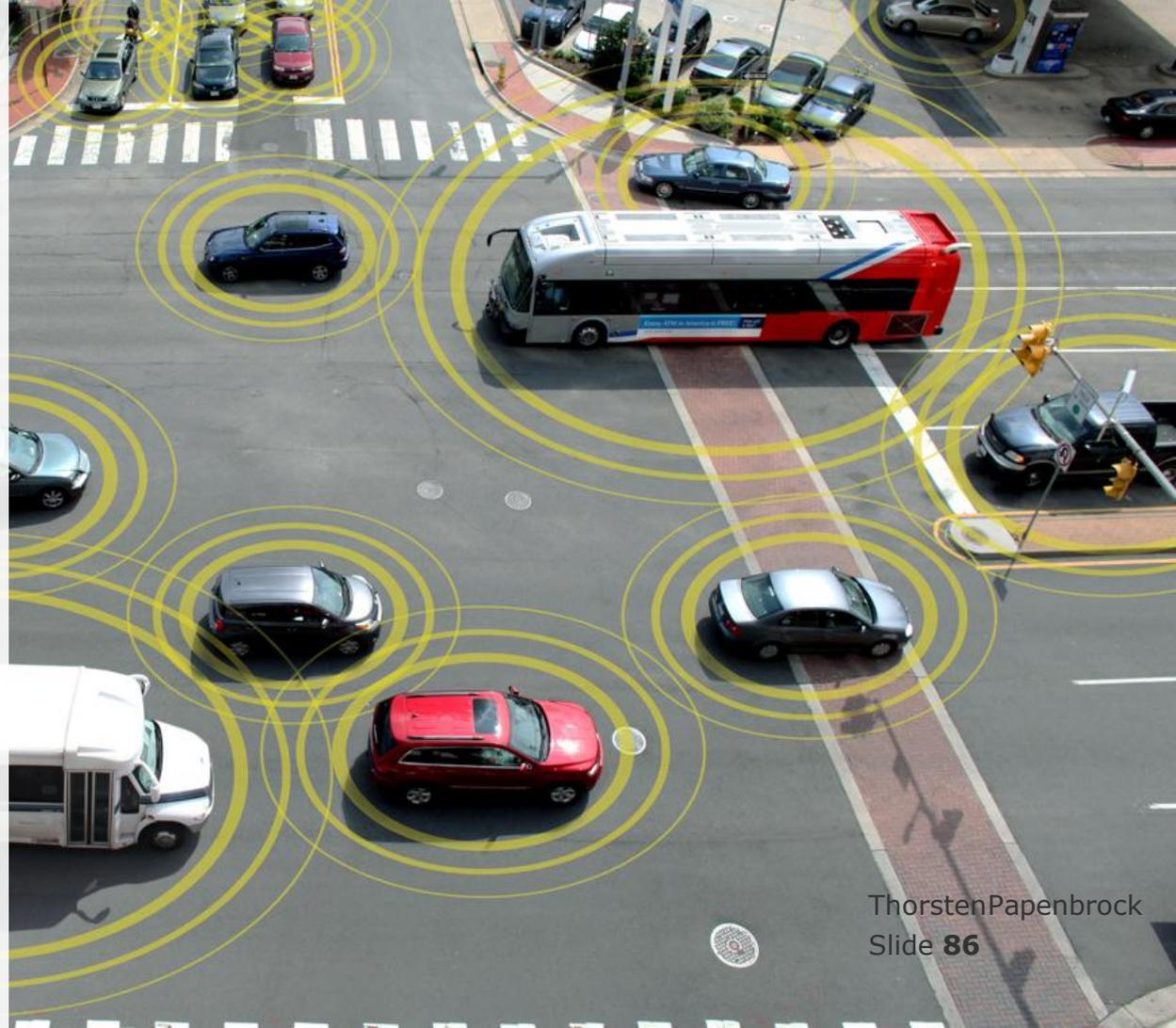
Distributed Data Management

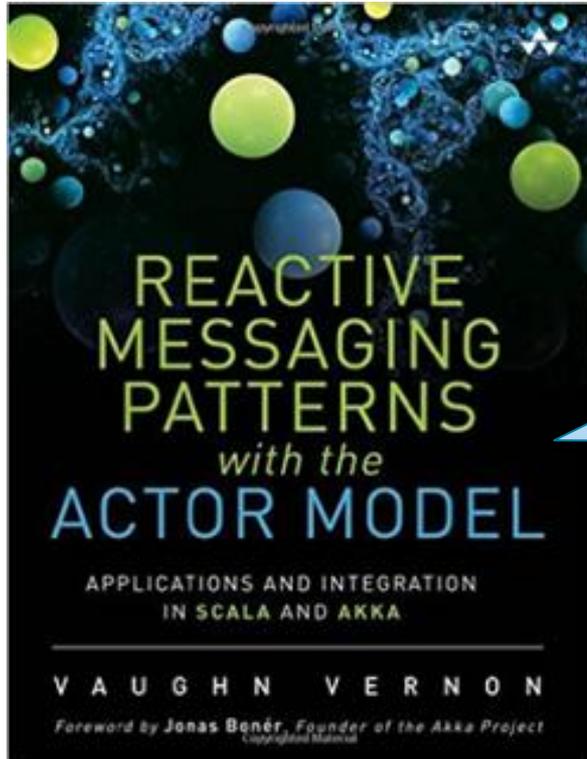
Akka Actor Programming

ThorstenPapenbrock
Slide 85

Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- **Patterns**
- Homework





Actor programming is a mathematical model that defines basic rules for communication (not a style guide for architecture)

Writing actor-based systems is based on patterns

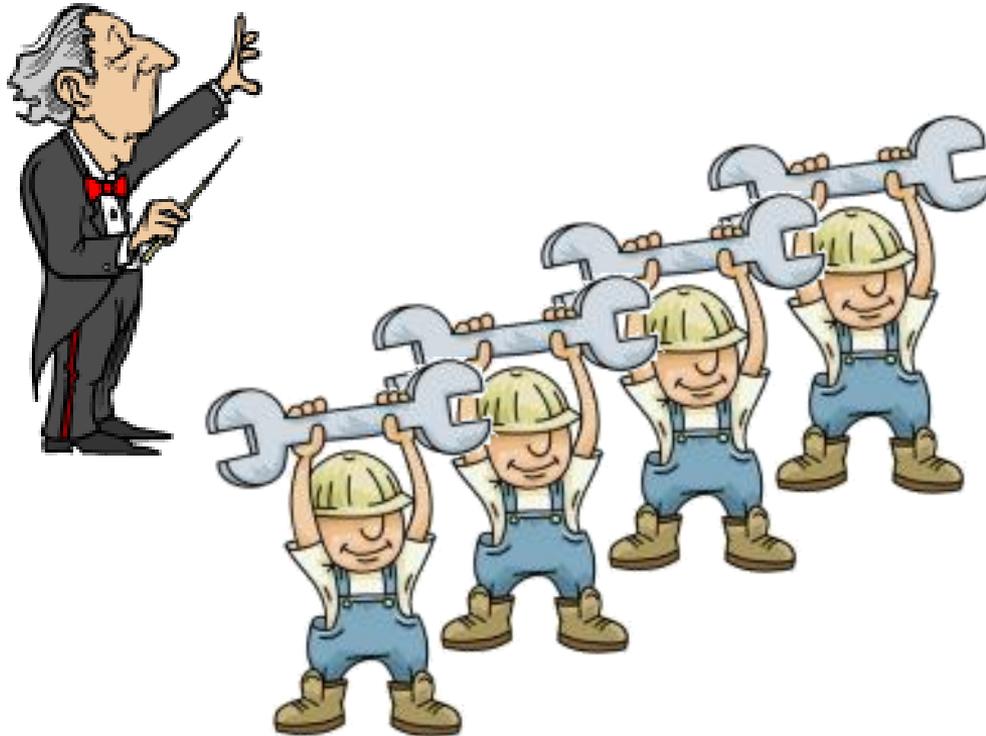
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 87

Patterns

Master/Worker



Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **88**

Patterns

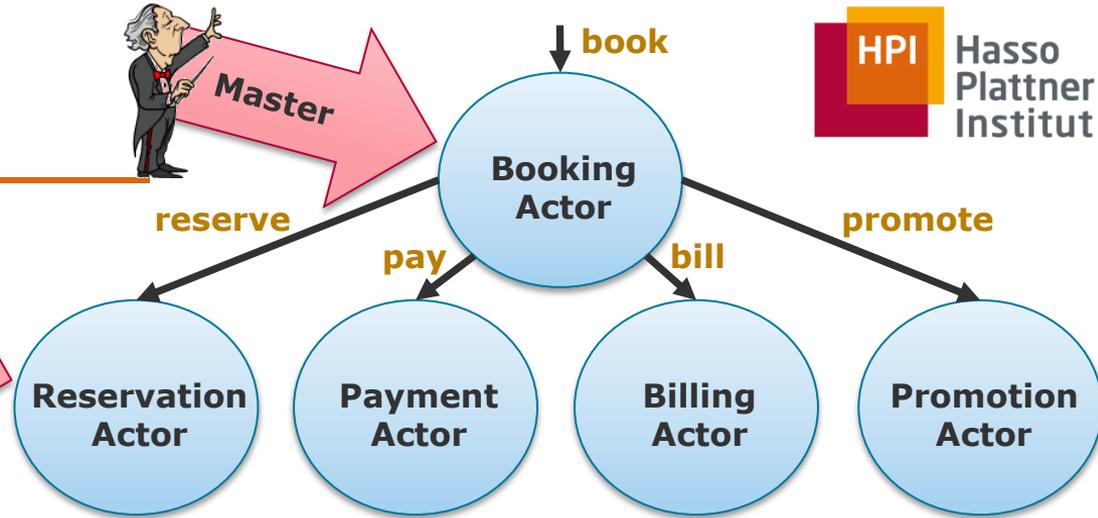
Master/Worker

Task-parallelism

- Distribute sub-tasks to different actors



Worker

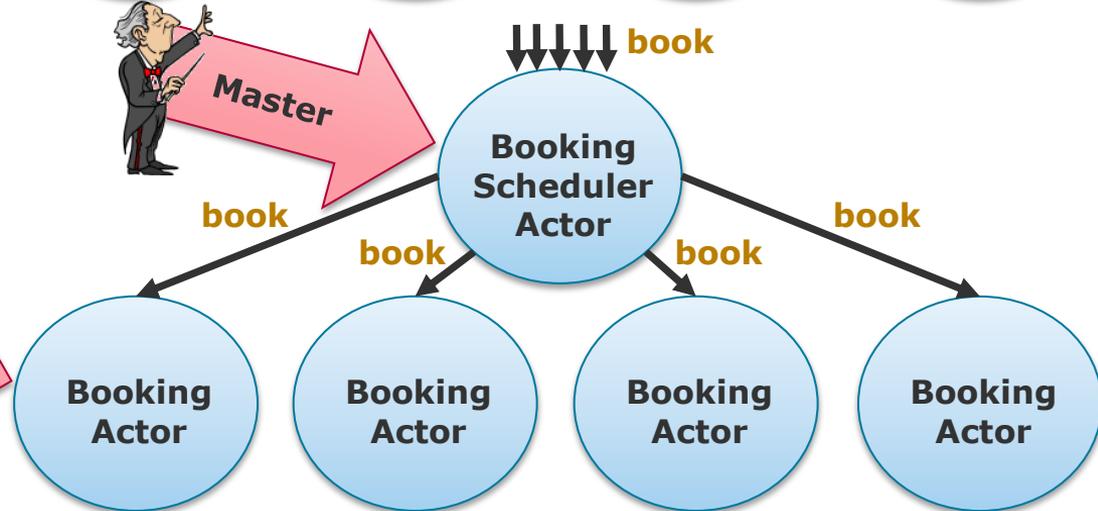


Data-parallelism

- Distribute chunks of data to different actors



Worker

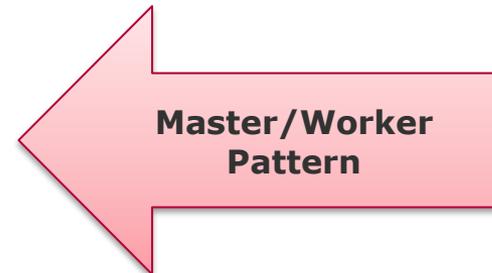


Patterns

Master/Worker

Work Propagation

- **Producer** actors generate work for other **consumer** actors
- **Push propagation:**
 - Producers send work immediately
 - Work queued in inboxes
 - **Fast propagation; risk for congestion**
- **Pull propagation:**
 - Consumers ask for more work
 - Work queued in producer
 - **Slower propagation; no congestion**



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 90

A concept for ...

1. Parallelization
2. Fault-Tolerance

Master

- Splits the task into work packages.
- Schedules the work packages to known workers.
- Watches available workers (register new workers; detect and unregister failed workers).
- Monitors task completion (assign pending work packages; re-assign failed work packages).
- Assemble final result (from partial work package results).
 - Does not know how to solve the individual tasks!

Worker

- Register at master.
- Accept and process work packages.
 - Does not know the overall task!

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **91**

Patterns

Master/Worker

Master

```
public class OneTaskMasterActor extends AbstractActor {  
    private final Queue<WorkMessage> unassignedWork = new LinkedList<>();  
    private final Queue<ActorRef> idleWorkers = new LinkedList<>();  
    private final Map<ActorRef, WorkMessage> busyWorkers = new HashMap<>();  
    private TaskMessage task;  
    private Result result;  
  
    @Override  
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(RegistrationMessage.class, this::handle)  
            .match(Terminated.class, this::handle)  
            .match(TaskMessage.class, this::handle)  
            .match(CompletionMessage.class, this::handle)  
            .build();  
    }  
    [...]  
}
```

work packages

worker

work packages + worker

Watch and try to assign work

Un-watch and re-assign work

Split and assign to workers

Collect and send new work

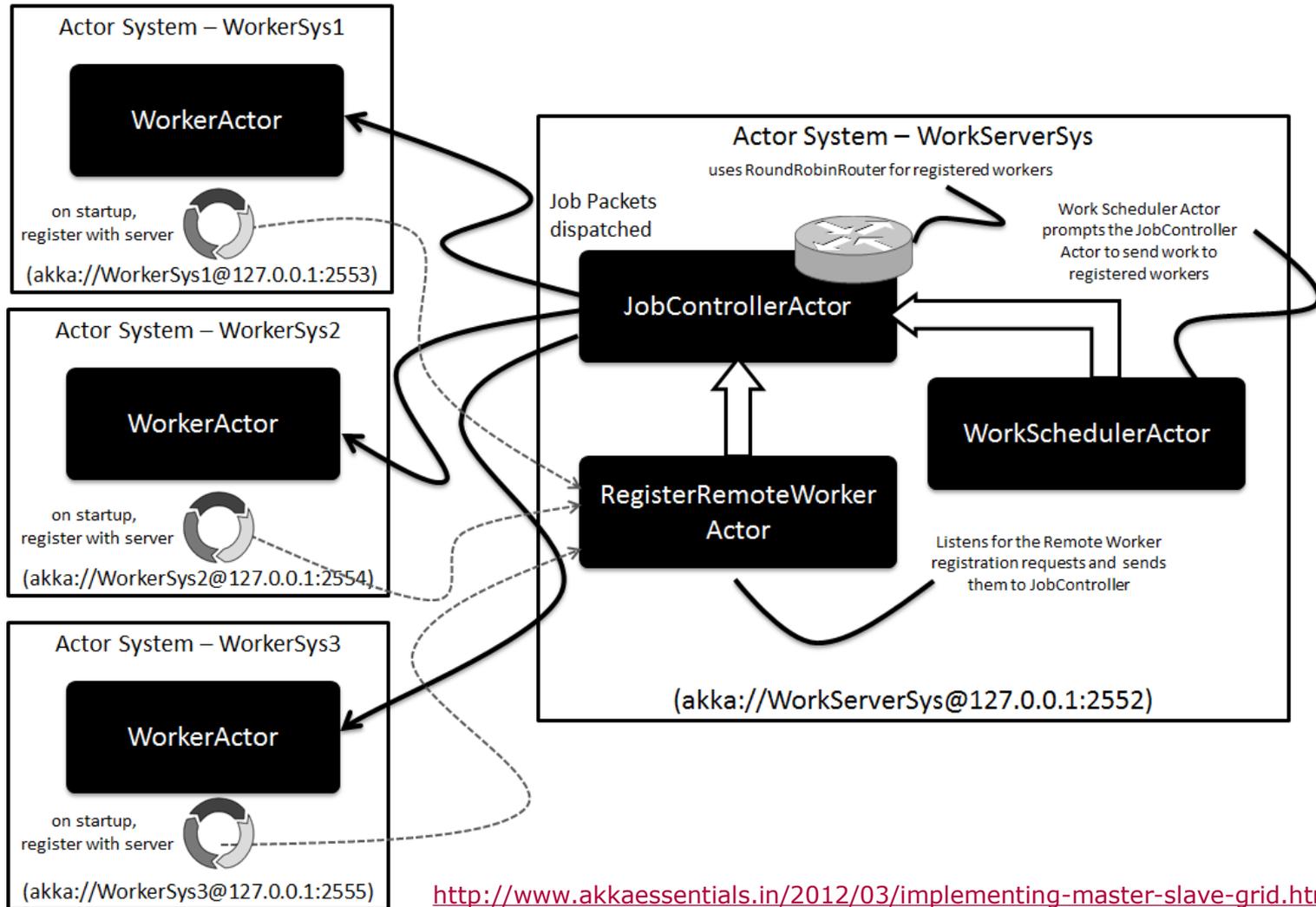
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **92**

Patterns Master/ Worker

Example

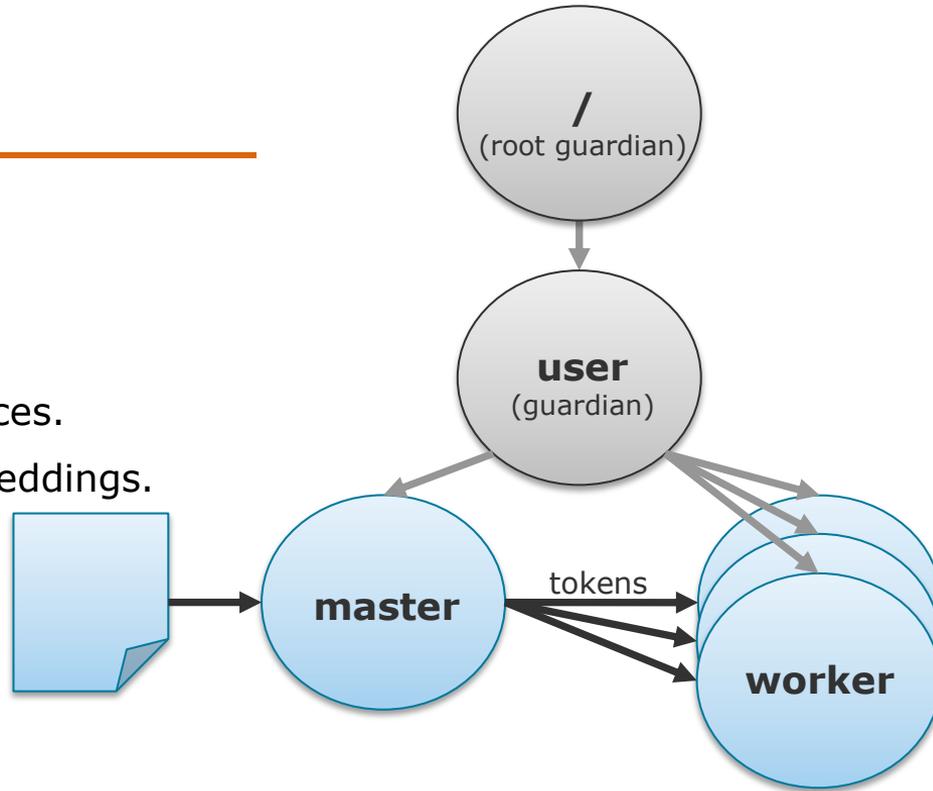


Patterns

Master/Worker

“The eager producers”

- Task:
 1. Read a file.
 2. Tokenize the sentences.
 3. Calculate token embeddings.
- Push-Propagation:
 - Each input file is read and tokenized by one master.
 - Each token range is processed by one worker.
- Works great for **one** input file!



