
# Distributed Data Management
# Akka Actor Programming

Thorsten Papenbrock

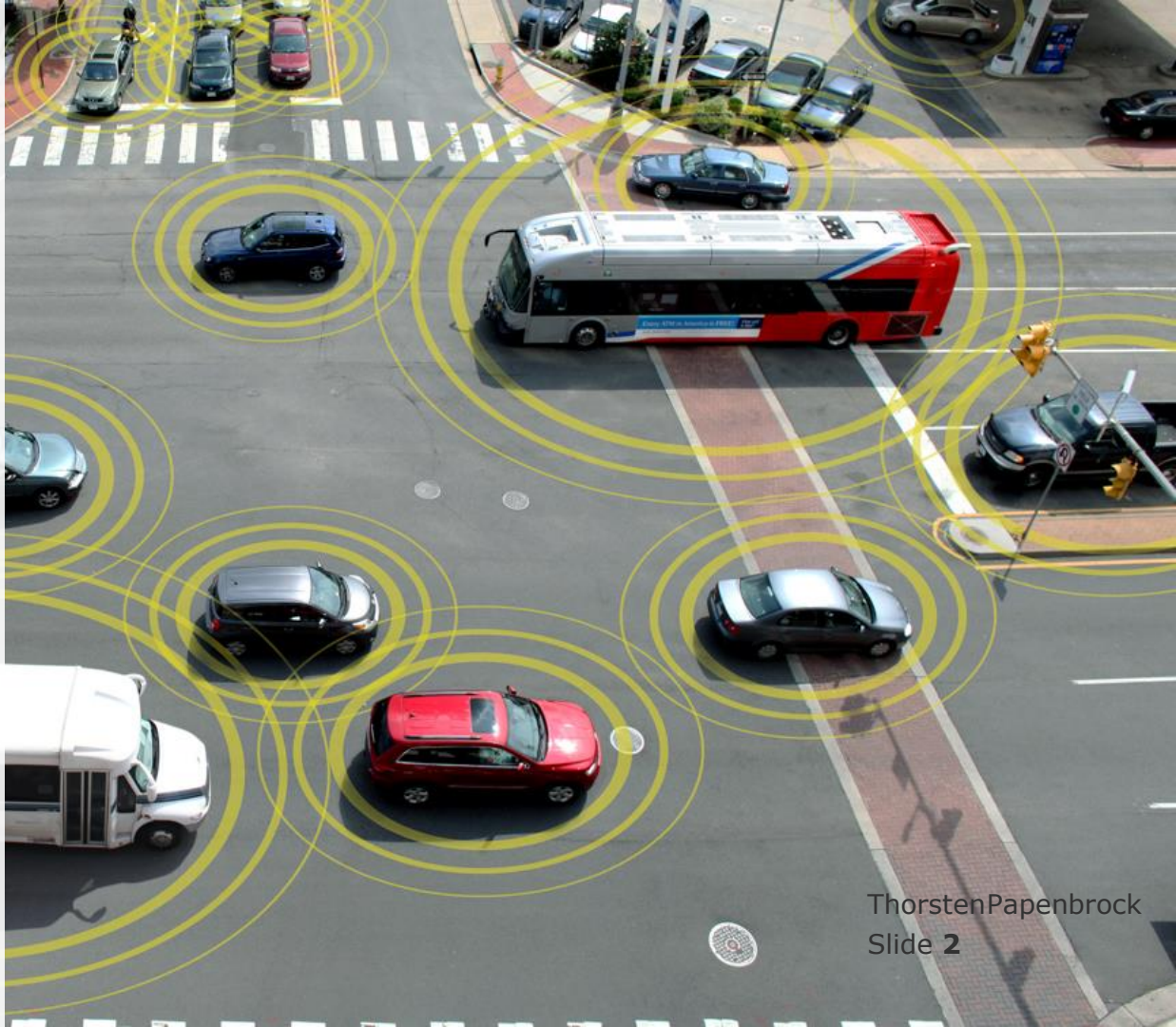
F-2.04, Campus II

Hasso Plattner Institut

Akka Actor Programming

# Hands-on

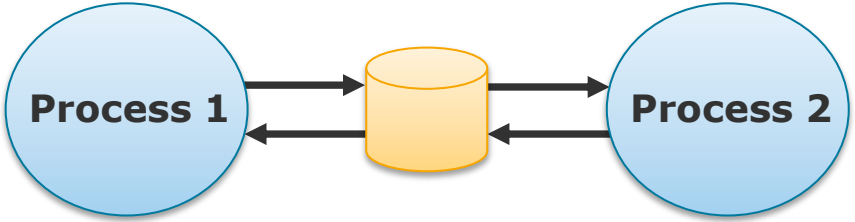
- **Actor Model (Recap)**
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- Remoting
- Clustering
- Patterns
- Homework



# Models of Dataflow

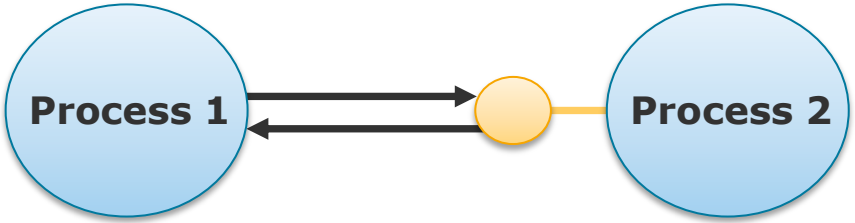
## Dataflow through Databases

- information storage and retrieval



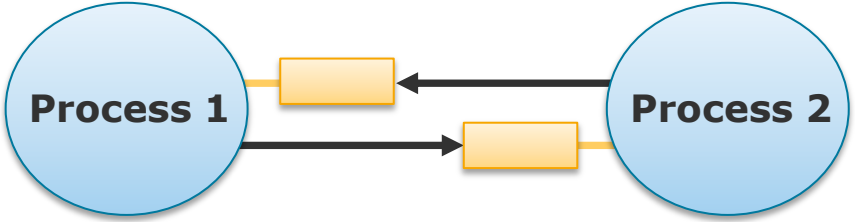

## Dataflow through Services

- service calls with responses

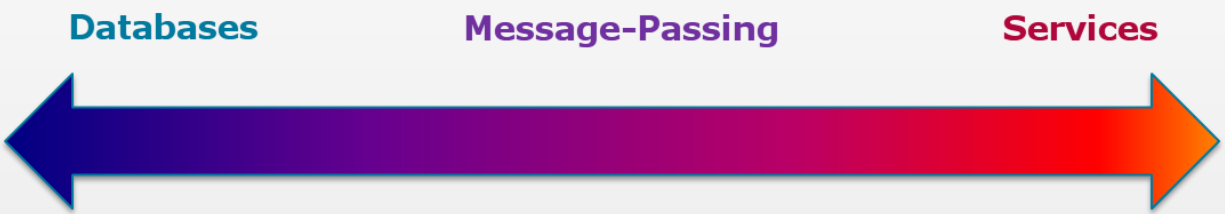



## Message-Passing Dataflow

- asynchronous messages



# Models of Dataflow




| Databases | Message-Passing | Services |
|---|---|---|
| Data | Messages | Function calls |
| No response | Maybe response | Response |
| Non-blocking | Usually non-blocking | Blocking |
| Asynchronous | Asynchronous | Synchronous |
| No addressing | Addressing recipient | Addressing recipient |



# 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.

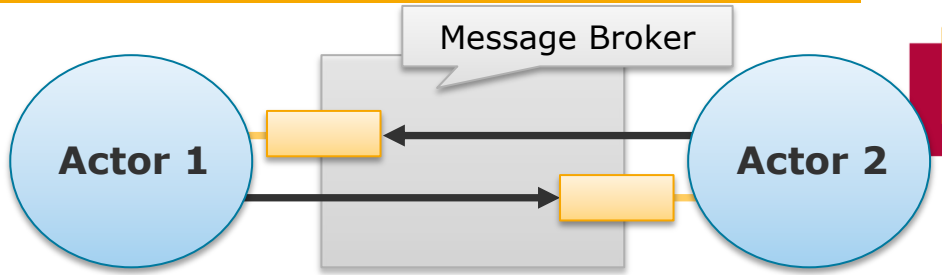
➢ Combines the advantages of object- and task-oriented programming.