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **94**

Patterns

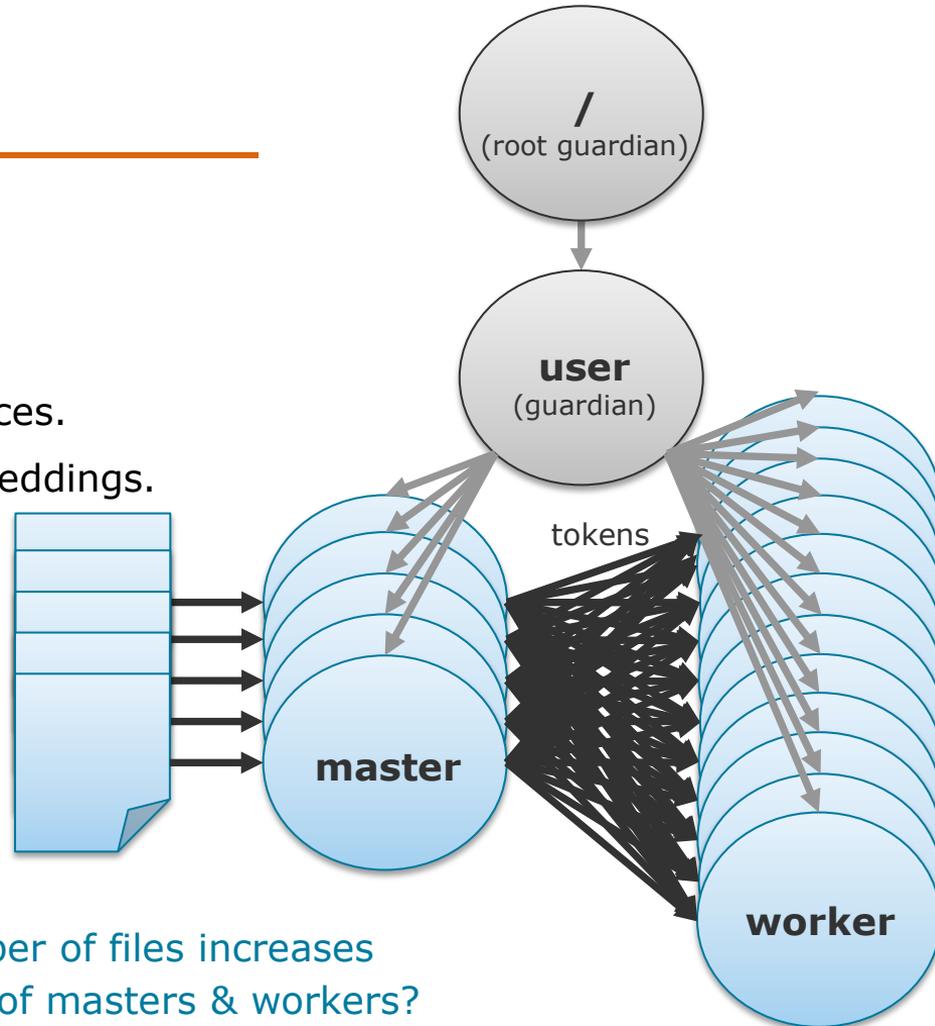
Master/Worker

“The eager producers”

- Task:
 1. Read a file.
 2. Tokenize the sentences.
 3. Calculate token embeddings.

- Push-Propagation:

- Each input file is read and tokenized by one master.
- Each token range is processed by one worker.



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 95

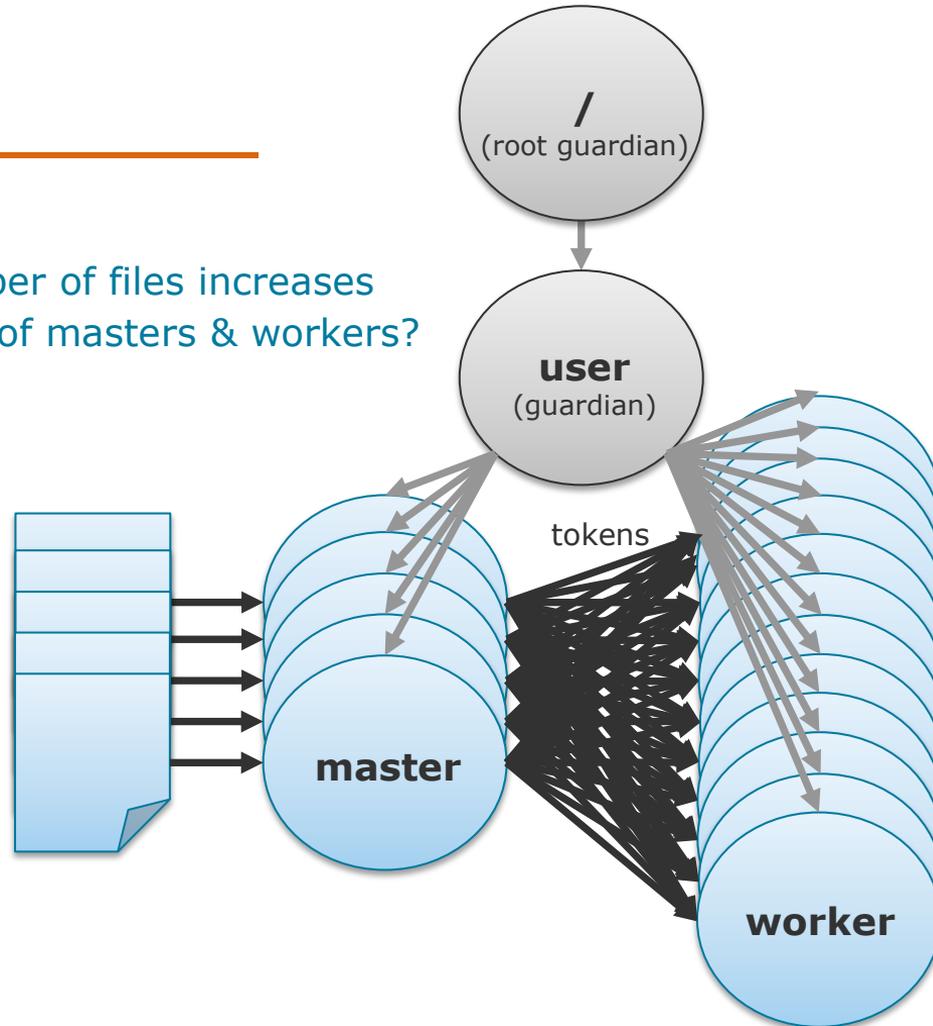
- What happens if the number of files increases and we scale the number of masters & workers?

Patterns

Master/Worker

“The eager producers”

- What happens if the number of files increases and we scale the number of masters & workers?
- The masters will take and block more threads. (file reading takes long!)
- The workers will get less CPU time.
- The work will pile up in the in-boxes of the workers.
- The system will get slow and OOM at some point.



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **96**

Patterns

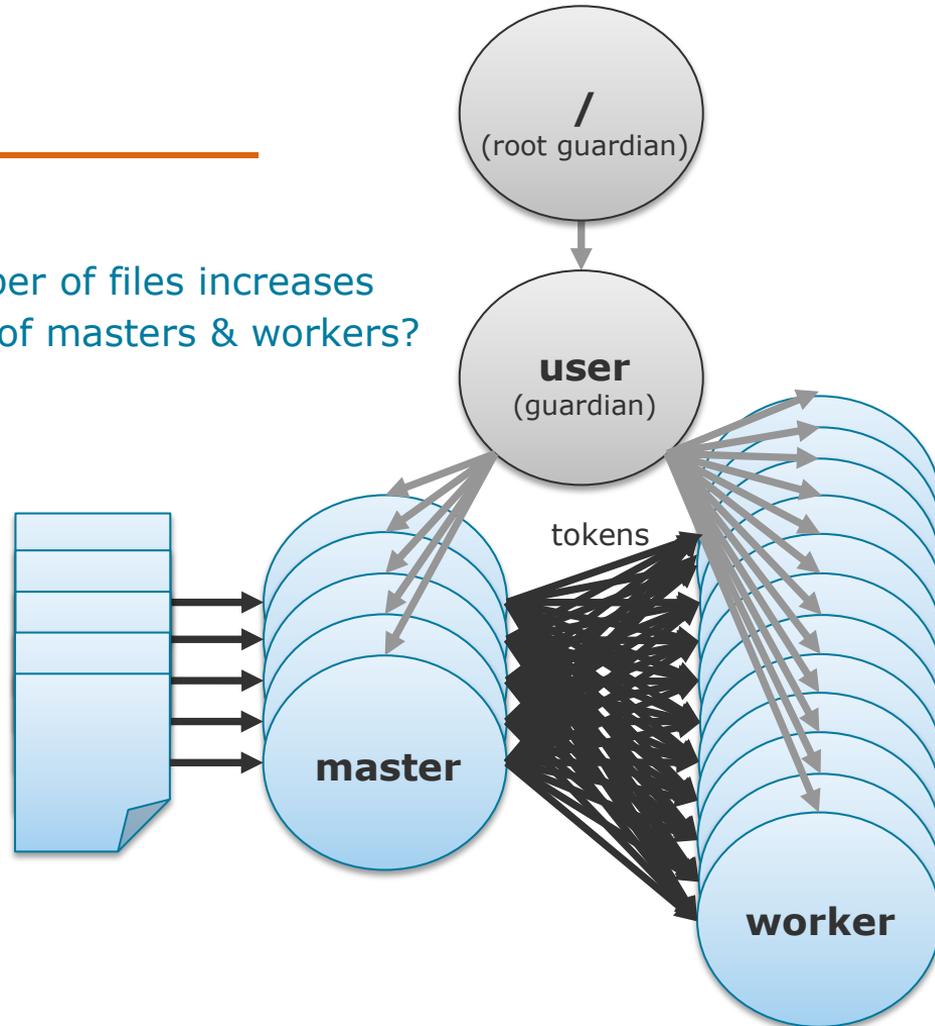
Master/Worker

“The eager producers”

- What happens if the number of files increases and we scale the number of masters & workers?

Solutions:

- Pull-Propagation: Pause long running tasks and free threads.
- File limit: Control the number of actors with long running tasks (in particular fewer than number of cores).



Distributed Data Management

Akka Actor Programming

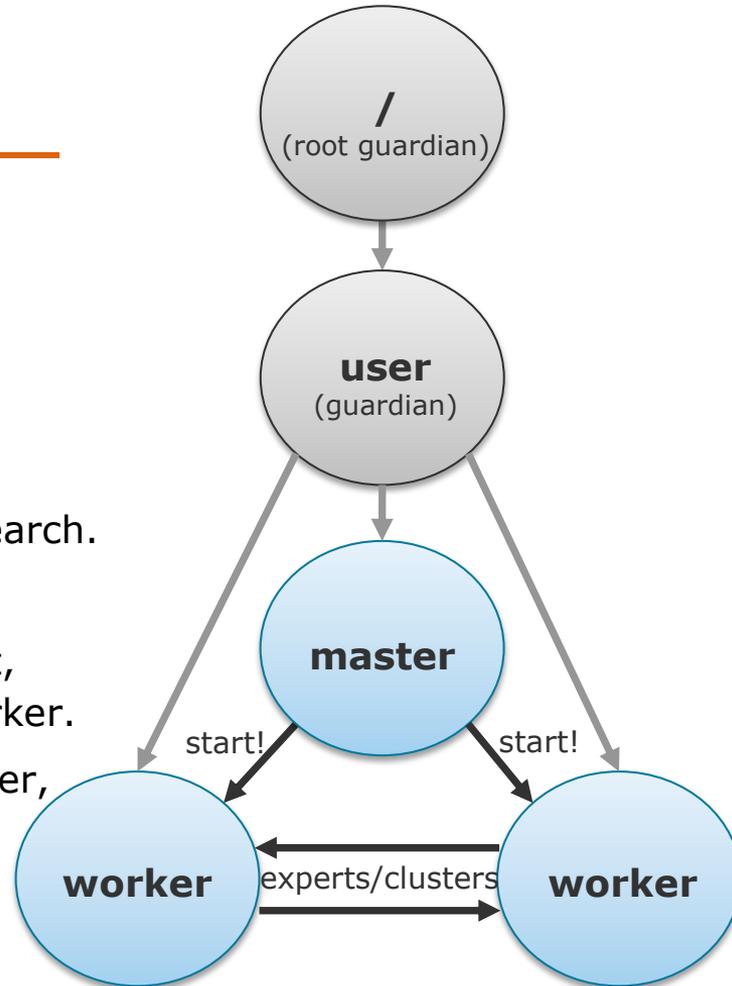
ThorstenPapenbrock
Slide 97

Patterns

Master/Worker

“The non-reactive workers”

- Task:
 - Search for expert-users in social networks.
- Approach:
 - Each worker starts a random search.
 - For search pruning:
 - If a worker finds an expert, it sends it to the other worker.
 - If a worker finishes a cluster, it sends a notification to the other worker.
- What could be a problem here?



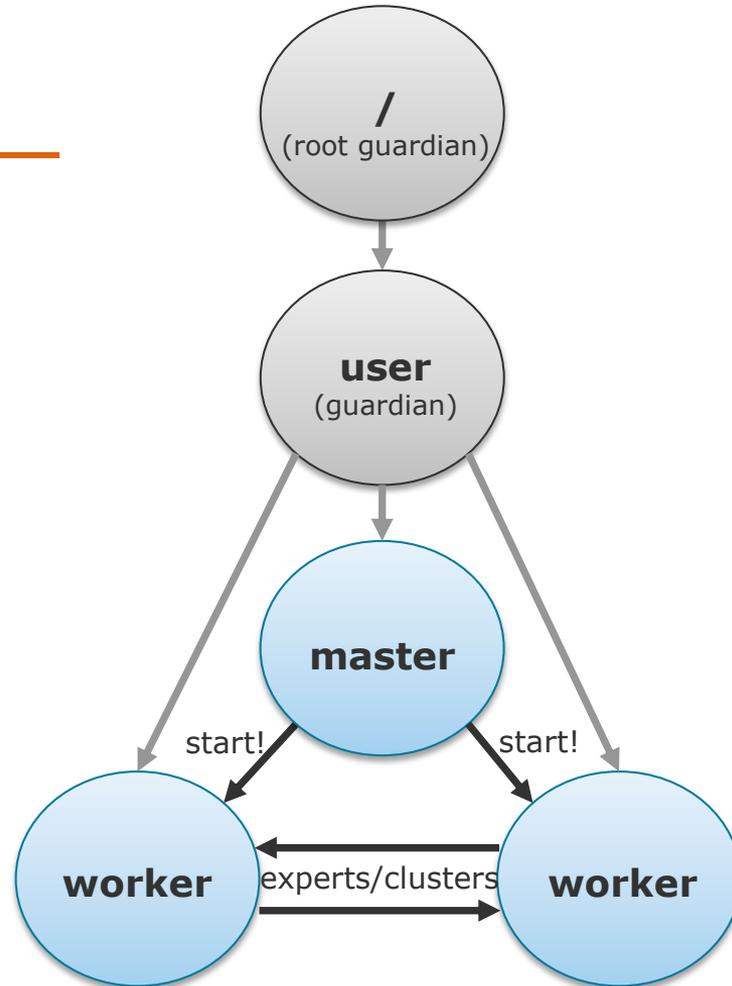
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 98

“The non-reactive workers”

- What could be a problem here?
- Search is a long running job and actors are not interrupted when messages arrive.
- If the workers do not check their inboxes frequently, the inboxes might overflow.
- Due to the unpredictability and burstiness of expert/cluster messages, the inboxes may overflow even if checked frequently.

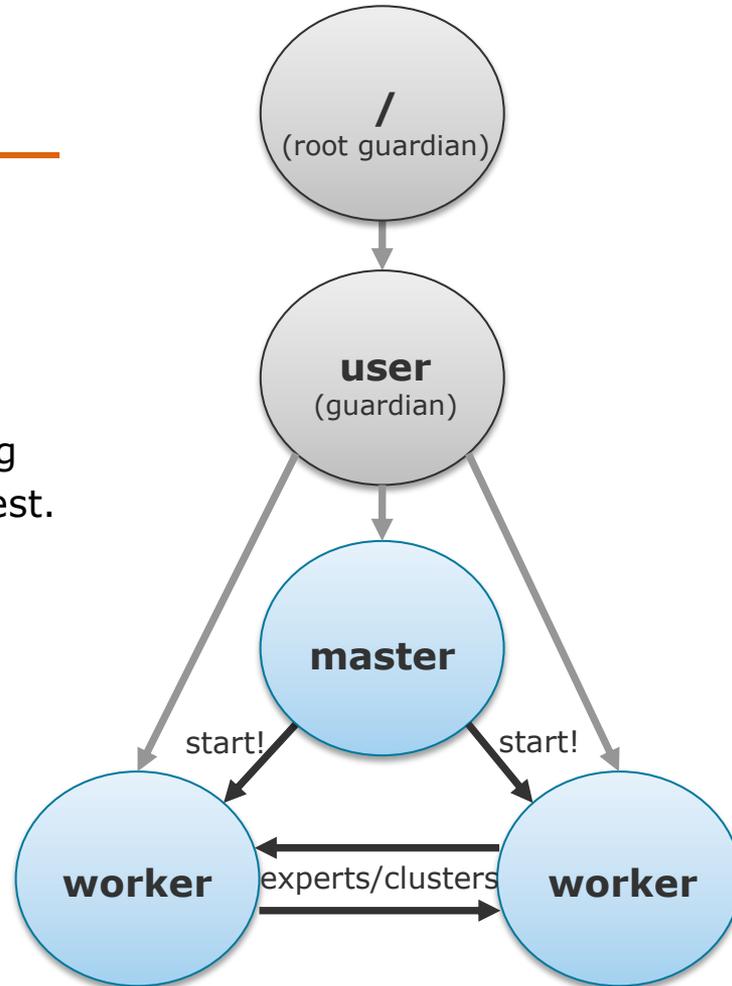


Patterns

Master/Worker

“The non-reactive workers”

- What could be a problem here?
- Solution:
Proxy actors that aggregate incoming messages and deliver them on request.



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 100



**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **101**



Proxy Actor

- Acts as an agent or surrogate for some other actor.
 - Handles a certain (standard) task.
 - Serves to ...
 - externalize behavior/state.
(e.g., prevent cluttering code in real actor)
 - hide the real actor.
(e.g., protect against DOS attacks)
 - handle short-lived concepts.
(e.g., communications)
 - handle resource/time intensive actions.
(e.g., data transfer)
- Other actors “think” they were talking to the real actor!

Example: Simple Proxy

- Delegate a new communication to a proxy.
- If the communication returns a result, the proxy reports it to the real actor.

```
public class MyRealActor extends AbstractActor {  
    @Override  
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(HelloMessage.class, message -> {  
                ActorRef proxy = this.context().actorOf(Proxy.props());  
                this.sender().tell(new HelloBackMessage(), proxy);  
            })  
            .match(ProxyResultMessage.class, this::handle)  
            .build();  
    }  
}
```

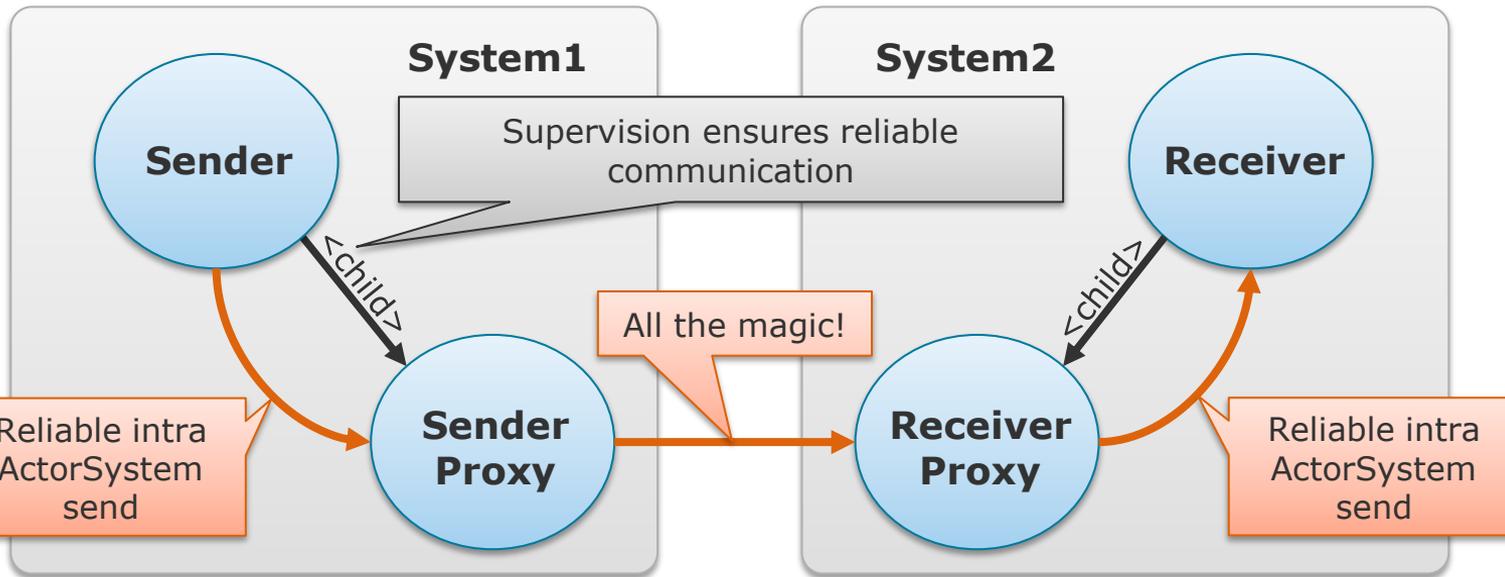
Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **103**

Example: Reliable Communication Proxy

- Provides **exactly-once messaging** on top of at-most-once messaging
- Implements an ACK-RETRY protocol



Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **104**

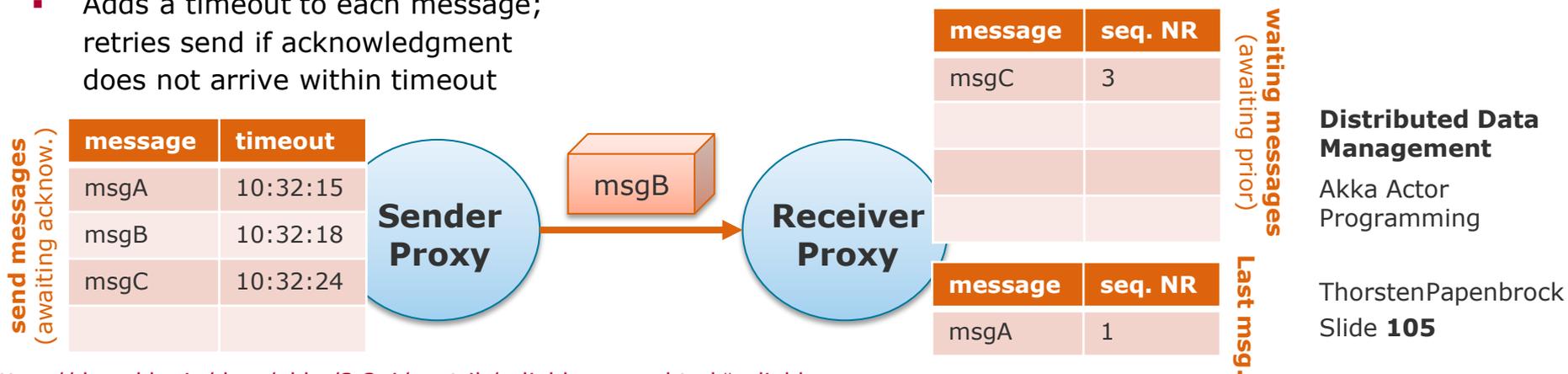
Example: Reliable Communication Proxy

Sender Proxy

- Adds sequence numbers to messages
- Forwards messages to Receiver Proxy
- Stores messages until successfully acknowledged by Receiver Proxy
- Adds a timeout to each message; retries send if acknowledgment does not arrive within timeout

Receiver Proxy

- Acknowledges received messages to Sender Proxy
- Forwards acknowledged messages to Receiver
- Detects missing/duplicate messages by checking sequence number of last forwarded message



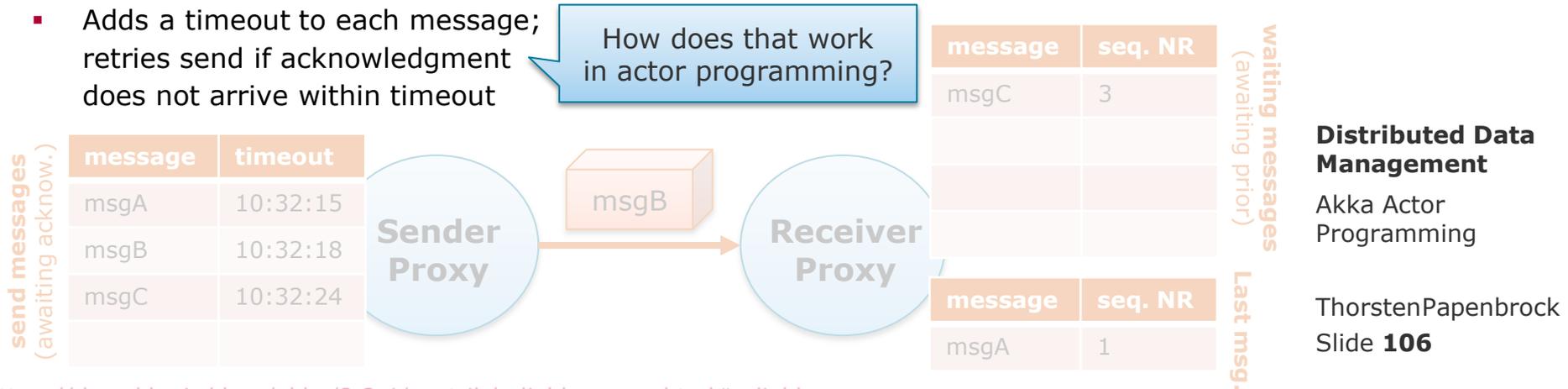
Example: Reliable Communication Proxy

Sender Proxy

- Adds sequence numbers to messages
- Forwards messages to Receiver Proxy
- Stores messages until successfully acknowledged by Receiver Proxy
- Adds a timeout to each message; retries send if acknowledgment does not arrive within timeout

Receiver Proxy

- Acknowledges received messages to Sender Proxy
- Forwards acknowledged messages to Receiver
- Detects missing/duplicate messages by checking sequence number of last forwarded message



Example: Reliable Communication Proxy

- Quick digression: `akka.actor.Scheduler` and `akka.actor.Cancellable`
 - Useful to schedule future and periodic events (e.g. message sends)

```
public class ReceiverProxy extends AbstractActor {  
    [...]  
    // On messageA receive  
    Cancellable sendMessageA = this.getContext().system().scheduler().schedule(  
        Duration.create(0, TimeUnit.SECONDS),  
        Duration.create(3, TimeUnit.SECONDS),  
        receiverProxy, messageA, this.getContext().dispatcher(), null);  
    [...]  
    // On messageA acknowledge  
    sendMessageA.cancel();  
    [...]  
}
```

(Re-)send messageA
to receiverProxy
every 3 seconds

Stop resending
messageA

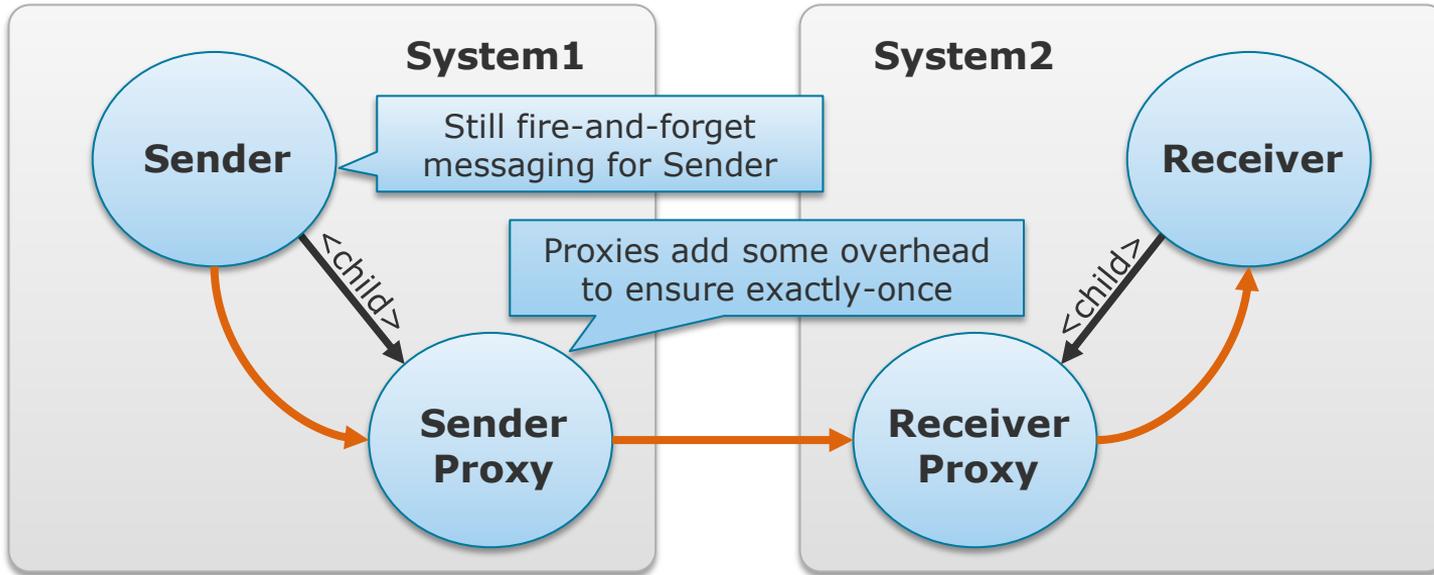
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 107

Example: Reliable Communication Proxy

- Provides **exactly-once messaging** on top of at-most-once messaging
- Implements an ACK-RETRY protocol



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **108**

Patterns
Ask



Patterns

Ask

Tell messaging

- non-blocking, asynchronous, fire-and-forget
 - Java: `someActor.tell(message)`
 - Scala: `someActor ! Message`

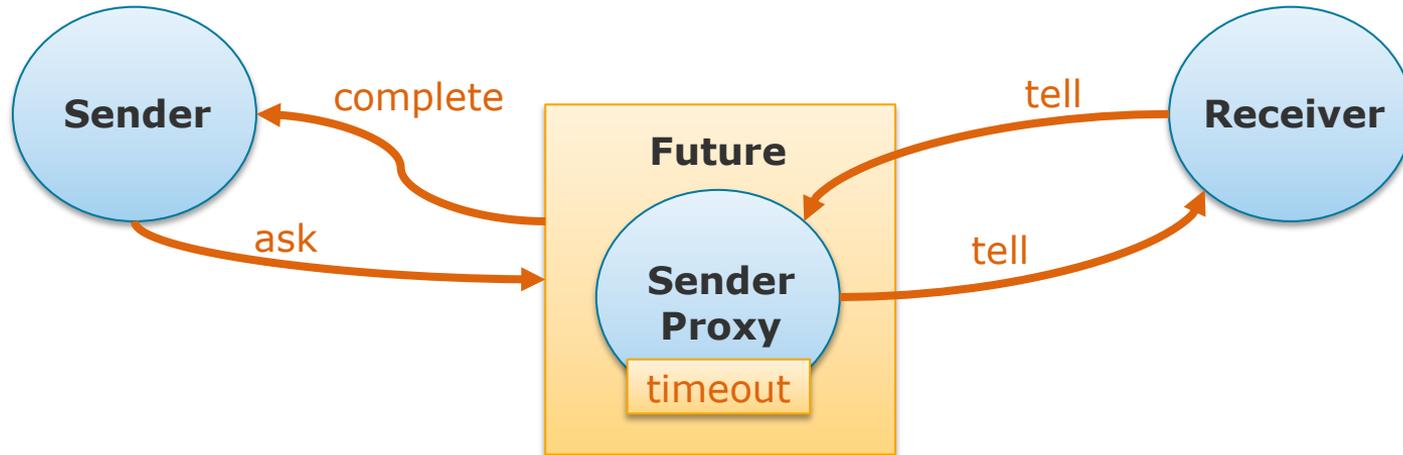
Ask pattern

- blocking, synchronous
 - Java: `someActor.ask(message)`
 - Scala: `someActor ? message`
- Returns a `Future` that the calling entity can wait for
- Implemented in `akka.pattern.PatternsCS.ask` and not a default message send option



Ask pattern

- Via `ask`, the **Sender** creates a `Future` that wraps a **Proxy** actor that tells the message to a **Receiver** with a `timeout` for that message.
- The **Proxy** completes the `Future` either when it receives a response or the `timeout` elapses.