**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.

# Actor Model

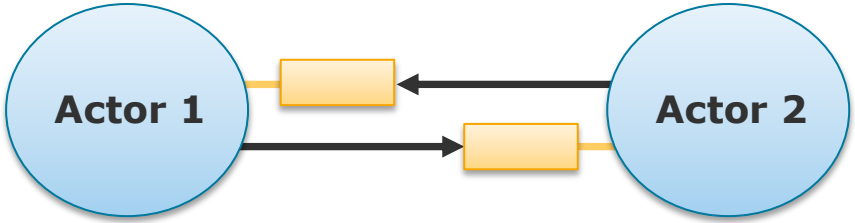



## Actor Model

- 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 prevents many parallel programming issues (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

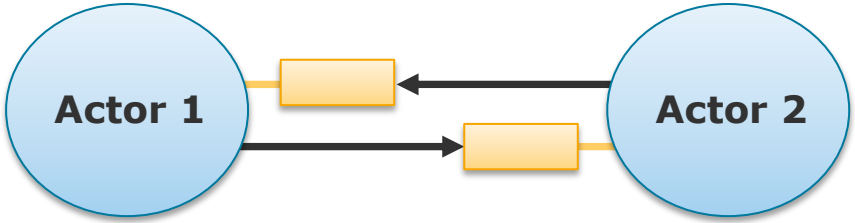
# Actor Model




## Advantages over pure RPC

- Fault-tolerance:
  - "Let it crash!" philosophy to heal from unexpected errors
  - Automatic restart of failed actors; resend/re-route of failed messages
    - Errors are expected to happen and implemented into the model:
- Deadlock/starvation prevention:
  - Asynchronous messaging and private state actors prevent many parallelization issues
- Parallelization:
  - Actors process one message at a time but different actors operate independently (parallelization between actors not within an actor)
  - Actors may spawn new actors if needed (dynamic parallelization)

# Actor Model



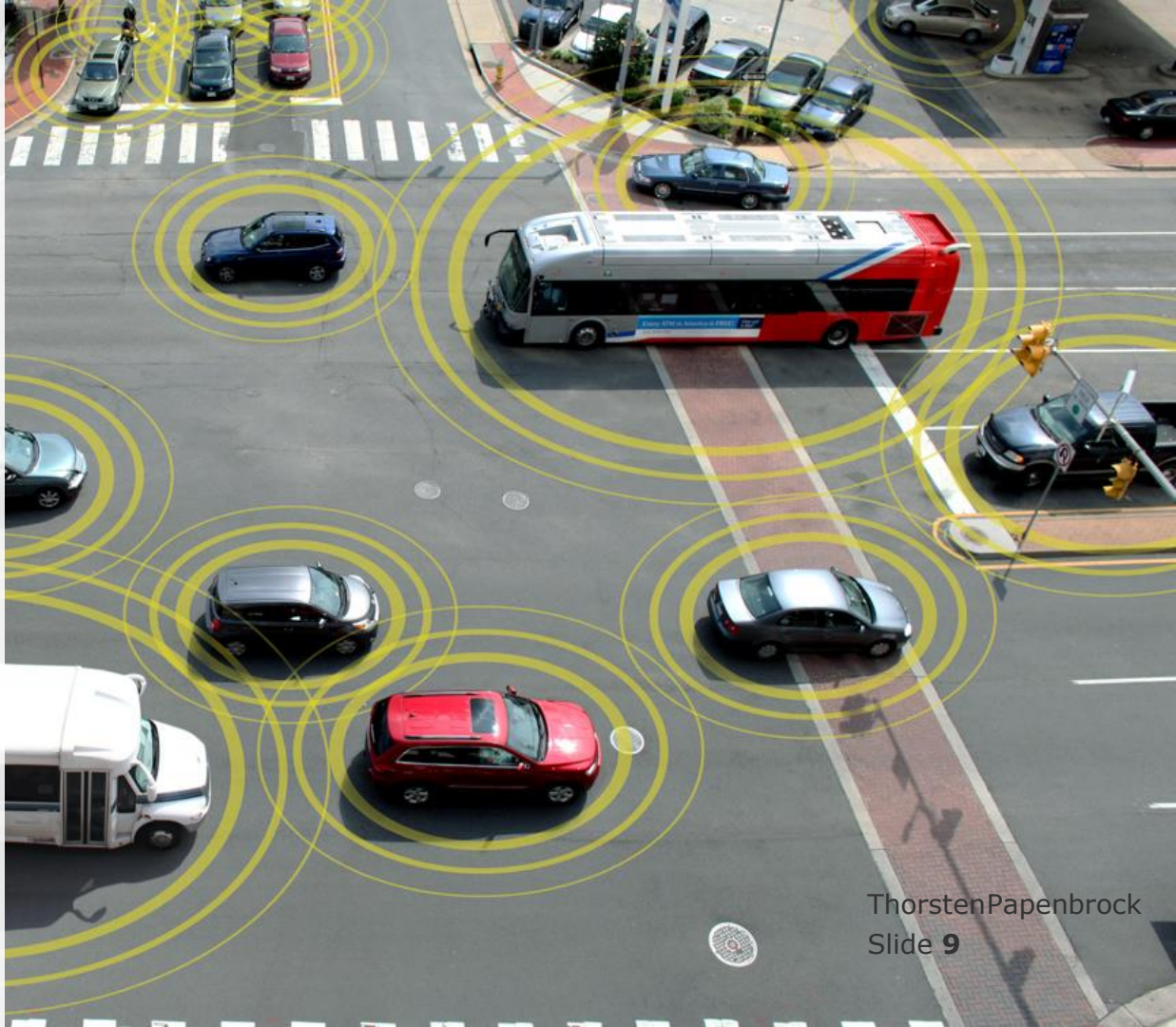

## Popular Actor Frameworks

- Erlang:
  - Actor framework already included in the language
  - First popular actor implementation
  - Special: Native language support and strong actor isolation
- Akka:
  - Actor framework for the JVM (Java and Scala)
  - Most popular actor implementation (at the moment)
  - Special: Actor Hierarchies
- Orleans:
  - Actor framework for Microsoft .NET
  - Special: Virtual Actors (persisted state and transparent location)

Akka Actor Programming

# Hands-on

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



# Akka Toolkit and Runtime


- A free and open-source toolkit and runtime for building concurrent and distributed applications on the JVM (https://akka.io/)
- Supports multiple programming models for concurrency, but emphasizes actor-based concurrency
- Inspired by Erlang (https://erlang.org/)
- Written in Scala (https://scala-lang.org/)
  - included in the Scala standard library
- Invented by Jonas Bonér; maintained by Lightbend (https://lightbend.com/)
- Offers interfaces for Java and Scala

# 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

# Small Setup

## Maven – pom.xml

**Base actor library**
actors, supervision, scheduling, …

**Remoting library**
remote actors, heartbeats …

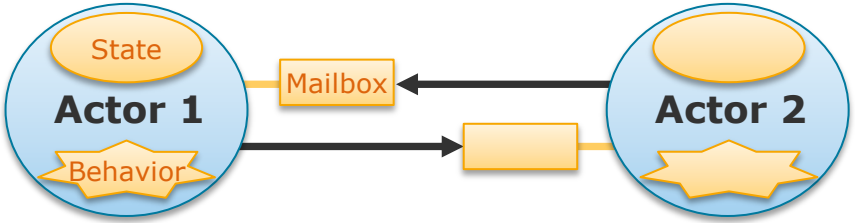
**Logger library**
logging event bus for akka actors