Ask pattern

- Useful if ...
 - the outside, non-actor world needs to communicate with an actor.
 - an actor must not continue working at all until a response is received (very rare case).
- Not a good solution to ...
 - make the communication reliable, i.e., enable exactly-once messaging (use reliable proxy pattern).
 - implement timeouts for message sends (use scheduled tasks).

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **112**

Avoid “ask” if possible!

(the need to ask is usually the result of a bad architecture)

Why to avoid ask:

- **Paradigm violation**
 - Synchronous calls break the strict decoupling of actors and introduce alternative parallelization paradigms (under the hood).
- **Resource blocking**
 - Actively waiting for other actors locks resources (in particular threads) and the Future-construct does require additional threads.
- **Inefficient messaging**
 - Asking requires more effort than telling messages (i.a. for additional threads and timeouts).
- **Error susceptibility**
 - Asking is highly prone to deadlocks (e.g. A asks B while B asks A; both cannot see that they are waiting for each other).

Patterns

Ask

Wait!

We can use **Futures** with the **Pipe-Pattern** in a non-blocking way!

- Example from the official Akka Docu:

<https://doc.akka.io/docs/akka/2.5/actors.html>

```
import static akka.pattern.PatternsCS.ask;
import static akka.pattern.PatternsCS.pipe;

import java.util.concurrent.CompletableFuture;
Timeout t = Timeout.create(Duration.ofSeconds(5));

// using 1000ms timeout
CompletableFuture<Object> future1 =
    ask(actorA, "request", 1000).toCompletableFuture();

// using timeout from above
CompletableFuture<Object> future2 =
    ask(actorB, "another request", t).toCompletableFuture();

CompletableFuture<Result> transformed =
    CompletableFuture.allOf(future1, future2)
        .thenApply(v -> {
            String x = (String) future1.join();
            String s = (String) future2.join();
            return new Result(x, s);
        });

pipe(transformed, system.dispatcher()).to(actorC);
```

Actors vs. Futures (+Pipes)

- Futures are an alternative model for parallelization.
- There are many discussions on “Actors vs. Futures” as means for parallelization control.
 - The question here is more which pattern for parallelization you prefer!
- Actors + Futures is a bad decision, because ...
 - the mix both models makes your **code harder to understand and maintain.**
 - futures should never mutate actor state to avoid **concurrency issues.**
 - futures cannot be passed between actors and, hence, **violate location transparency.**
 - futures make **assumptions about the timing of messages.**
 - **callbacks** are needed to avoid blocking – why this is problematic:
 - Callbacks are executed by non-actor threads on the side (i.e., the callback thread might be completing a Future while the actor that created the Future might process a different message at the same time).
 - Complicates **debugging, parallelization control** and **resource management.**
 - Prone to introduce **shared mutable state** and, hence, to **break encapsulation.**
 - Interferes with **failure handling**, because the asked actor needs to reply with certain, ask-specific error messages that completes the Future.

For instance, Chris Stuccio’s blog:
https://www.chrisstucchio.com/blog/2013/actors_vs_futures.html

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock

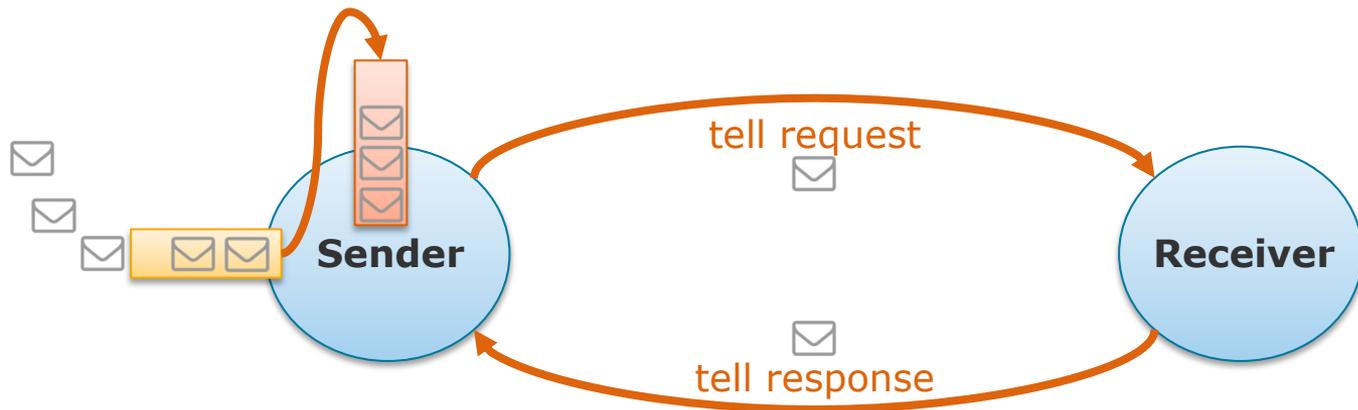
Slide **115**

Patterns

Reactive Waiting

Ok, no ask. So what else can we do to **wait** for a particular response?

1. Create a simple queue in the actor's private state.
2. Append all messages, which are not the awaited message, to this queue.
3. Once the awaited message arrives ...
 1. process the awaited message;
 2. process all messages from the queue in FIFO order;
 3. continue processing messages from the actor's mailbox.

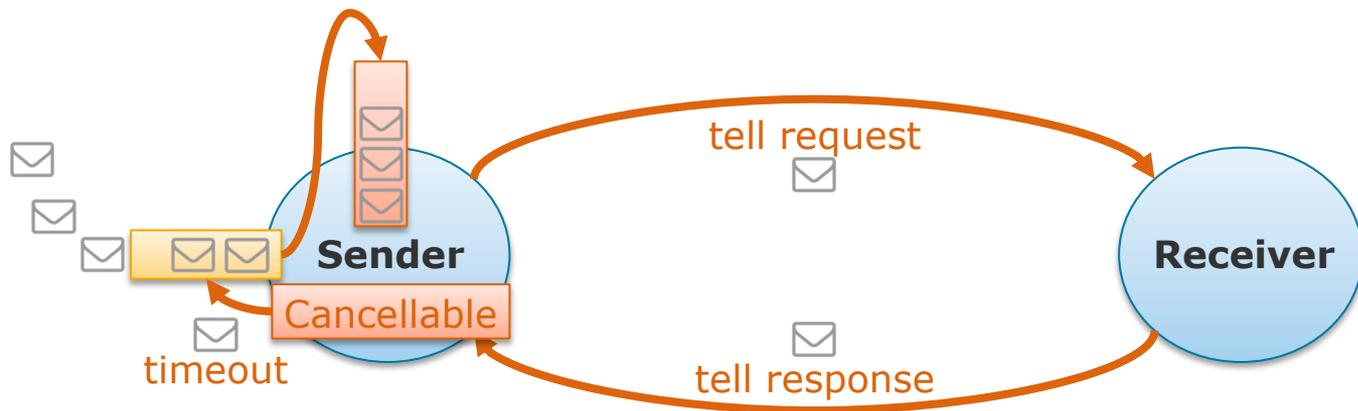


Patterns

Timeouts

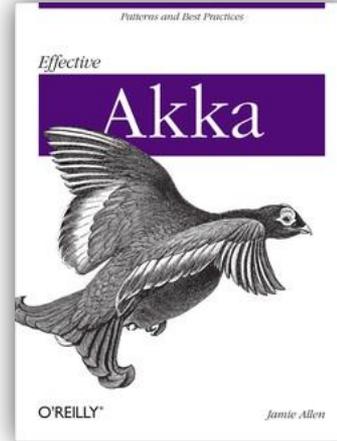
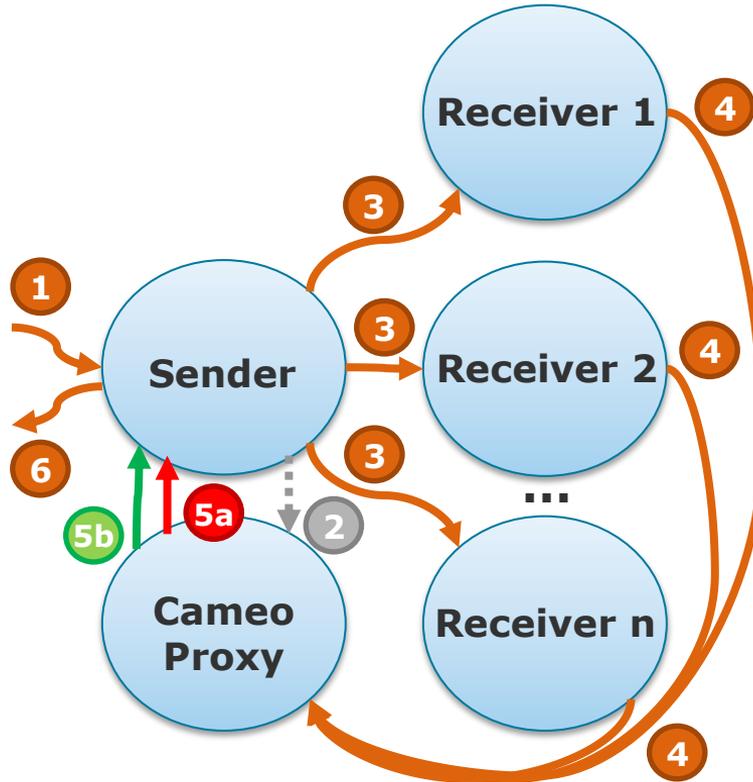
And how to **ensure** that either the result or a "message-failed" status is recorded?

- Create a scheduled Cancellable (with `akka.actor.Scheduler` and `akka.actor.Cancellable`)
- If the Cancellable sends a timeout message, assume the call to have failed.
- If the Receiver sends a response, cancel the Cancellable.



How to **recognize** that a complex task has succeeded or failed?

- The Sender issues a set of coherent subtasks.
- The CameoProxy ...
 - collects multiple responses that belong to the same context;
 - encapsulates waiting and timeout functionality;
 - Ensures that the sender receives the **entire result** or a **failure message**.



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **118**

Patterns

Singleton

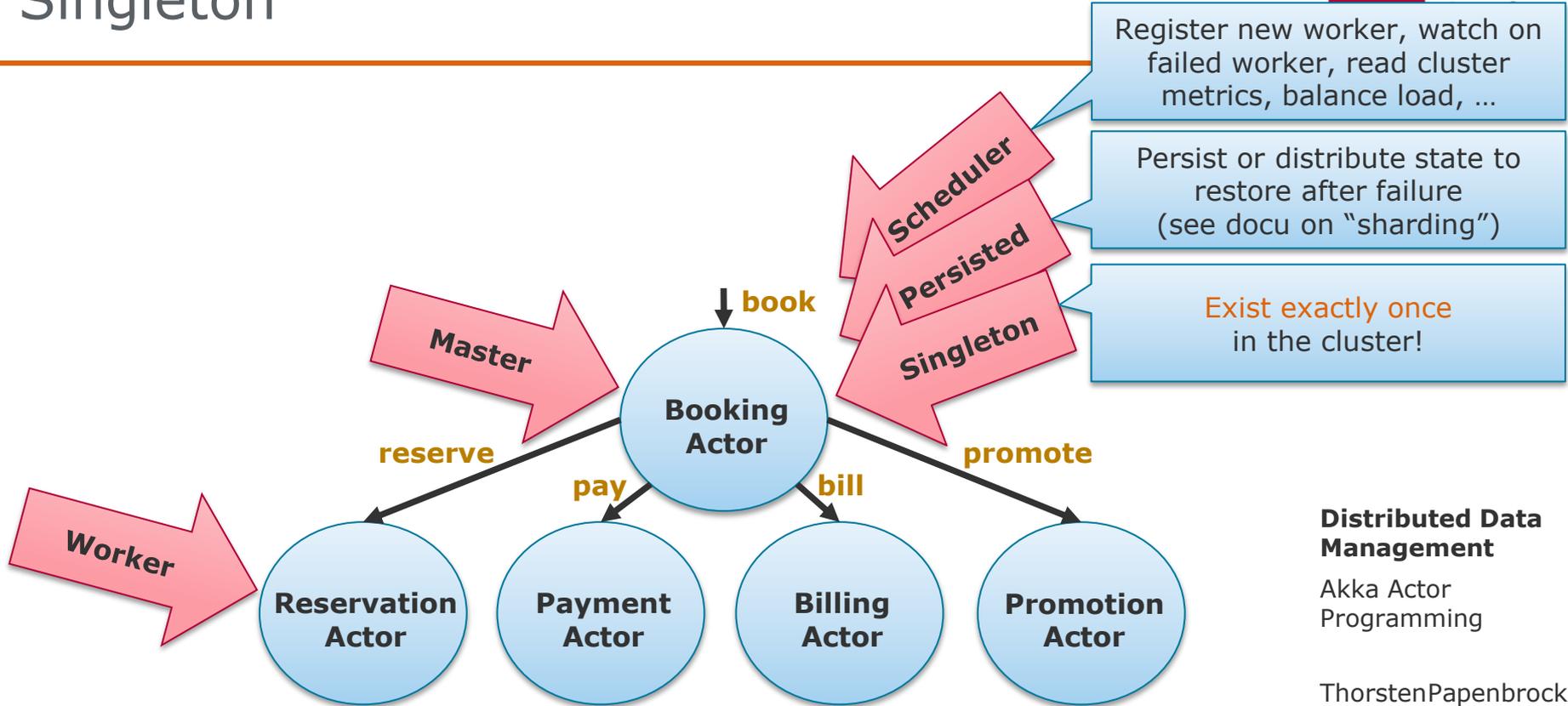


Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **119**

Patterns Singleton



Distributed Data Management

Akka Actor Programming

Motivation

- Sometimes, there needs to be exactly one actor of some type, e.g.,
 - one Endpoint actor for external communication.
 - one Leader actor for consensus enforcement.
 - one Resource actor of some type.

First idea:

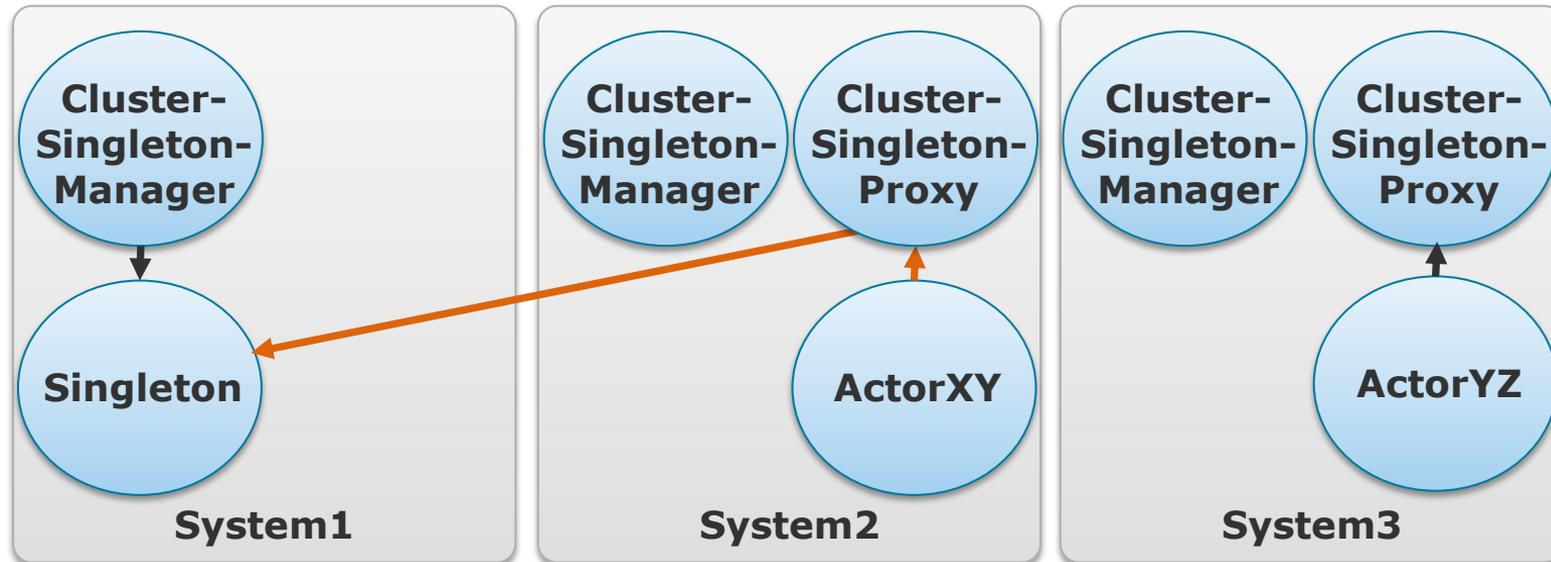
- Simply create that actor once in the cluster.
- Problems:
 1. Requires a **dedicated ActorSystem** that is responsible for creating the singleton.
 2. If that ActorSystem dies, the **singleton is unavailable** until the ActorSystem is back.
 3. Starting the same dedicated ActorSystem twice might cause **split brain**.
 4. Every ActorSystem needs to **know the address** of the singleton.

Patterns

Singleton

Cluster Singleton

- `akka.cluster.singleton.ClusterSingletonManager`
 - Runs in every ActorSystem (start early!)
 - Creates exactly one Singleton actor in the cluster (on the oldest node; singleton moves if node goes down)
- `akka.cluster.singleton.ClusterSingletonProxy`
 - Create one to communicate with the singleton
 - Redirects messages to the current Singleton actor (buffering messages if singleton is temporarily unavailable)



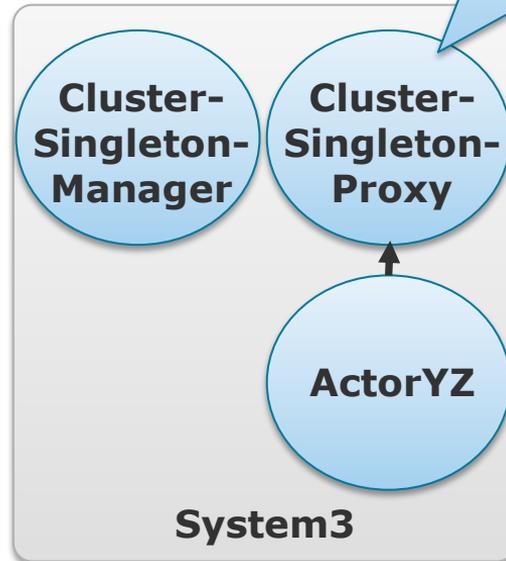
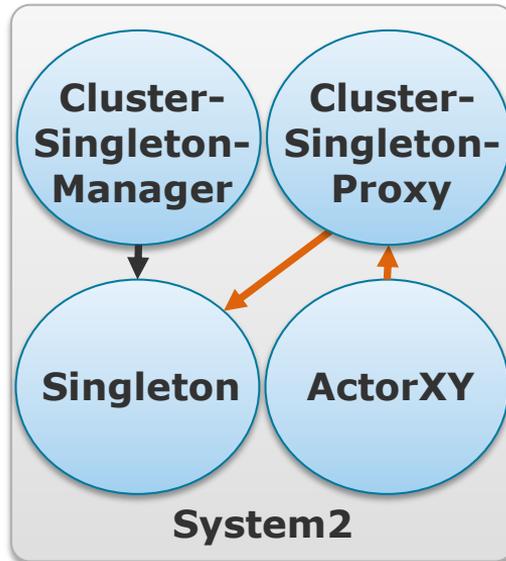
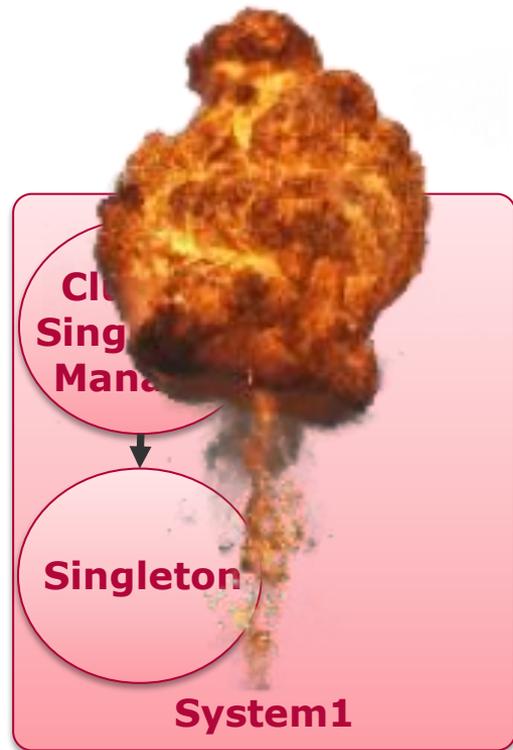
Patterns

Singleton

Cluster Singleton

- `akka.cluster.singleton.ClusterSingletonManager`
 - If an ActorSystem hosting the singleton dies, the singleton is re-created on the then oldest node.
- `akka.cluster.singleton.ClusterSingletonProxy`
 - Knows where the current singleton lives and tracks singleton movements.

Note:
This is no "reliable proxy"
so messages can get lost!



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide 123

Cluster Singleton

```
// On ActorSystem startup
ActorRef manager = system.actorOf(
  ClusterSingletonManager.props(
    LeaderActor.props(),
    PoisonPill.class,
    ClusterSingletonManagerSettings.create(system).withRole("master")),
  "leaderManager");

// If an actor needs to talk to the singleton
ActorRef proxy = system.actorOf(
  ClusterSingletonProxy.props(
    "/user/leaderManager",
    ClusterSingletonProxySettings.create(system).withRole("master")),
  "leaderProxy");

proxy.tell(new HelloLeaderMessage(), this.self());
```

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **124**



**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **125**

Application Shutdown?

Task vs. Actor shutdown

- Tasks finish and vanish.
- Actors finish and wait for more work.
 - Actors need to be **notified** to stop working.

How to detect that an application has finished?

- **All mailboxes empty?**
 - No: Actors might still be working on messages (and produce new ones).
- All mailboxes empty **and all actors idle?**
 - No: Messages can still be transferred, i.e., on the network.
- All mailboxes empty and all actors idle **for “a longer time”?**
 - No: Actors might be idle for “longer times” if they wait for resources.
- **Only the application knows when it is done** (e.g. a final result was produced).



Application Shutdown?

Problem

- ActorSystems stay alive when the main application thread ends.

Forced Shutdown

- Kill the JVM process.
- Problems:
 - Risk of resource corruption (e.g. corrupted file if actor was writing to it)
 - Many, distributed JVM processes that need to be killed individually

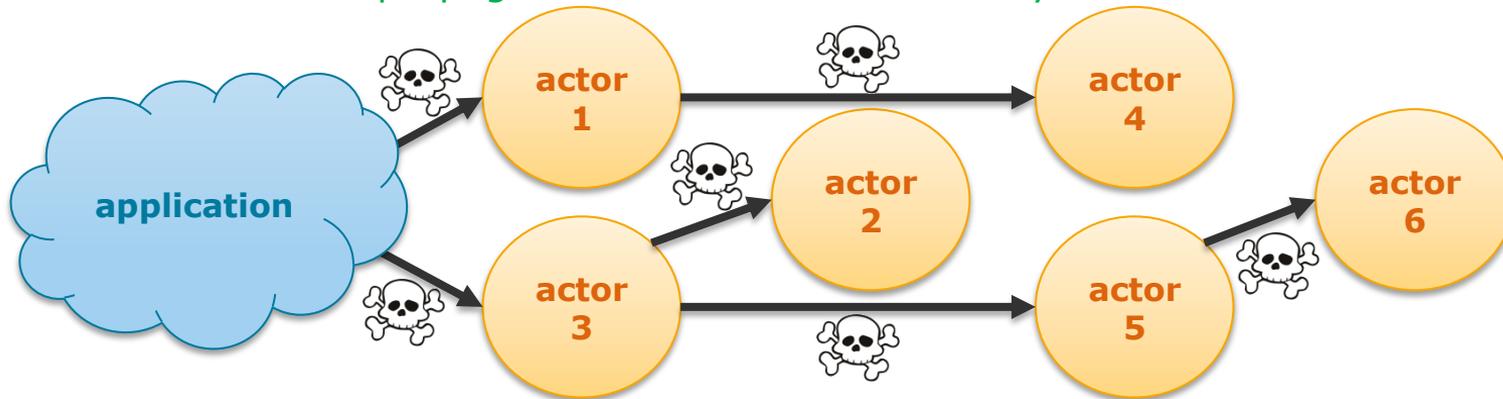
Actor System terminate()

- Calling terminate() on an ActorSystem will stop() all its actors.
- Problem:
 - ActorSystems on remote nodes are still alive!

PoisonPill Shutdown

- If an application is done, **send a PoisonPill message to all actors.**
- Actors automatically **forward the PoisonPill** to all children.
- The PoisonPill finally **stops an actor.**
- Advantages:
 - Pending messages prior to the PoisonPill are properly processed.
 - PoisonPill propagates into all remote Actor Systems.

Use `postStop()` to also forward a PoisonPill to other actors



PoisonPill Shutdown

- If an application is done, **send a PoisonPill message to all actors.**
- Actors automatically **forward the PoisonPill** to all children.
- The PoisonPill finally **stops an actor.**
- Advantages:
 - Pending messages prior to the PoisonPill are properly processed.
 - PoisonPill propagates into all remote Actor Systems.

Use `postStop()` to also forward a PoisonPill to other actors

```
import akka.actor.PoisonPill;
```

```
[...]
```

```
this.otherActor.tell(PoisonPill.getInstance(), ActorRef.noSender());
```

PoisonPill is an Akka message that is handled by all actors

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **129**

Patterns

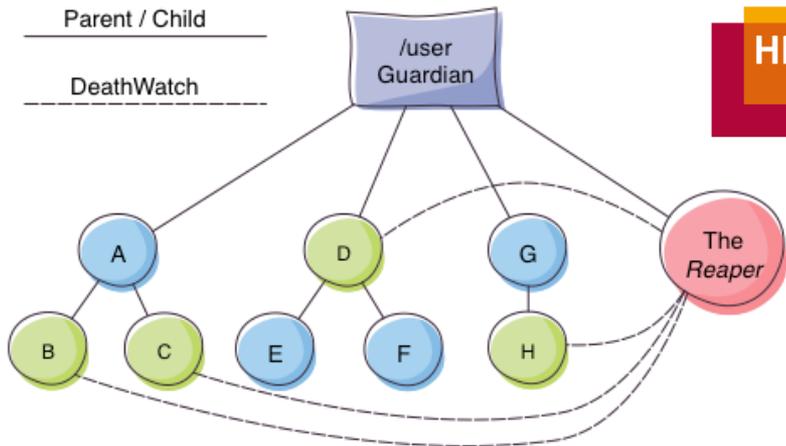
Reaper

PoisonPill Shutdown

- Problem:
 - If all actors are stopped, the Actor System is still running!
- Solution:
 - Reaper Pattern**

Reaper

- A dedicated actor that “knows” all actors
 - “Reaps actor souls and ends the world!”
- Listens to death-events (Termination events)
- Call the `terminate()` function on the Actor System if all actors have stopped (e.g. due to PoisonPills)



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **130**

```
public class Reaper extends AbstractLoggingActor {
```

```
    public static class WatchMeMessage implements Serializable { }
```

```
    public static void watchWithDefaultReaper(AbstractActor actor) {  
        ActorSelection reaper = actor.context().system().actorSelection("/user/reaper");  
        reaper.tell(new WatchMeMessage(), actor.self());  
    }
```

After creation, every actor needs to register at its local reaper

```
    private final Set<ActorRef> watchees = new HashSet<>();
```

```
    @Override
```

```
    public Receive createReceive() {  
        return receiveBuilder()  
            .match(WatchMeMessage.class, message -> {  
                if (this.watchees.add(this.sender()))  
                    this.context().watch(this.sender());  
            })  
            .match(Terminated.class, message -> {  
                this.watchees.remove(this.sender());  
                if (this.watchees.isEmpty())  
                    this.context().system().terminate();  
            })  
            .matchAny(object -> this.log().error("Unknown message"))  
            .build();  
    }  
}
```

Watch a new actor

Witness actor dying

End the world



Reasons to die without a PoisonPill

- If **an actor's parent dies**, the orphaned actor dies too.
- If **a client loses its master** Actor System, it might decide to die.
- If **an error occurs**, the supervisor might choose to let the failing actor die.



Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **132**

Stop a running system

- What if the system operates an endless stream of jobs and should be stopped?
 - Send a **custom termination** message.
 - Upon receiving this termination message, an actor should ...
 1. **refuse** all incoming new jobs.
 2. **finish** all current jobs (i.e., wait for other actors that work on it).
 3. **let** child actors finish their jobs.
 4. **stop** child actors.
 5. **stop** itself.



**Distributed Data
Management**

Akka Actor
Programming

ThorstenPapenbrock
Slide **133**

Patterns

Further Reading

Akka documentation

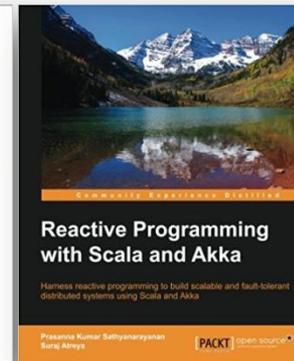
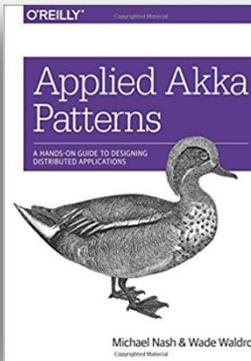
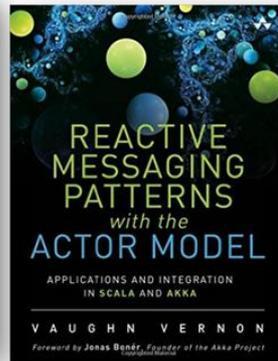
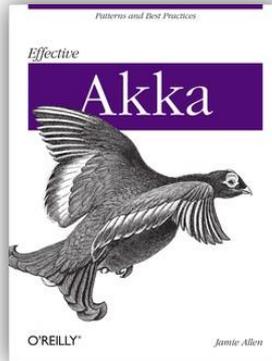
- <http://doc.akka.io/docs/akka/current/java/index.html>
- <http://doc.akka.io/docs/akka/current/scala/index.html>

Example code @ GitHub:
<https://github.com/akka/akka-samples>

Experiences, best practices, and patterns

- <http://letitcrash.com>
- <http://akka.io/blog>
- <https://github.com/sksamuel/akka-patterns>

Akka actor programming literature:



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **134**



akka-sample-camel-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-camel-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-cluster-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-cluster-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-distributed-data-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-distributed-data-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-fsm-java	DinningHakkersTyped - dinning hakkers implementation with Akka Typed ...	3 days ago
akka-sample-fsm-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-main-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-main-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-multi-node-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-osgi-dining-hakkers	Akka 2.5.16 (#73)	a month ago
akka-sample-persistence-dc-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-persistence-dc-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-persistence-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-persistence-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-sharding-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-sharding-scala	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-supervision-java	Upgrade to 2.5.17 (#75)	14 days ago
akka-sample-vavr	Akka 2.5.16 (#73)	a month ago

Example code @ GitHub:
<https://github.com/akka/akka-samples>

Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
 Slide **135**

Measure, log, record and test!

- Distributed systems are difficult to debug and profile:
No stop-the-world breakpoints, limited use of JVisualVM, ...
- Solutions:
 - **Actor hierarchies** for failure debugging
 - **Metrics collection** for runtime profiling (see `ClusterMetricsExtension`)
 - **Logging** of important messages and events (see `LoggingAdapter`)
 - **Unit tests** for actors and code modules (see Akka's `TestKit`)

Distributed Data Management

Akka Actor
Programming

ThorstenPapenbrock
Slide **136**

Avoid premature optimization!

- Start simple and test correctness first.
- Then, carefully identify bottlenecks and where necessary:
 - Add concurrency e.g. via schedulers and further child actors.
 - Turn immutable into mutable data structures.

Be explicit!

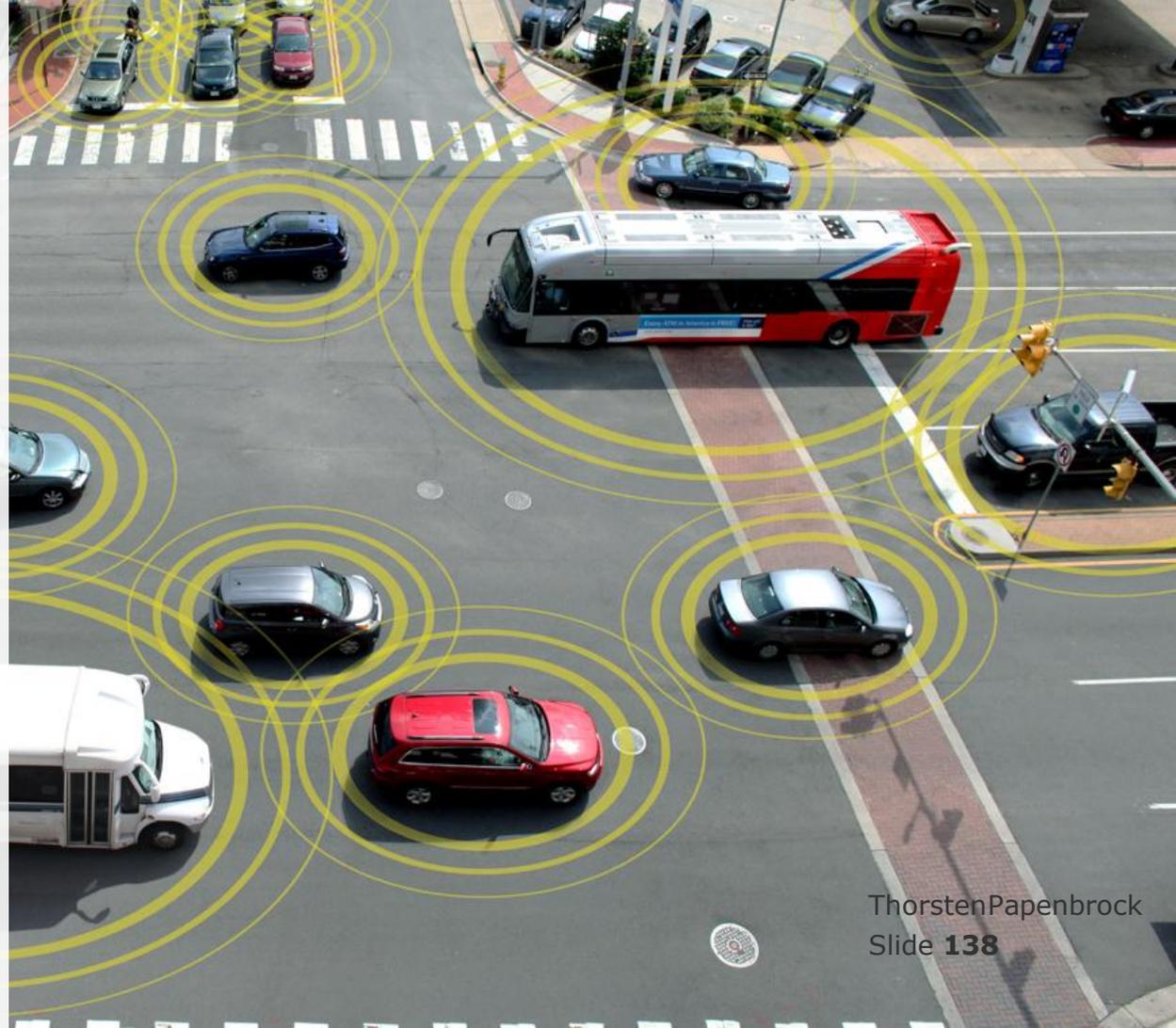
- Name actors and ActorSystem instances (to find them by name).
- Create specialized message types (don't match on Strings, Ints, ...).
- Create specialized exceptions and SupervisorStrategies for them.