**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>
```

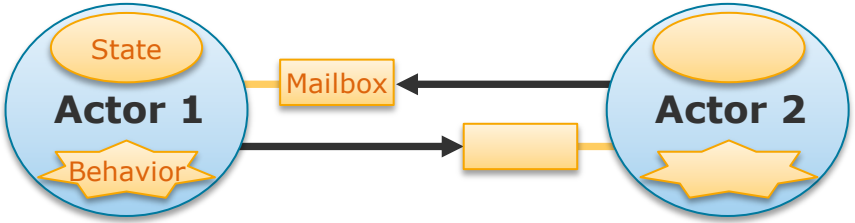
# Akka Actors




- Actor = State + Behavior + Mailbox
- Communication:
  - Sending messages to mailboxes
  - Unblocking, 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!**

# Akka Actors




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

```
public class Wor  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());
  }
}
```

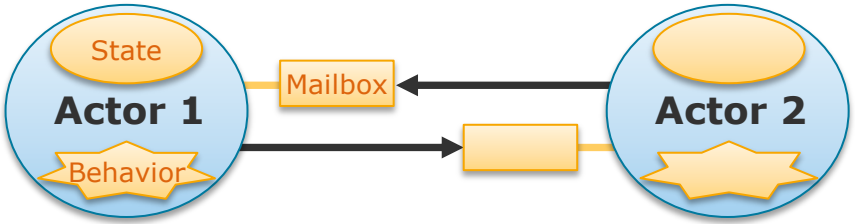
Inherit default actor behavior, state and mailbox implementation

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)

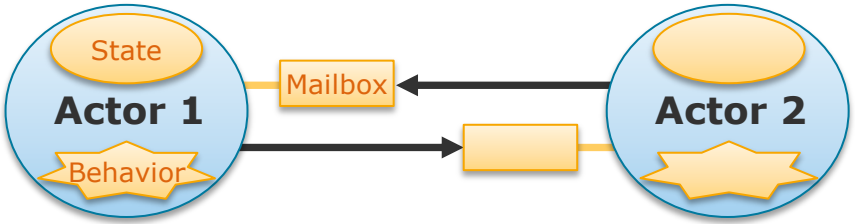
# Akka Actors




```
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(Doube.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

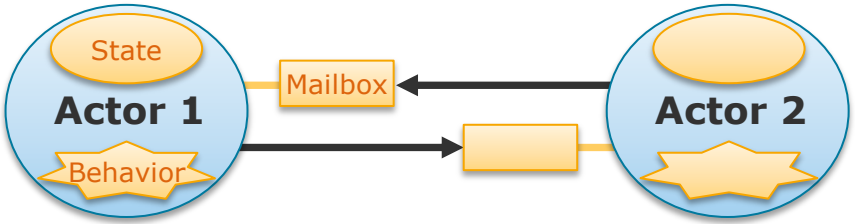
# Akka Actors




```
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(Doube.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 propper logging

# 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)

```
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 a ActorRef over which it can expect responses

# 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!

```scala
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")
  }
}
```

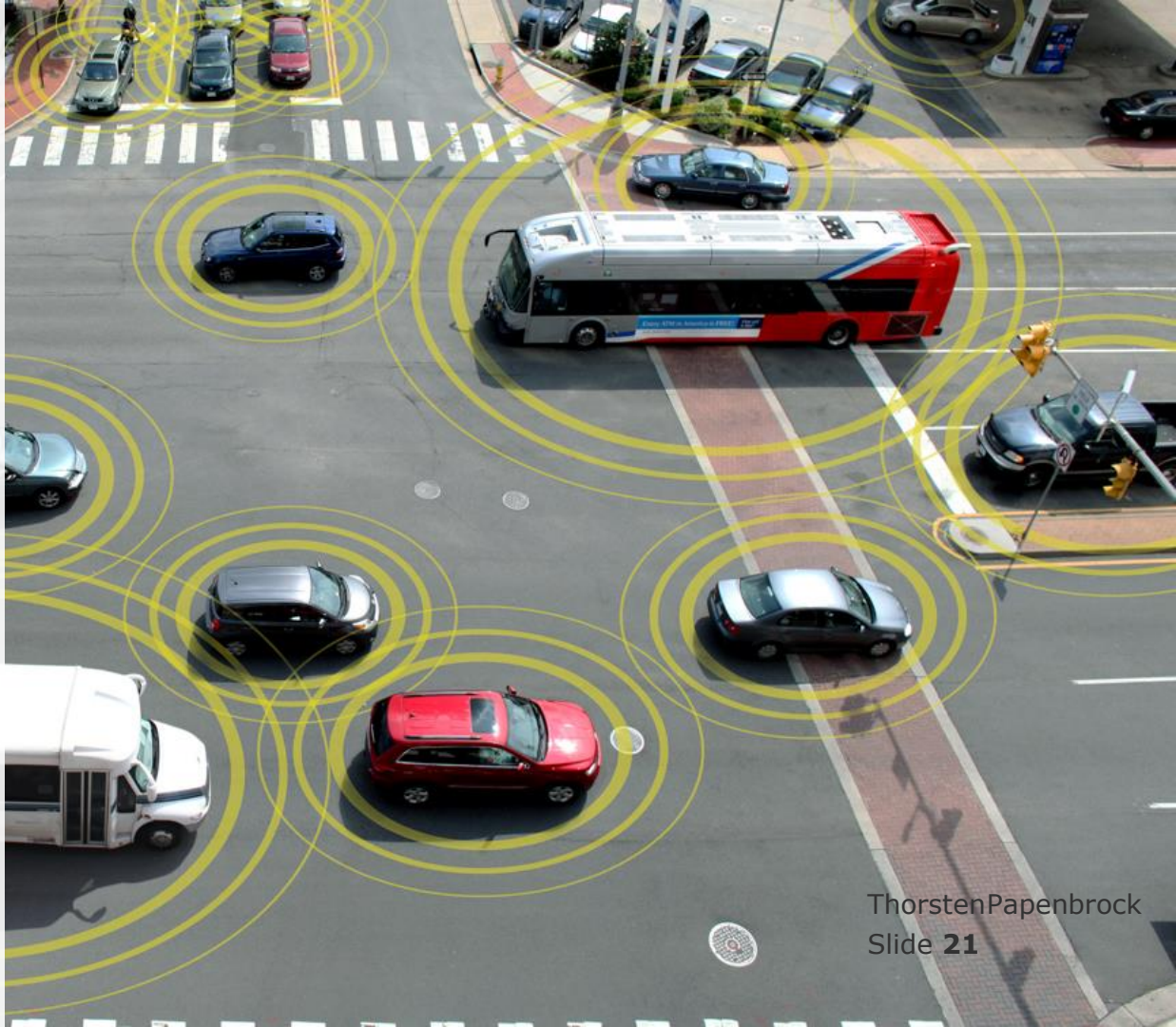
# Akka Scala Interface

## A Master-Worker Example

Akka Actor Programming

# Hands-on

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



# Actor Hierarchies

## Task- and data-parallelism

- Actors can dynamically create new actors.

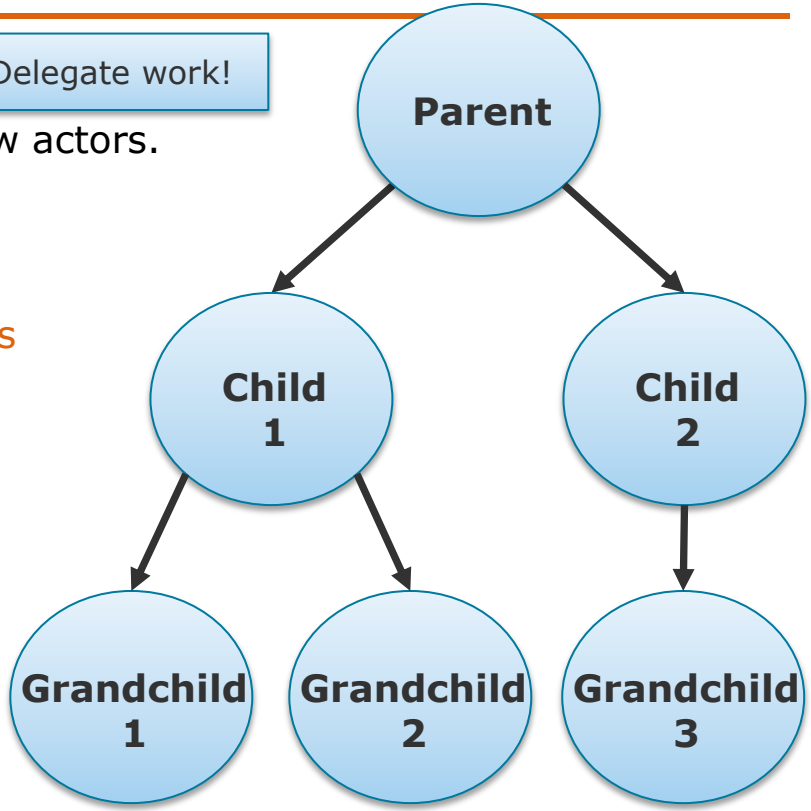
Delegate work!

## Supervision hierarchy

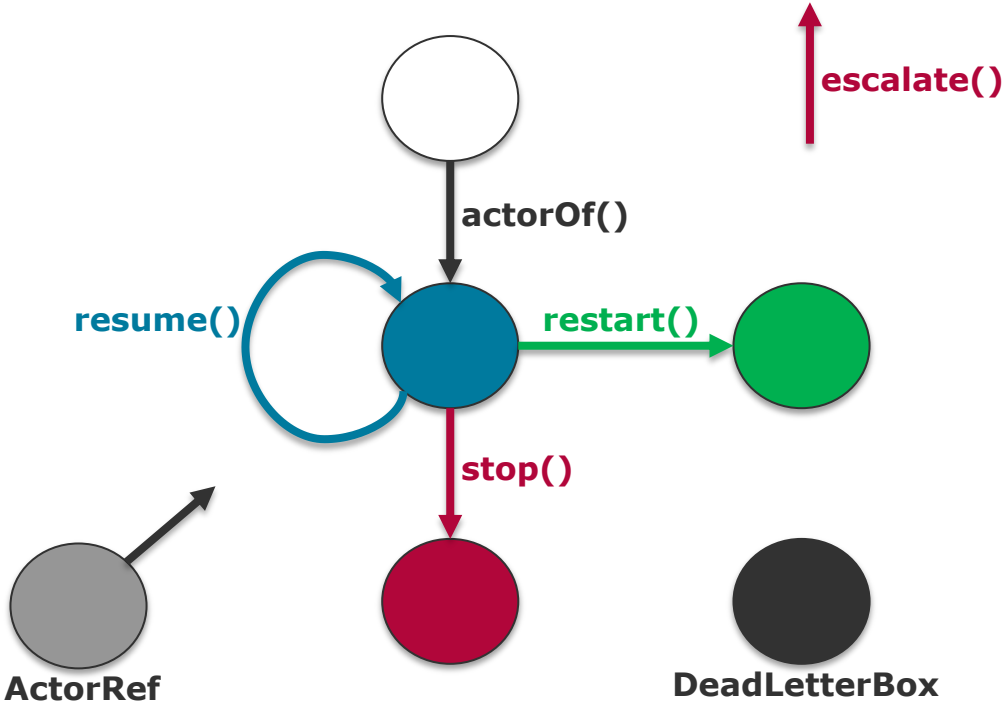
- Creating actor (parent) supervises created actor (child).

## Fault-tolerance

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



# Actor Lifecycles



# 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());
  }
}
```

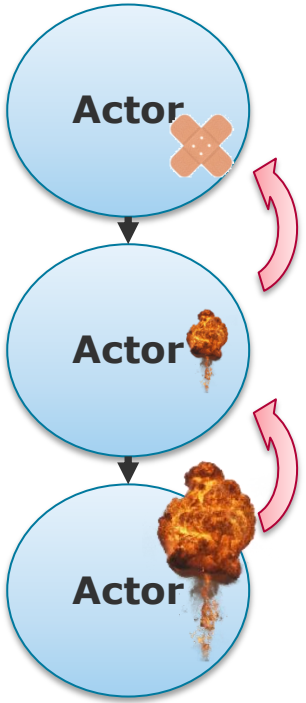
Log that **MyActor** was stopped

# 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 looses a database connection due to a DB restart.
  - It decides to crash.
  - Its supervisor restarts the actor, which re-creates the DB connection.



# Actor Hierarchies

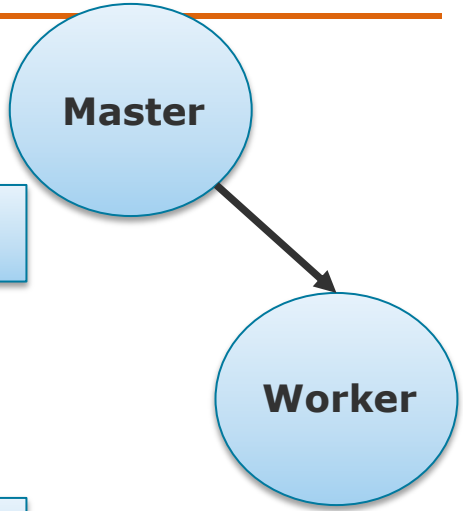
```
public class Master extends AbstractLoggingActor {
  public Master() {
    ActorRef worker = this.context().actorOf(Worker.props());
    this.context().watch(worker);
  }
  @Override
  public SupervisorStrategy supervisorStrategy() {
    return new OneForOneStrategy(3,
      Duration.create(10, TimeUnit.SECONDS),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