Avoid mutable states in messages!

- Check that the payload of a message is always immutable (or at least treated as if it was immutable).
- E.g. sending mutable Java Maps lets the target actor see mutable state!

Akka Actor Programming Hands-on

- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- **Homework**





HPI-Information-Systems / akka-tutorial

Notifications

Star

12

Fork

21

- Code
- Issues
- Pull requests 3
- Actions
- Projects
- Wiki
- Security
- Insights

master

4 branches 0 tags

Go to file

Code

About

Code for the Akka tutorial

Readme

Apache-2.0 License

Releases

No releases published

Packages

No packages published

Contributors 6

thorsten-papenbrock Updated dependency versions for akka-tutorial and o... 904c8b1 3 minutes ago 62 commits

akka-tutorial Updated dependency versions for akka-tutorial and octopus 3 minutes ago

ddm-exercise Merge commit 12 minutes ago

octopus Updated dependency versions for akka-tutorial and octopus 3 minutes ago

.gitignore Merged the Imp and pc tasks into one project, updated the librari... 23 minutes ago

LICENSE Added the octopus project to the repository. 3 years ago

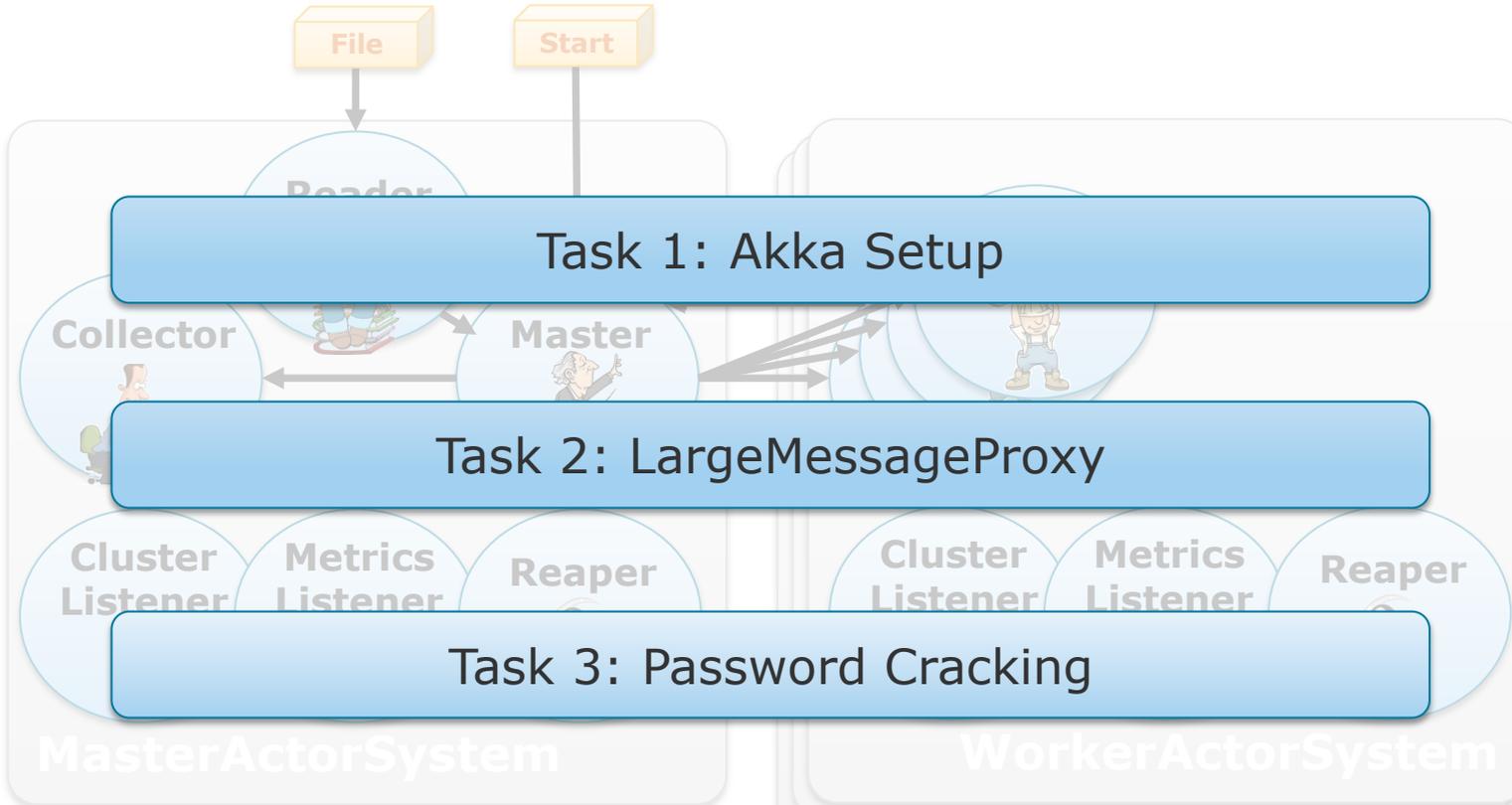
README.md Merged the Imp and pc tasks into one project, updated the librari... 23 minutes ago

README.md

Akka tutorial

Homework

ddm-exercise



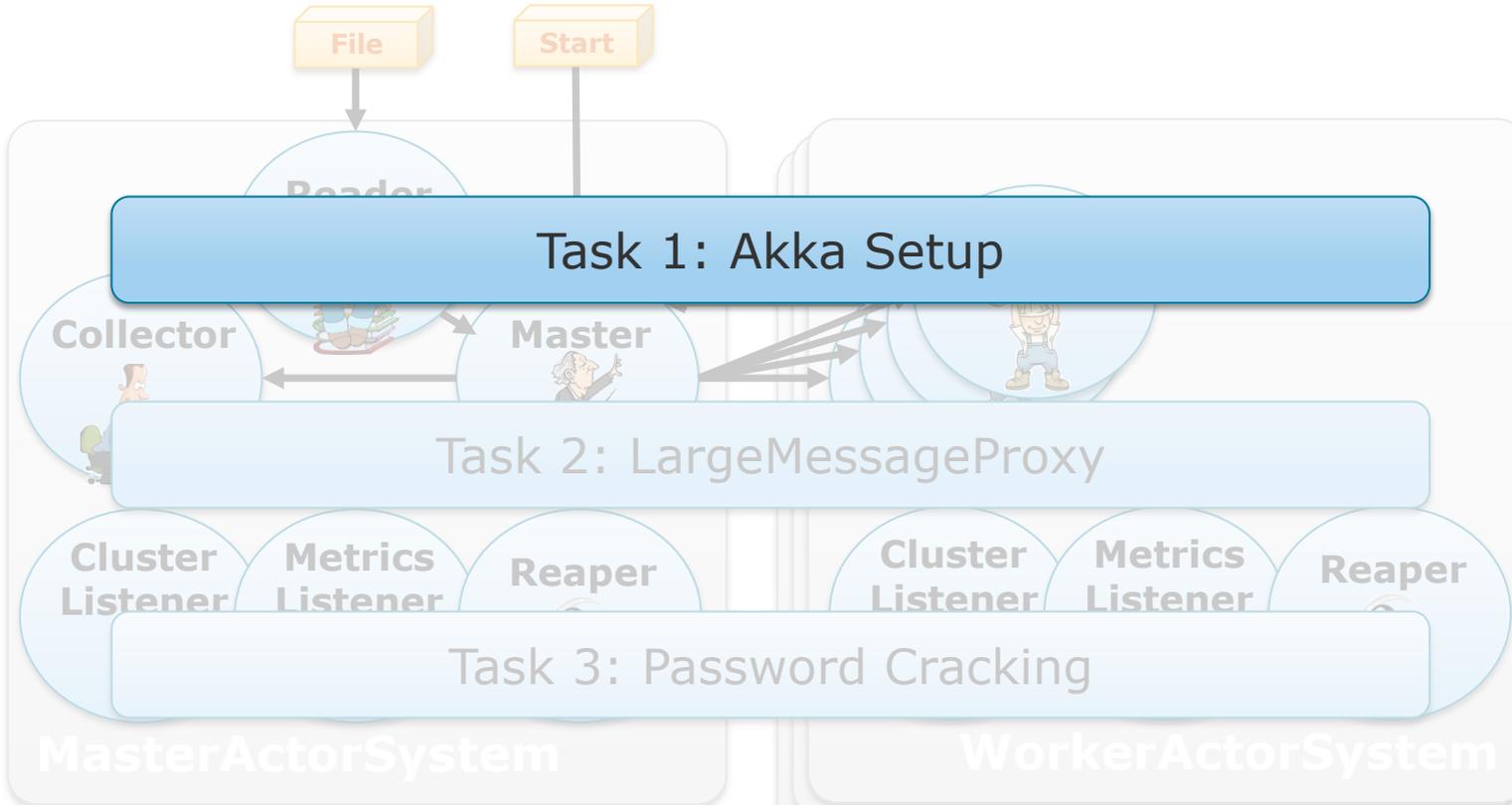
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **140**

Homework

ddm-exercise



Distributed Data Management

Akka Actor Programming

Task 1 – Akka Setup

1. Form teams of **two** students.
2. Create a **public** GitHub repository.
3. Copy or fork the ddm-exercise project from the exercise repository <https://github.com/HPI-Information-Systems/akka-tutorial> into your repository.
4. **Build**, **understand** and **test** the ddm-exercise project.
5. Optional: Check out and play with the akka-tutorial and octopus projects.
6. Send your **first and last names**, a **group name** and the **link of your repository** via email to: thorsten.papenbrock@hpi.de

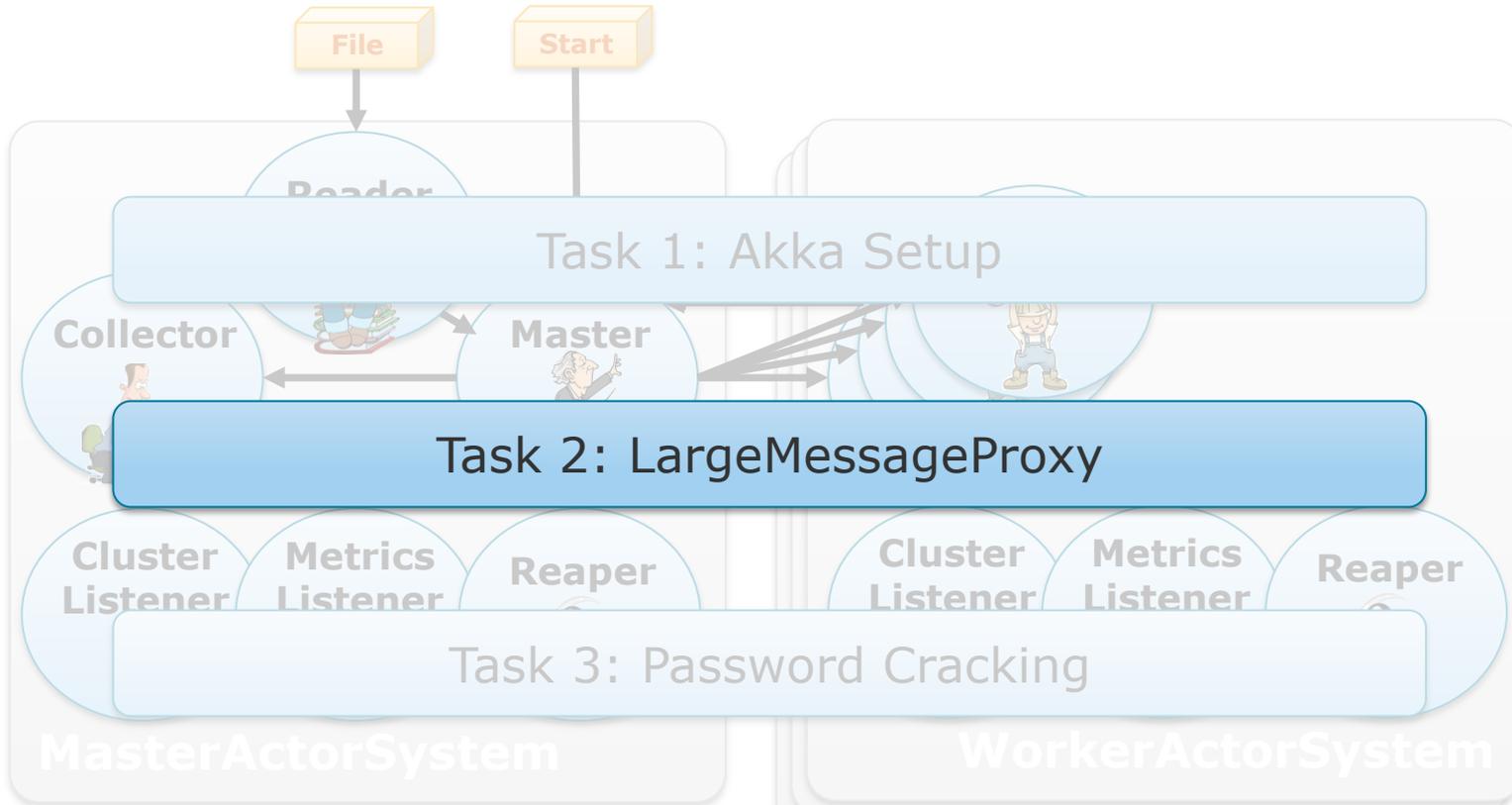
Task 1 – Akka Setup

Submission

- **Deadline**
 - 15.05.2021 21:00:00 (next Saturday!)
- **Artifacts**
 1. Email
 - Header: DDM Team <groupname>
 - Body:
<firstname1> <lastname1>
<firstname2> <lastname2>
<groupname>
<GitHub-URL>

Homework

ddm-exercise

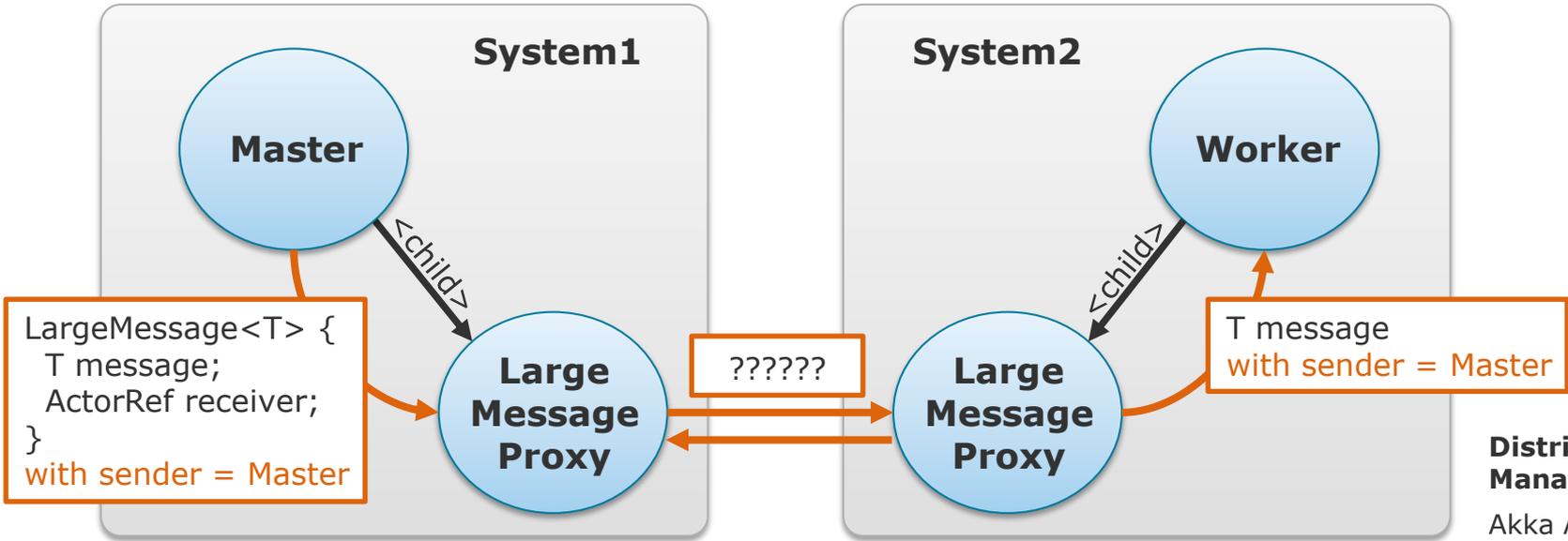


Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **144**

Task 2 – LargeMessageProxy



Distributed Data Management

Akka Actor Programming

Task

- Implement the LargeMessageProxy actor!

```

@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(LargeMessage.class, this::handle)
        .match(ByteMessage.class, this::handle)
        .matchAny(object -> this.log().info("Received unknown message: \"{}\"", object.toString()))
        .build();
}

private void handle(LargeMessage<?> largeMessage) {
    Object message = largeMessage.getMessage();
    ActorRef sender = this.sender();
    ActorRef receiver = largeMessage.getReceiver();
    ActorSelection receiverProxy = this.context().actorSelection(receiver.path().child(DEFAULT_NAME));

    // TODO: Implement a protocol that transmits the potentially very large message object.
    // The following code sends the entire message wrapped in a ByteMessage, which will definitely fail in a distributed setting if the message is large!
    // Solution options:
    // a) Split the message into smaller batches of fixed size and send the batches via ...
    //   a.a) self-build send-and-ack protocol (see Master/Worker pull propagation), or
    //   a.b) Akka streaming using the streams build-in backpressure mechanisms.
    // b) Send the entire message via Akka's http client-server component.
    // c) Other ideas ...
    // Hints for splitting:
    // - To split an object, serialize it into a byte array and then send the byte array range-by-range (tip: try "KryoPoolSingleton.get()").
    // - If you serialize a message manually and send it, it will, of course, be serialized again by Akka's message passing subsystem.
    // - But: Good, language-dependent serializers (such as kryo) are aware of byte arrays so that their serialization is very effective w.r.t.
    //   serialization time and size of serialized data.
    receiverProxy.tell(new ByteMessage<>(message, sender, receiver), this.self());
}

private void handle(ByteMessage<?> message) {
    // TODO: With option a): Store the message, ask for the next chunk and, if all chunks are present, reassemble the message's content, deserialize it and
    //   pass it to the receiver.
    // The following code assumes that the transmitted bytes are the original message, which they shouldn't be in your proper implementation ;-)
    message.getReceiver().tell(message.getBytes(), message.getSender());
}

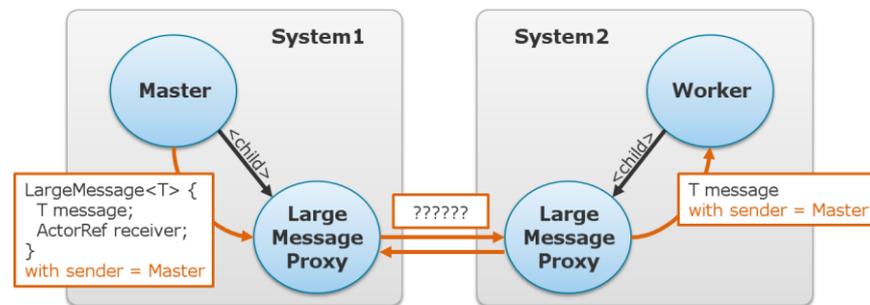
```

Homework

Task 2 – LargeMessageProxy

Rules

- Do not mess with the time measurement: It should start with the registration time and it should end when receiving the data.
- Do not change the command line interface or app name; otherwise, the automatic test scripts will fail.
- Do not change the LargeMessage class; the LargeMessageProxy should be able to send messages of any type T.
- Use maven to import additional libraries if you need some.
- Do not use the disk.
- Feel free to change everything inside the LargeMessageProxy!