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

Receive **Terminated**-messages for watched actors

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

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

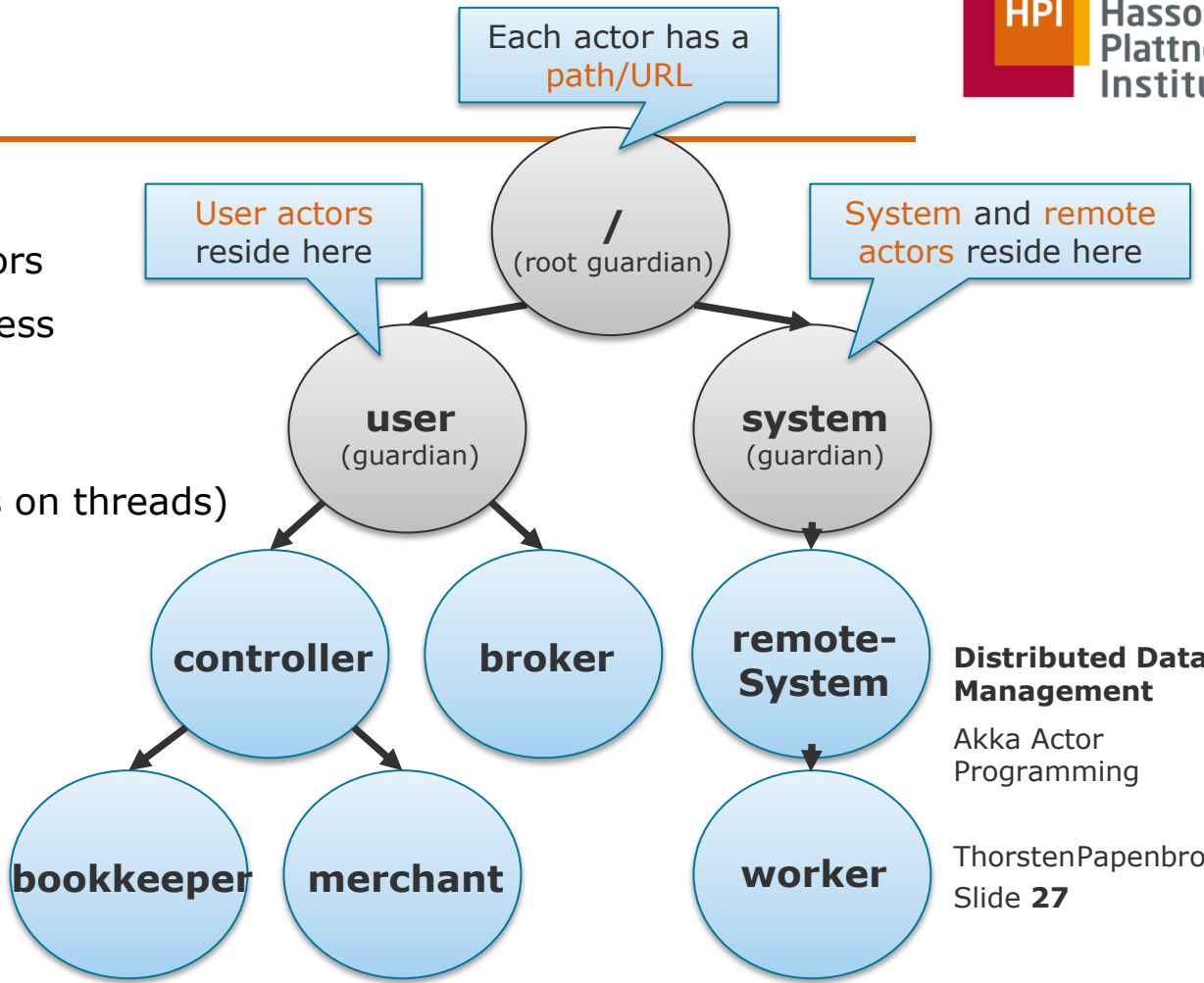
A factory pattern



# 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)
  - …



# Actor Systems

Event stream

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

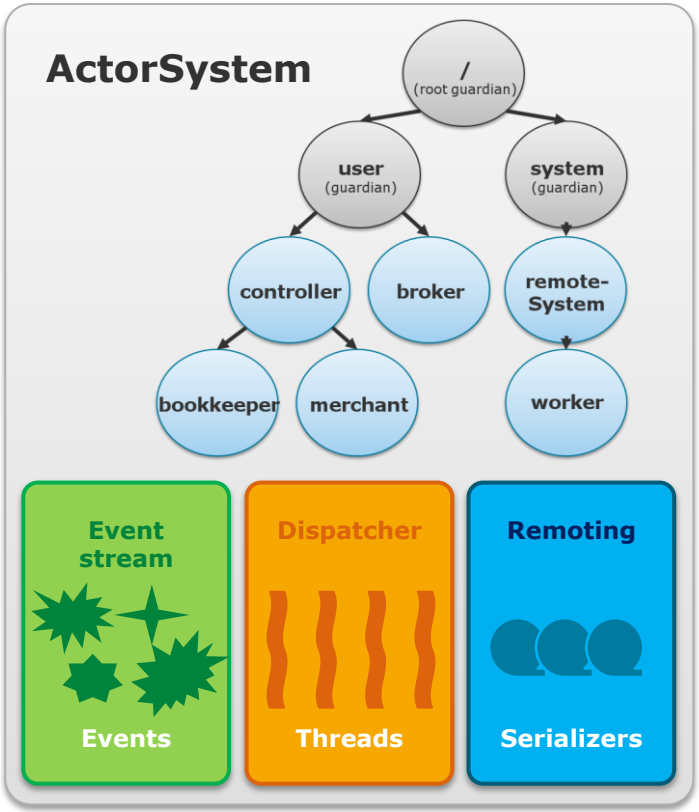
Dispatcher

- Assigns threads dynamically to actors.
- Transparent multi-threading
  - # Threads ≈ # CPU cores
  - # Actors > # CPU cores (usually many hundreds)
  - ➢ Over-provisioning!

Idle actors don't bind resources

Remoting

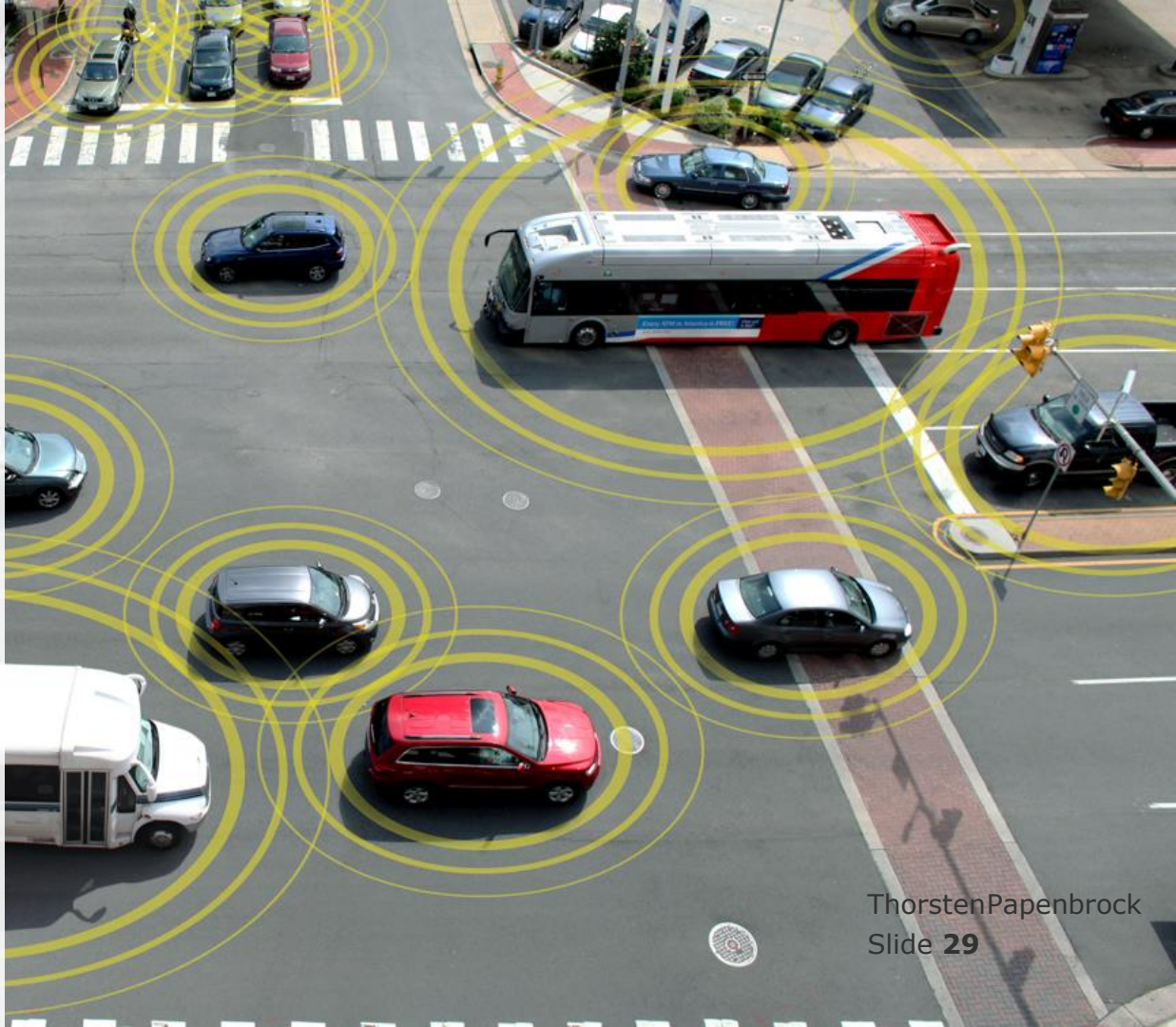
- Resolves remote actor adresses.
- Sends messages over network.
  - serialization + de-serialization