Distributed Data Management

Akka Actor Programming

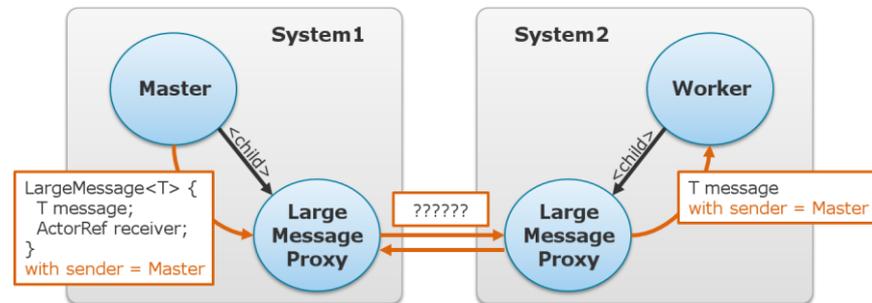
ThorstenPapenbrock
Slide 147

Homework

Task 2 – LargeMessageProxy

Submission

- **Deadline**
 - 28.05.2021 23:00:00
- **Artifacts** (in GitHub repository)
 1. Source code
 2. "assignment2" folder with ...
 1. a jar file of your algorithm;
 2. a pdf or ppt slide describing your solution.



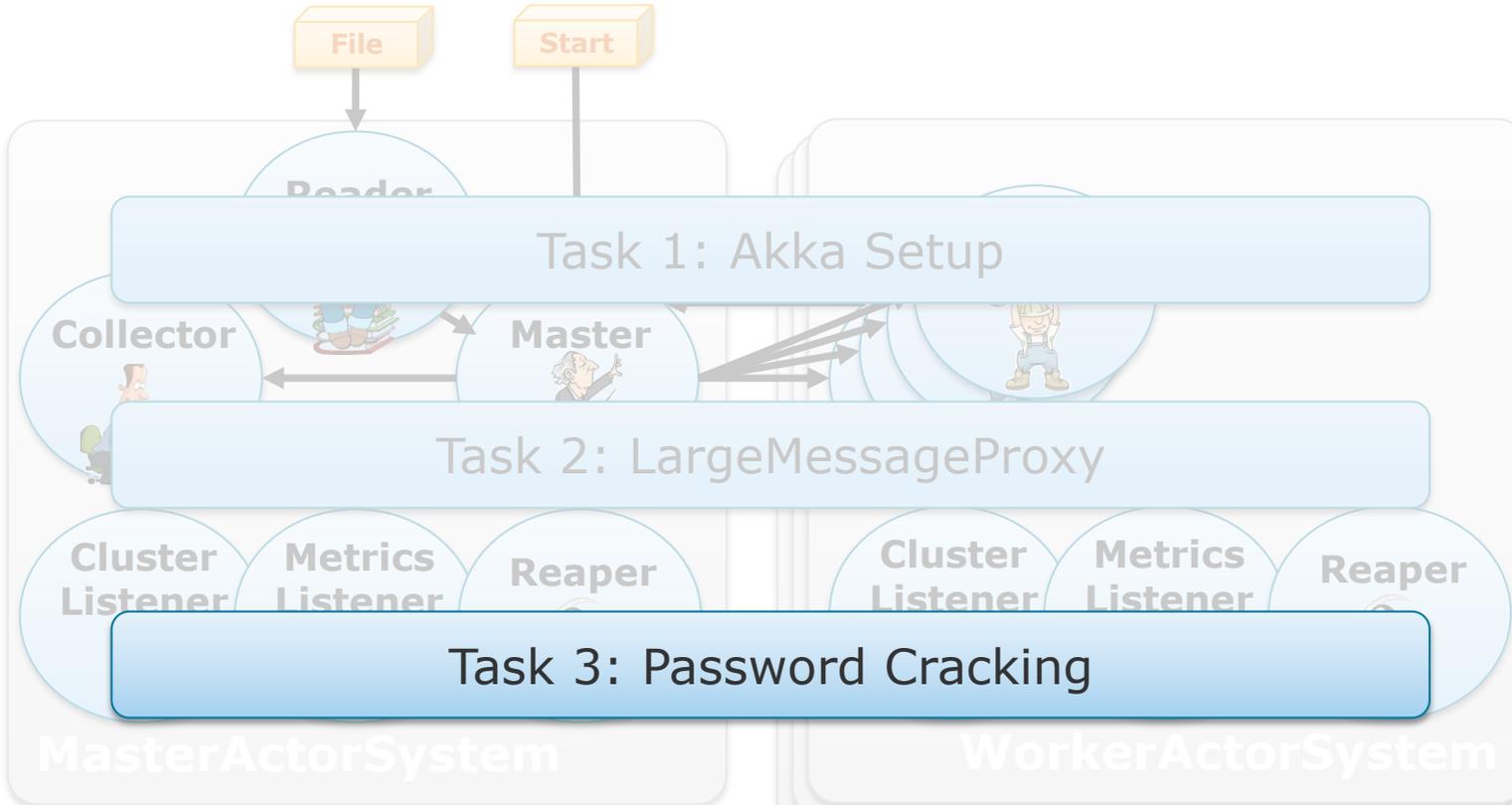
Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **148**

Homework

ddm-exercise



Distributed Data Management

Akka Actor Programming

ThorstenPapenbrock
Slide **149**

Task 3 – Password Cracking

ID	Name	PasswordChars	PasswordLength	Password	Hint1	Hint2	Hint3	Hint4	Hint5	Hint6	Hint7	
1	Sophia	ABCDEFHGHIJK	10	GGGFGFFFFG	HJKGDEFBIC	FCJADEKGHI	FAJBIDIEKGH	AGCJEHFKIB	BHKICGFADJ	JIFAGKDBCE	GAHDKJBCEF	
2	Jackson	ABCDEFHGHIJK	10	EEEE		AEHJIDGFKC	IDAHFGEKBJ	EHFIJKBGAC	HFJIEDACBK	FJKIDJCEAB	KDHGCAEJFB	
3	Olivia	ABCDEFHGHIJK	10	KDD		CAKEIFHGJD	JBFEDHIKAG	IDAKGHBFC	KGBAEICHDJ	DKHFBEJIAC	EABJGFIKDC	
4	Liam	ABCDEFHGHIJK	10	CCCCGGCC		FAICGJDHEK	CHBKIGEJAF	AICDKGHJBF	EDACKBJHIC	JDKIFACEGB	BGKJDAHCFE	
5	Emma	ABCDEFHGHIJK	10	BDDBDDBDD		EGICDFKHB	HEAJBDGFK	BAHCKDFIJG	HBEDKAGCI	IBHCEFJADK	FAGDEJICKB	GFHEAKCDBJ
6	Noah	ABCDEFHGHIJK	10	GHGGHGGHHH		CFKBJGDIEH	CAIGHEJFDK	GJBEKIADFH	AIBCGHGEKF	GDIBCKFHJA	CGJHDEAIBK	DGKFBACJH
7	Ava	ABCDEFHGHIJK	10	DEEFDFFD		FHIKEBGDJC	KHFICAJGED	KIAHDFEJGB	CGFAKIBDHI	ACEHFKBIJ	GBADIJEKFC	AFCKGHBDJE
8	Aiden	ABCDEFHGHIJK	10	HHIHI		GCIFEHDKBJ	JDHIEGKACF	FJHBEGAKDI	AIBJEHKGFC	CGJAFBIHDK	JECAIDGHBK	IJBDCKEAFH
9	Isabella	ABCDEFHGHIJK	10	CJCJCJC		EHDGCIKBJF	IFJCAEHGKD	AFBHEGKIJC	KGFBIADJHC	JKAGEDHIBC	CBKIDEAHFJ	CJAFKEIBDG
10	Lucas	ABCDEFHGHIJK	10	BBCBCCC		KGJHIDECFB	BHFACKEGIJ	ICJGHFKBAD	KEICHGAJDB	BCDKEJIFAH	IDGEBAJKCF	FCBDKGHJAE
11	Mia	ABCDEFHGHIJK	10	DDDDDDDD		FIB						BKIGAEDFCJ
12	Caden	ABCDEFHGHIJK	10	DDDAADDDDD		AEI						EKFGAJCBHD
13	Aria	ABCDEFHGHIJK	10	CCCCCFCC		GKH						HBJEAFCGK
14	Grayson	ABCDEFHGHIJK	10	DDJJJBBJJ		GEH						BKGEJFADCK
15	Riley	ABCDEFHGHIJK	10	BGGGBBB		GHE						JBCHAGEKDF
16	Mason	ABCDEFHGHIJK	10	AAAJAJA		EKJ						BCGEFAKJID
17	Zoe	ABCDEFHGHIJK	10	JJJJJJJ		JHC						CBEFIDGKAI
18	Elijah	ABCDEFHGHIJK	10	EJJEJJEE		KDC						FJBCGADIEKF
19	Amelia	ABCDEFHGHIJK	10	GDDGGGDGDD		GCI						HEGDCAKJFB
20	Logan	ABCDEFHGHIJK	10	FFEEEEFFF		KHE						JFHDCKEAGB
21	Layla	ABCDEFHGHIJK	10	CCCHCCHCCC		GIF						JBGAHFDCFK
22	Oliver	ABCDEFHGHIJK	10	ABBBAABAAA		AFK						FJGKHEADBC
23	Charlotte	ABCDEFHGHIJK	10	BGBGBBBBG		ECK						GJBECKAFHD
24	Ethan	ABCDEFHGHIJK	10	HHBBHHH		DHJ						BCHJEAKFDD
25	Aubrey	ABCDEFHGHIJK	10	EJJEJE		HDFEJBKICG	IFJCAEHGKD	KCEFBAGI	IDCJBAGEHK	IKHJCDBAEF	JCGIFDBEAK	
26	Jayden	ABCDEFHGHIJK	10	CCCCGGGG		DJEHCBKFI	FCJADEKGHI	ECFKBIAHJG	GDCIFKBJAH	HJBGAIKCED	DICKFBGAJE	GFDCCKBAHJE
27	Lily	ABCDEFHGHIJK	10	DHHDHDD		KCJFIBHEDG	FJDKCAIEGH	DABGJEFKIH	DCGIKHFABJ	KDGBEIHACJ	IDCKDFHJAE	EHADCKFCJG
28	Muham	ABCDEFHGHIJK	10	CBCBBBC		EDFGHKIBJC	HEICAKBJFG	CABIDFGHKJ	DHAKICBGJE	IHAJCEBFD	CEABFJGKID	ABGKFDHCEJ
29	Chloe	ABCDEFHGHIJK	10	CEECECCCEC		KEGDHFCBIJ	GCIIAEDKFJ	HFGKIBACEJ	CJHGKBDIAE	FECIBJKADH	GAFCIBEKJD	CJBAKEGDFH
30	Carter	ABCDEFHGHIJK	10	BBIIBIIBI		CKFGBIHDEJ	EJDKIHGABF	ECHIJGFBAK	AFHIBCKGDJ	IHCBKGEJAD	AFHIDJKBCE	IBGCJKFAED
31	Harper	ABCDEFHGHIJK	10	EAAAEAEAE		IAFKCHJGDE	AKFDJIHGBE	BGECFIJKAH	BJDAIGKEHC	IHEBKACJDF	BJDIGAKFCE	EFADJKCBGH
32	Michael	ABCDEFHGHIJK	10	DCDDCCDDDD		JGFEICBKHD	CKJDHGIEAF	KIGDHABJCF	GDEIHACBJK	BICEAFDHKJ	JGFCKBDEAI	HCBGDKFAJE
33	Evelyn	ABCDEFHGHIJK	10	IICICICCC		CDBJHIEGKF	FIKGEHCAJD	BIJAGEKFHC	CADBIHFJKG	GHJDBKAEIC	KJEIDHABC	ACBGKFDJIE
34	Sebastian	ABCDEFHGHIJK	10	IIIIIIIII		EDFKHGJBCI	KFHIDJGACE	KHAIDGJBEF	CIABGJKEHF	JKEIDCBAHG	KIDFJABHEC	FDBGJCAEIK
35	Adalyn	ABCDEFHGHIJK	10	JJJJDJDDJJ		IKEJFCBHDG	AHDCRFEGJI	DBIKAHJGFE	DKAIHGFCJB	CABKHEDJIG	ABJEFCHDKI	GJAFEBCKDI
36	Alexander	ABCDEFHGHIJK	10	HKHKHKHKHK		CEBHKDFIJG	FGJADKCIEH	AGDIFHKJBE	FHCEKABGIJ	BFJAIDCGHK	BCAHEJDGIK	JFCDIHKAE

Passwords to be cracked

All characters that may appear in the password

Number of characters in the password

These two fields have always the same value for all records.

Hints:

- Every hint contains all PasswordChars besides one char, i.e., $|\text{Hint}| = |\text{PasswordChars}| - 1$
- The missing char is the hint, because it does not appear in the password.
- The number of hints can change!
- The more hints we have, the easier it is to find the password.