Akka Actor Programming

# Hands-on

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



# HPI-Information-Systems / akka-tutorial

Watch 7 Star 5 Fork 7

Code Issues 0 Pull requests 0 Projects 0 Security Insights

## Code for the Akka tutorial

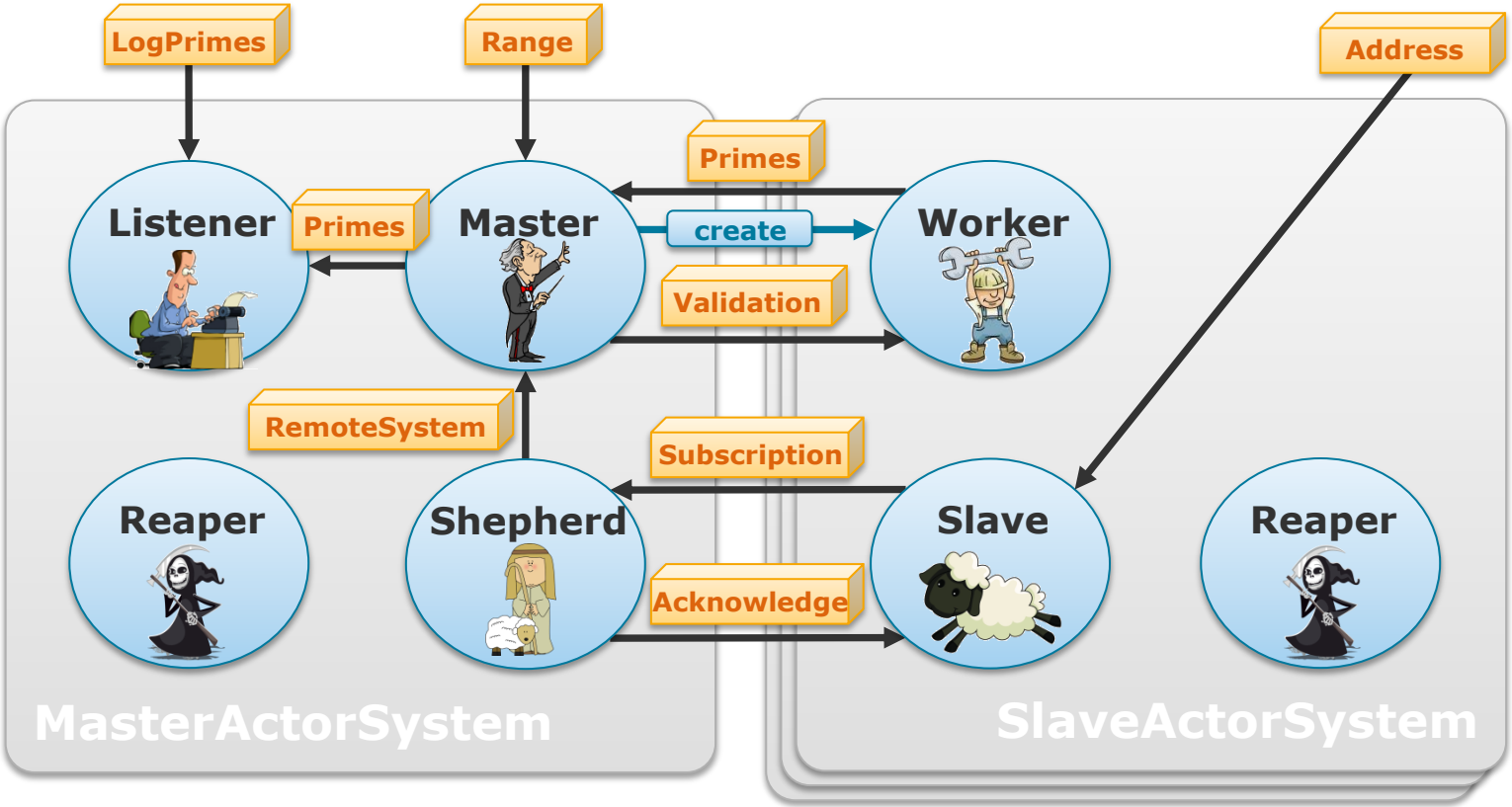
**53** commits **1** branch **0** releases **5** contributors Apache-2.0

Branch: **master** New pull request Find file Clone or download

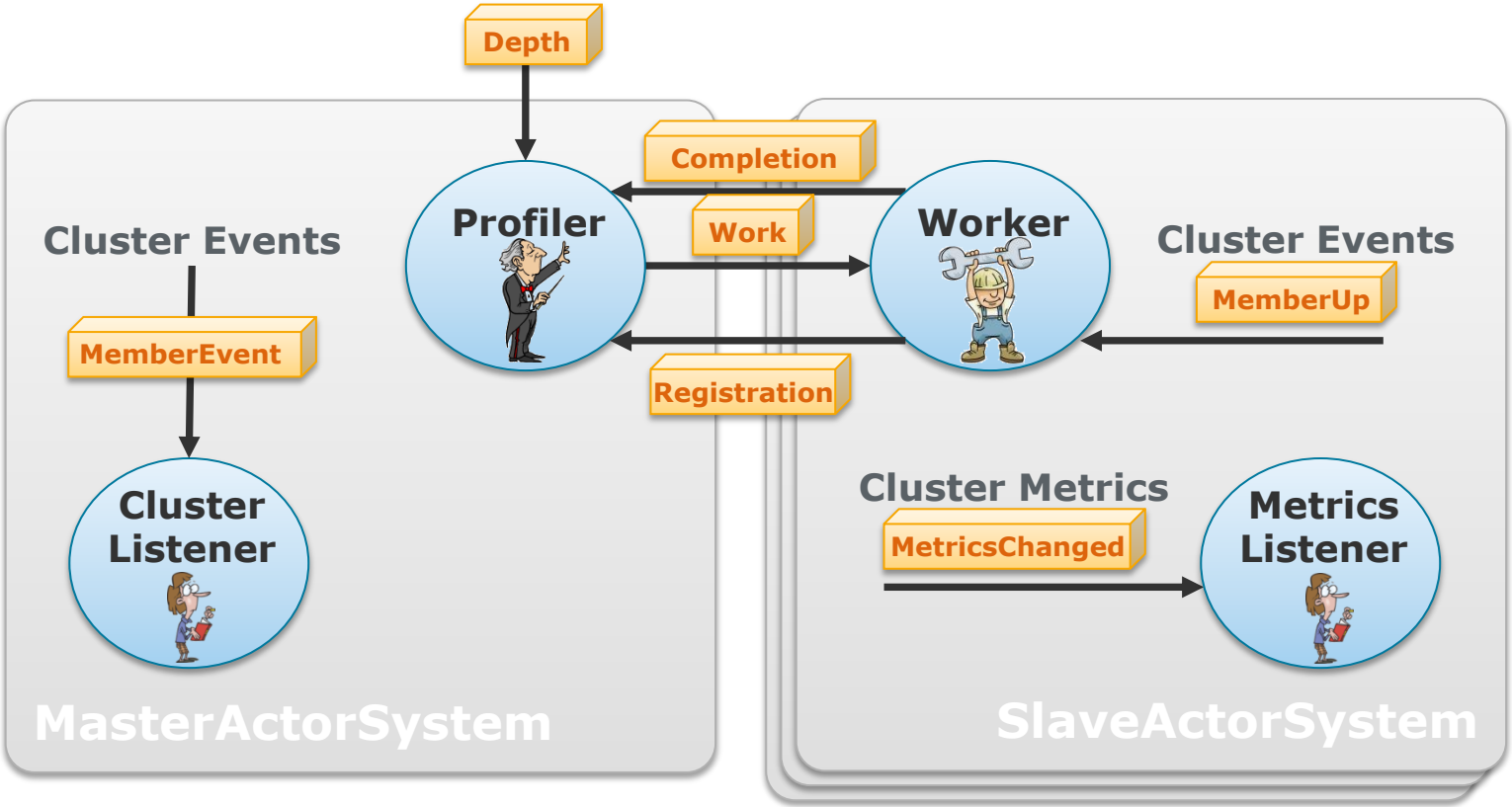
**thorsten-papenbrock** Split the ddm project into ddm-lmp and ddm-pc. Latest commit `bfe21b6` 5 minutes ago