Task 3 – Password Cracking

ID	Name	PasswordChars	PasswordLength	Password	Hint1	Hint2	Hint3	Hint4	Hint5	Hint6	Hint7
1	Sophia	ABCDEFGHIJK	10	c4712866799881ac48ca55bf78a9540b1582824a01c41e91aca467f5a2b52be0093f91b98052d9420a20ca70f765d8c1b570d3ada41def224061bd0359							
2	Jackson	ABCDEFGHIJK	10	c178ef3bd2dbf4e92291a9b563c0ae2c7624e76e72b52834d255d0276b2e939a89b780f0c2aefcfc4b3d22b558963201e0066eb98a0f321b5a6f0b9c15							
3	Olivia	ABCDEFGHIJK	10	b6d							
4	Liam	ABCDEFGHIJK	10	109							
5	Emma	ABCDEFGHIJK	10	607							
6	Noah	ABCDEFGHIJK	10	6d4							
7	Ava	ABCDEFGHIJK	10	4121ab0055971e							
8	Aiden	ABCDEFGHIJK	10	fbe3613750f71d7996e9d63601dc7fd4de2617fb757fc06bb6d175e5d03ee78244a7287316b71fbfc49aab84d04556e87a65ceb83b59589c35f40243c							
9	Isabella	ABCDEFGHIJK	10	5a22e3bdef6c85307b361f2e1758f46123d6de9da42517af3c5c070a12824137665f56c71deb0e1e1849535ddb45d79271a854a0e06b2bdf84e0a							
10	Lucas	ABCDEFGHIJK	10	49afdd0a20ae497060405ec7b557faa0417341646430df7feecbe4bb046c1fec90e0a221a7c41ebf4dcbe04357c159ae51984b3c8ce5b090db2396							
11	Mia	ABCDEFGHIJK	10	77026d73fb8c33e0f45c3f6bc3							
12	Caden	ABCDEFGHIJK	10	484616315092a69ebd7cf4c1b							
13	Aria	ABCDEFGHIJK	10	3ff9b667a867fccaada0d823d0							
14	Grayson	ABCDEFGHIJK	10	ac923aa891c087fad57b02de9							
15	Riley	ABCDEFGHIJK	10	57203d2db503c69464900aed							
16	Mason	ABCDEFGHIJK	10	4d873360dd931098ead7d692							
17	Zoe	ABCDEFGHIJK	10	f2095d3f48f6c0366423436865							
18	Elijah	ABCDEFGHIJK	10	25e975a018dd7265dcb44a17							
19	Amelia	ABCDEFGHIJK	10	6fb693ee39e015290f087a0ca							
20	Logan	ABCDEFGHIJK	10	1d43da0376f725fff867e1096e3635c9a8fb							
21	Layla	ABCDEFGHIJK	10	c3647d6d4f8e8136cf7640d1976d2349							
22	Oliver	ABCDEFGHIJK	10	d2488287e89e2bb00bffc4e767fe587c4c588e7b790b911eca14216df52b85e1bb8c3ba17b3532e88848a569dcd0f9b2ecef6af2a9ab4b2577e1e3							
23	Charlotte	ABCDEFGHIJK	10	0e481c55eea1567f4a543cc0d713dd6426c5a36fa47578c180f1b1946588a7e0f05e5daaac4464d5bf1244e8e8879fe2147bd94da4f7a8c1b228272							
24	Ethan	ABCDEFGHIJK	10	d08ce9b35434a29b6d34ae4df99114e537ceb64562e2ac4c0b2db991b43e80f0be33cccd5b0f386b8868e118ee7ddfeed78dc7c439e30e4a6278bf0							
25	Aubrey	ABCDEFGHIJK	10	a54							
26	Jayden	ABCDEFGHIJK	10	482							
27	Lily	ABCDEFGHIJK	10	64e							
28	Muhammad	ABCDEFGHIJK	10	b24							
29	Chloe	ABCDEFGHIJK	10	314885f3b250cfad9a08ab7c6a0b71259ba60bc240c6f8b32fc6c7040z							
30	Carter	ABCDEFGHIJK	10	507b389927e0aa92bdf50e7ffe0c119c2221935370639ec62e5d714fd4f4ce80bb607e3e0e6e85ce7c1163a66461e18e32024bb204d35243c99da							
31	Harper	ABCDEFGHIJK	10	17649029a718c93179e9da331e78012f4aa95f0083c66b0c11abac19e8878f2bea2d961101eb1bfae6d694f11668eca52a2e2681f0d0f5194a8889827							
32	Michael	ABCDEFGHIJK	10	a926dea7e334a3992fbfa30d4d758238b63be6310da03fbc4f9b69fd96188307f70e6bdc76976d88e54cb6eabaa209c2f8383f1aa504bd3a11b5e4							
33	Evelyn	ABCDEFGHIJK	10	43079487b664ebafba46e77698d58a47b43a0546a75fc6c9a5d45c1f967cbc51d481eb1a79b242950f59099a87582e2ac6479c44e487d490919fceb7							
34	Sebastian	ABCDEFGHIJK	10	0306aed6a72de9d320e0b9d9ec430e928e5837886ae82a9f2b7b2e975effda9ae8fd9037bc1fd83a003d4b9e8ba7bc9379b2c12b91f1f3b79b0e05dc							
35	Adalyn	ABCDEFGHIJK	10	bef1a0cc6ba9868fe2071e80b7069f24e622ba2b0c45571087ebc69f5b0c28553d4a05874802c5978ebf50e7701462967ba4c14f03ca9d2bda495640a							
36	Alexander	ABCDEFGHIJK	10	f14a798017874d94e78421db5a126e650e4f0b88e214b5502b12a7d7d897e5993c0d11547ce885e7caae5f28e181e9cb8b00cda9da84080a088							

Both password and hints are SHA-256 encrypted.

Encryption cracking via brute force approach:

1. Generate sequence.
2. Encrypt sequence with SHA-256.
3. Compare current SHA-256 with existing one: if equal, encryption is broken.

Hint cracking is much easier than password cracking.

Homework

Task 3 – Password Cracking

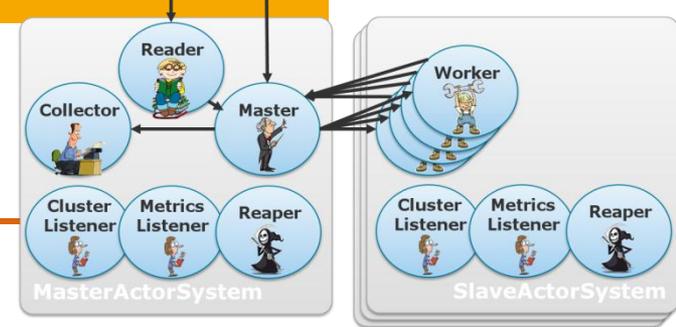
Hints

- The passwords and hints are encrypted with the following function:

```
private String hash(String password) {  
    MessageDigest digest = MessageDigest.getInstance("SHA-256");  
    byte[] hashedBytes = digest.digest(line.getBytes("UTF-8"));  
    StringBuffer stringBuffer = new StringBuffer();  
    for (int i = 0; i < hashedBytes.length; i++)  
        stringBuffer.append(Integer.toString((hashedBytes[i] & 0xff) + 0x100, 16).substring(1));  
    return stringBuffer.toString();  
}
```

- Useful code snippets for combination generation:

- <https://www.geeksforgeeks.org/print-all-combinations-of-given-length/>
- <https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/>



Distributed Data Management

Akka Actor Programming

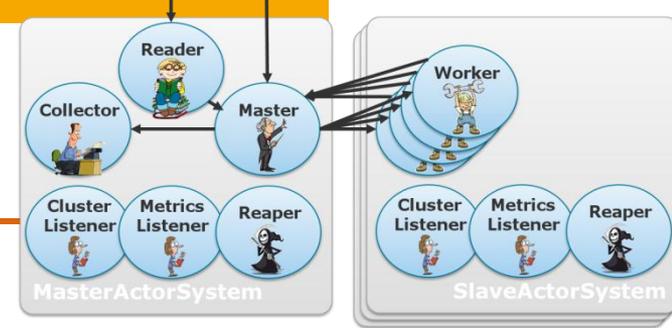
ThorstenPapenbrock
Slide **152**

Homework

Task 3 – Password Cracking

Hints

- Think agile:
 - How can I maximize the parallelization?
(e.g. the number parallel tasks should in the best case not depend on the input data)
 - How can I propagate intermediate results to other actors whenever needed?
(e.g. proxies, schedulers, master-worker, ...)
 - How can I re-use intermediate results to dynamically prune tasks?
(e.g. if I know that X is a solution, then I might be able to infer without testing that Y is also a solution)
 - How can I implement task parallelism?
(e.g. parts of subtask 2 might already be able to start with partial results of subtask 1)
 - How can I achieve elasticity in the number of cluster nodes?
(nodes may join or leave the cluster at runtime)



Distributed Data Management

Akka Actor Programming

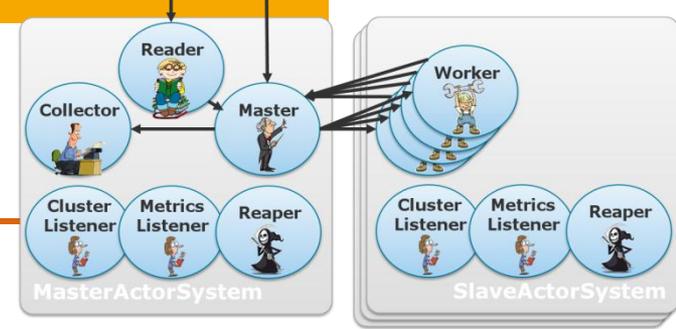
ThorstenPapenbrock
Slide **153**

Homework

Task 3 – Password Cracking

Notes

- Parameters that may change:
 - password length
 - password chars
 - number of hints (= width of file)
 - number of passwords (= length of file)
 - number of cluster nodes
(do not wait for x nodes to join the cluster; you do not know their number; implement elasticity, i.e., allow joining nodes at runtime)
- Parameters that may not change:
 - encryption function SHA-256
 - all passwords are of same length and have same character universe



Distributed Data Management

Akka Actor Programming

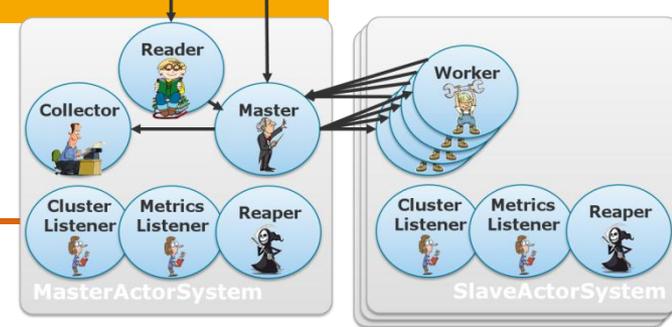
ThorstenPapenbrock
Slide **154**

Homework

Task 3 – Password Cracking

Rules

- **Do not mess with the time measurement:**
It should start with the StartMessage and it should end when the PoisonPills are sent.
- **Do not change the command line interface or app name;**
otherwise, the automatic test scripts will fail.
- **Use maven** to import additional libraries if you need some.
- **Do not use the disk.**
- **Feel free to change everything** (besides interface and time measurement);
you probably need a new shutdown protocol, you need a proper communication protocol for your Master/Worker actors and you probably need additional actors.
- **Write the cracked passwords with the Collector to the console;**
the current printouts from the master should be deleted.



Distributed Data Management

Akka Actor Programming

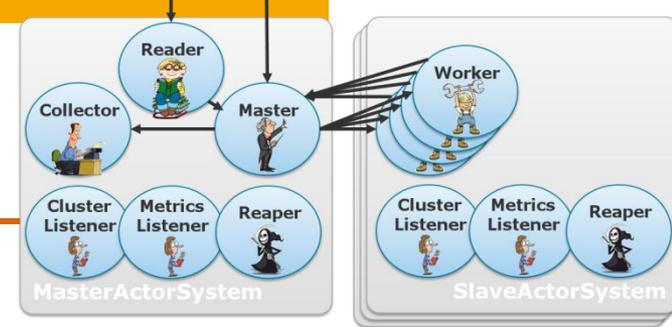
ThorstenPapenbrock
Slide **155**

Homework

Task 3 – Password Cracking

Submission

- **Deadline**
 - 28.05.2021 23:00:00
- **Artifacts** (in GitHub repository)
 1. Source code
 2. "assignment3" folder with ...
 1. a jar file of your algorithm;
 2. a pdf or ppt slide describing your solution.

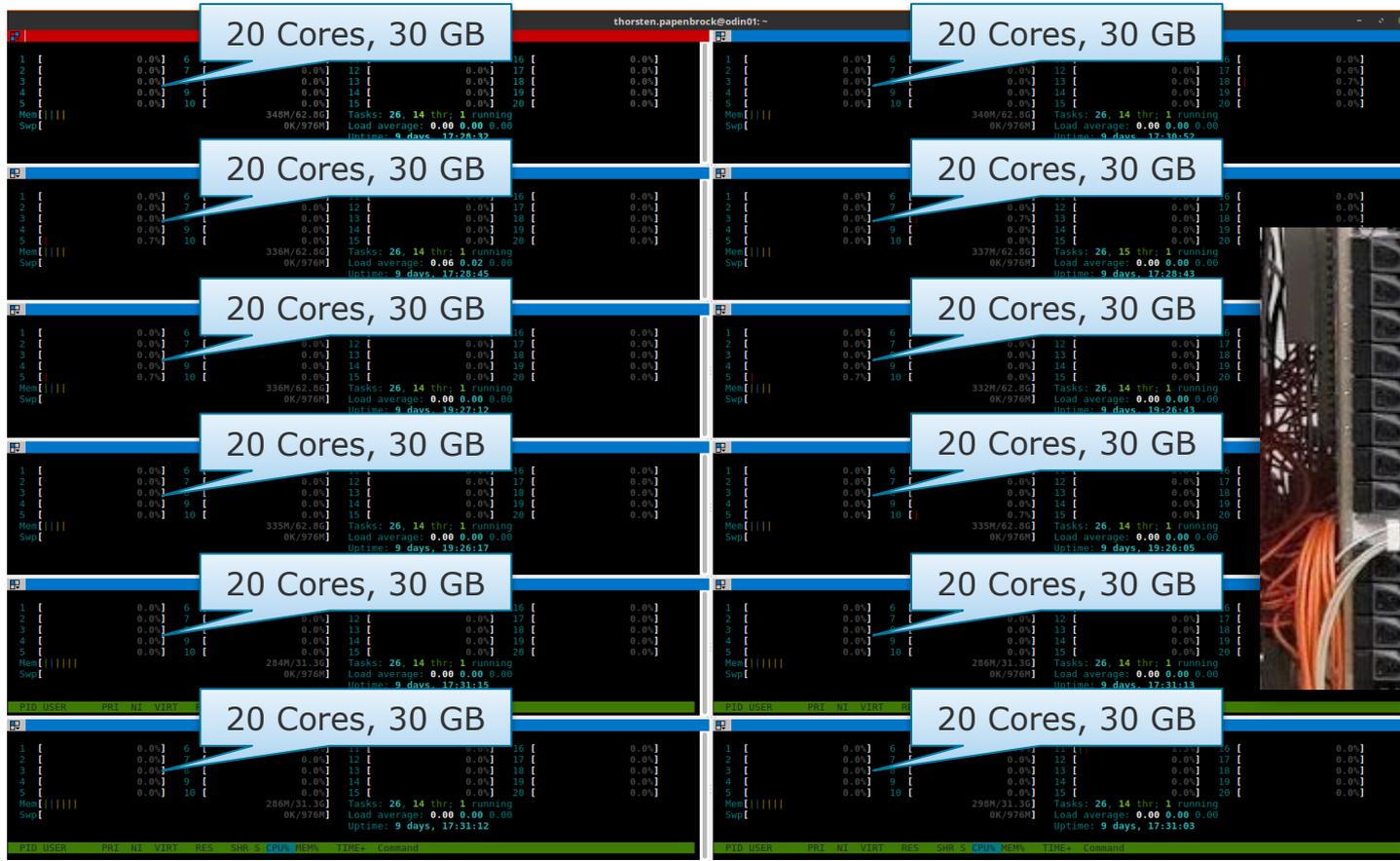


Distributed Data Management

Akka Actor Programming

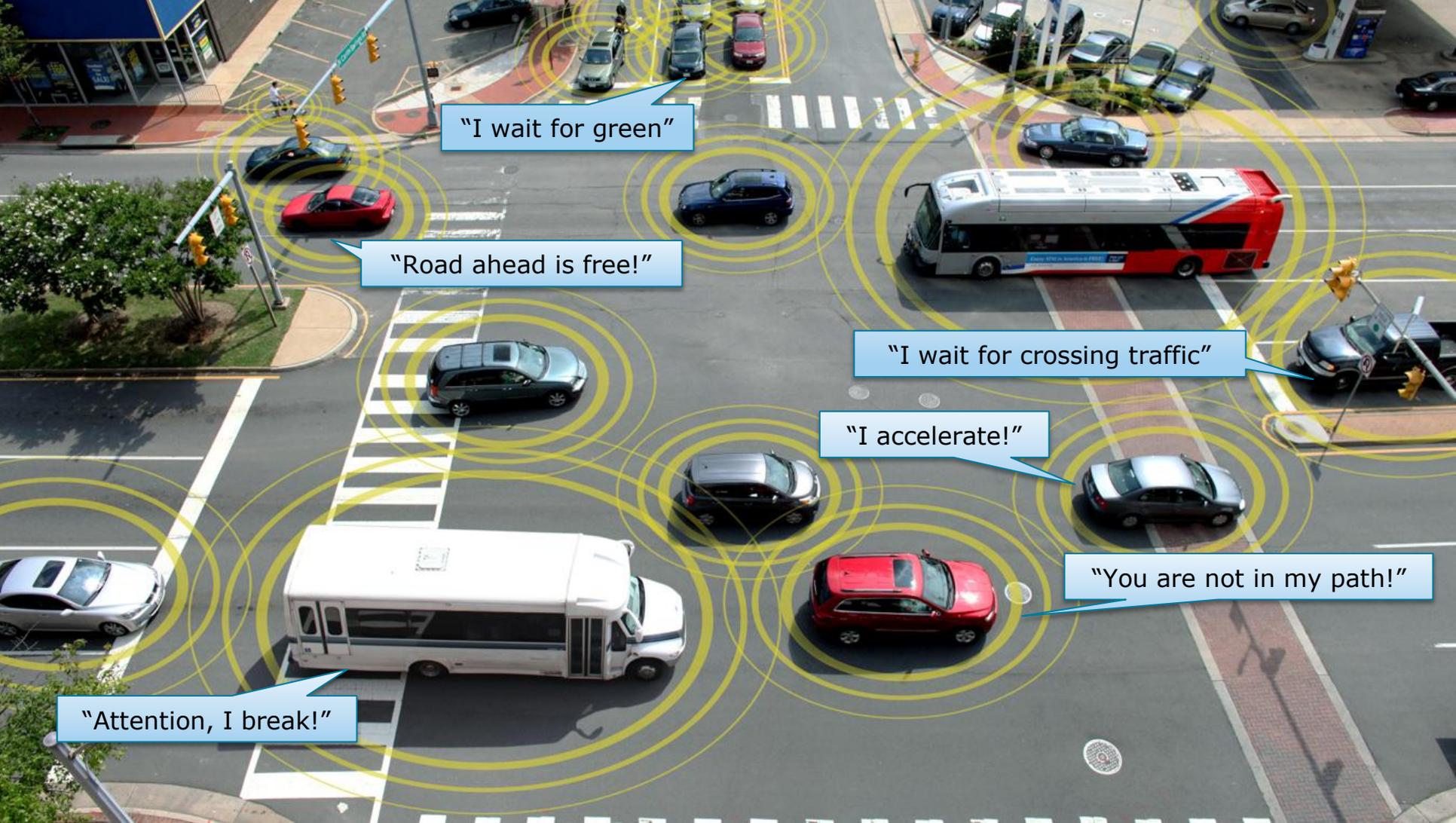
ThorstenPapenbrock
Slide **156**

Homework Evaluation – Odin/Thor Cluster





Best Team wins a price!



"I wait for green"

"Road ahead is free!"

"I wait for crossing traffic"

"I accelerate!"

"You are not in my path!"

"Attention, I break!"