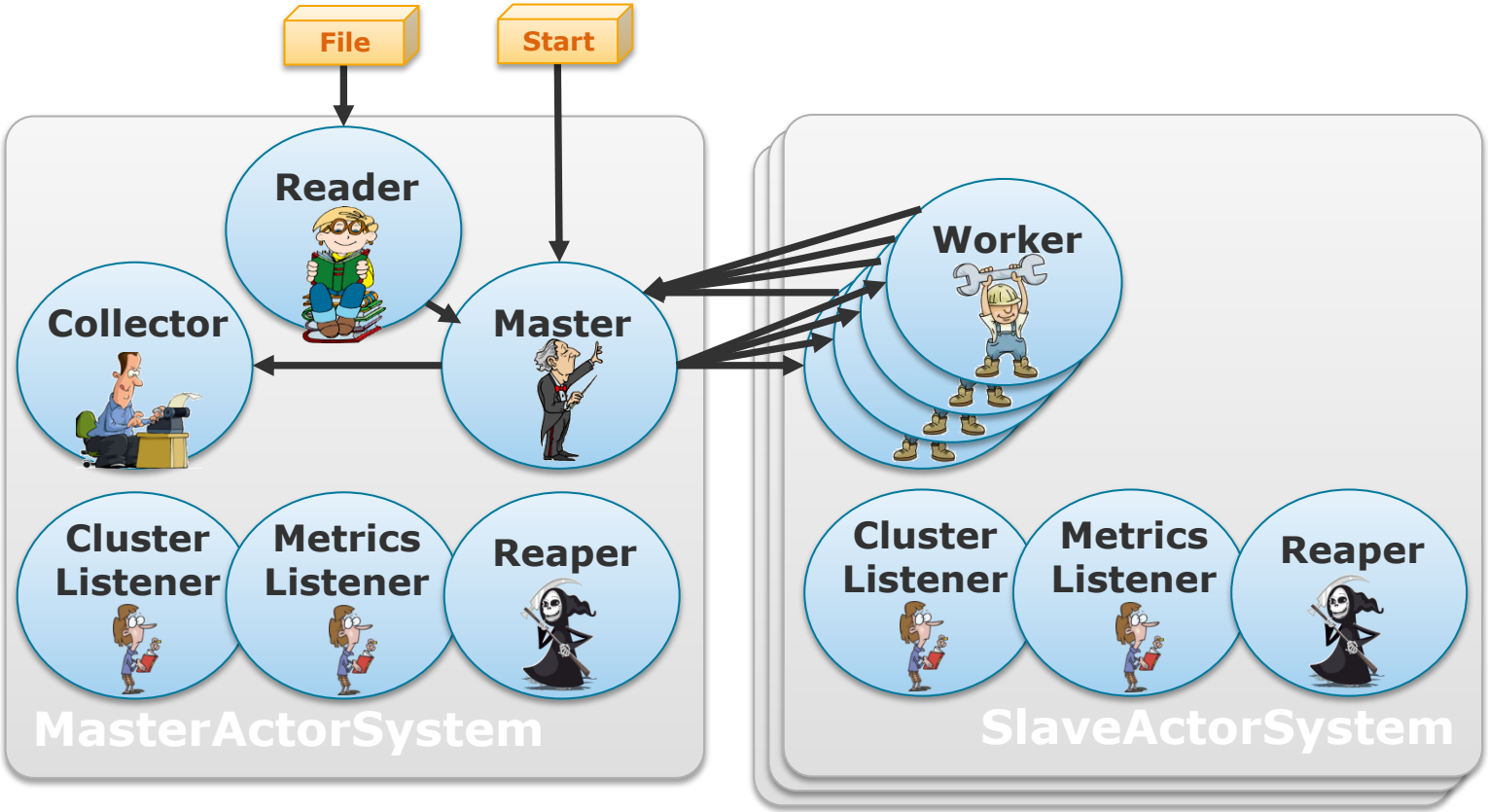
| | | |
|---|---|---|
| akka-tutorial | 1 is not a prime | 11 months ago |
| ddm-lmp | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| ddm-pc | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| octopus | 1 is not a prime | 11 months ago |
| .gitignore | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| LICENSE | Added the octopus project to the repository. | last year |
| README.md | Added the ddm project. | 21 days ago |

Demo
# akka-tutorial



# octopus

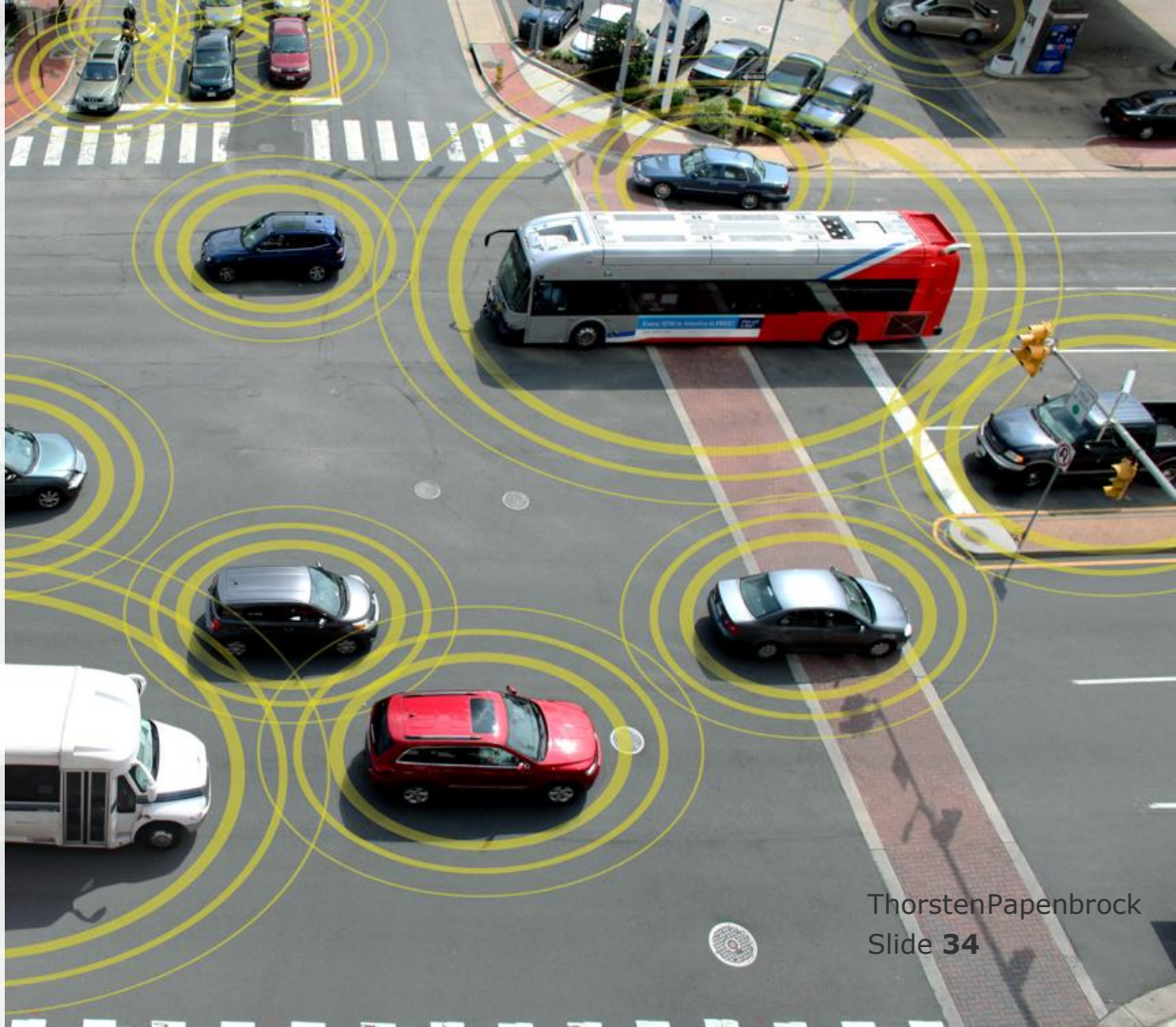

Depth
Completion
Work
Registration
Profiler
Worker
Cluster Events
MemberEvent
Cluster Listener
Cluster Events
MemberUp
Cluster Metrics
MetricsChanged
Metrics Listener
MasterActorSystem
SlaveActorSystem


File
Start
Reader
Collector
Master
Worker
Cluster Listener
Metrics Listener
Reaper
MasterActorSystem
Cluster Listener
Metrics Listener
Reaper
SlaveActorSystem

Akka Actor Programming

# Hands-on

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



# 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
- at-least-once: each message is delivered one or more times.
  - guaranteed delivery; possibly message duplication
  - ok-isch performance; state in sender
  - send-and-acknowledge
- exactly-once: each message is delivered once.
  - guaranteed delivery; no message duplication
  - bad performance; state in sender and receiver
  - send-and-acknowledge-and-deduplicate

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

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

https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html

# Message Delivery Guarantees

## Message ordering

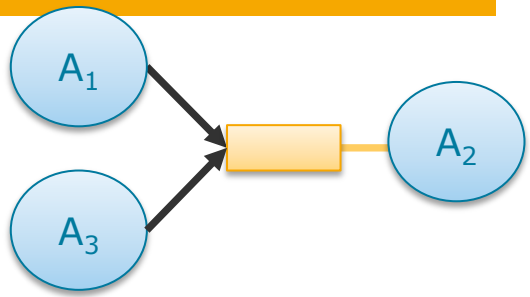
- no ordering: all messages can be arbitrarily out of order
  - no guaranteed ordering
  - highest performance; no implementation overhead
- **sender-receiver ordering**: all messages between specific sender-receiver pairs are ordered (by send order)
  - ordered individual communications
  - good performance; 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!



https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html

# 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 $A_2$

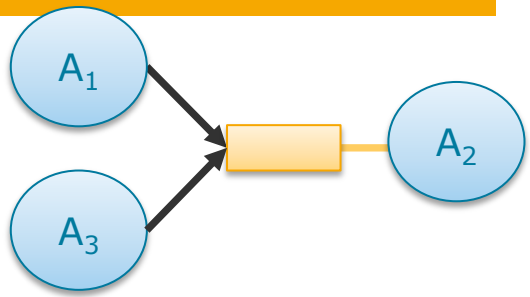
"If X is delivered…"
➢ No guaranteed delivery, i.e., messages may get lost and not arrive at $A_2$!

- ➢ 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 $A_3$



https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html

# 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!

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

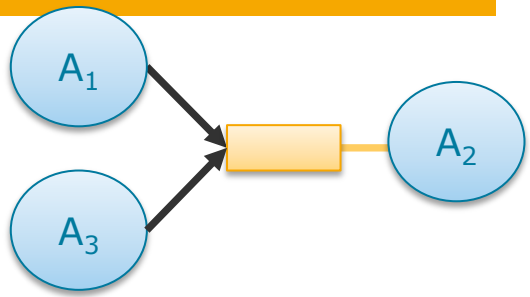
- Failure communication uses different channel:
  $A_1$ has child $A_2$
  $A_2$ sends $M_1$ to $A_1$
  $A_2$ fails causing failure message $M_2$ being send to $A_1$
  - $A_1$ may receive $M_1$ and $M_2$ in any order!

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



https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html

# 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!

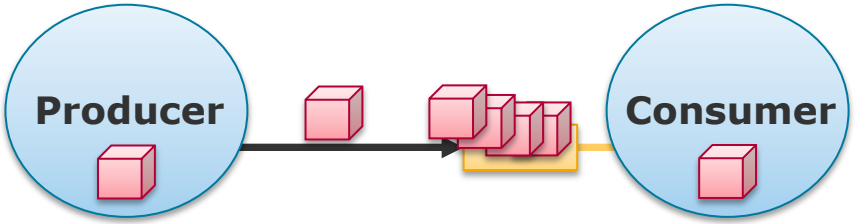


https://doc.akka.io/docs/akka/2.5/general/message-delivery-reliability.html#how-does-local-ordering-relate-to-network-ordering

# 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

# Pull vs. Push

```
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();
  }
}
```

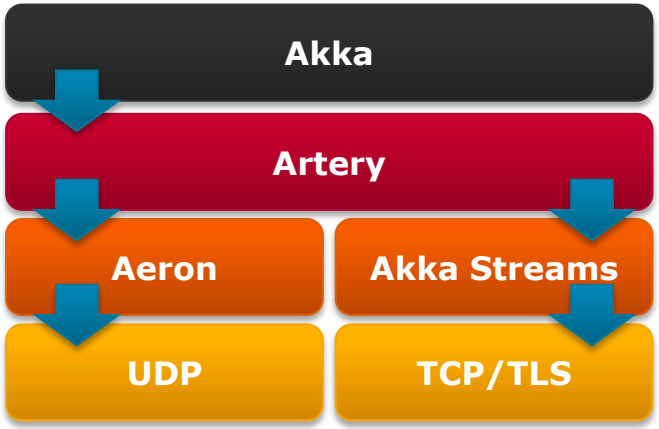
## Work Propagation

- **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

# Akka's Messaging System

## Artery

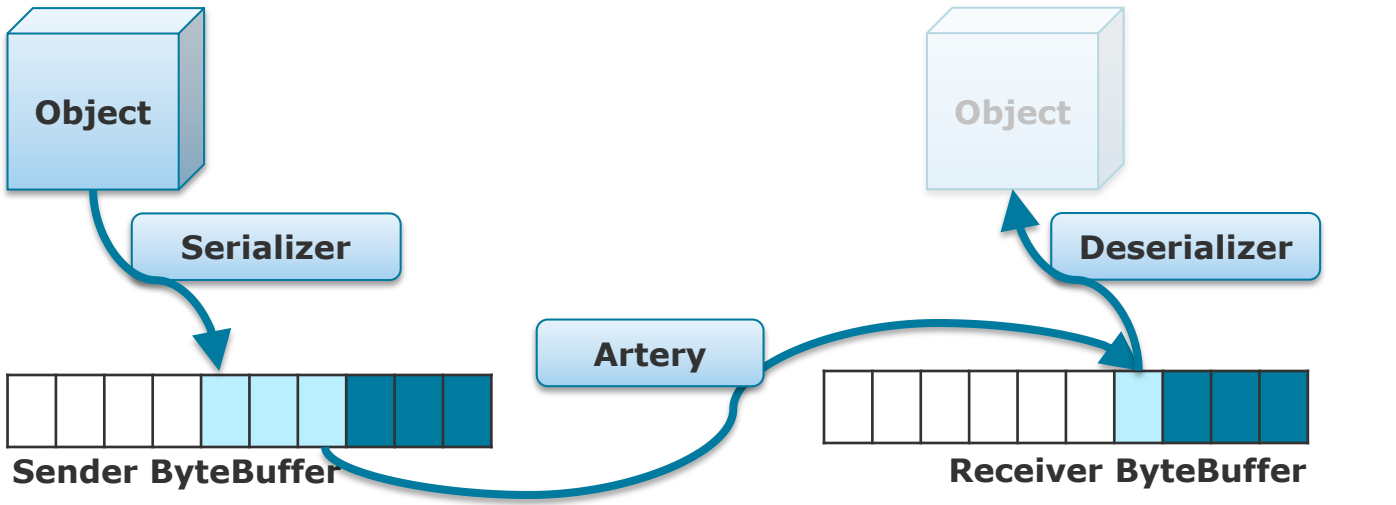
- High-performance, streaming-based messaging system
- Part of the Akka toolkit
- Compression of actor paths to reduce general message overhead
- Based on Aeron for UDP channels and Akka Streams for TCP/TLS channels



# Akka's Messaging System

## Artery

- Focused on high-throughput, low-latency communication
- Mostly allocation-free operation
- Support for faster serialization/deserialization using ByteBuffers directly



# Akka's Messaging System

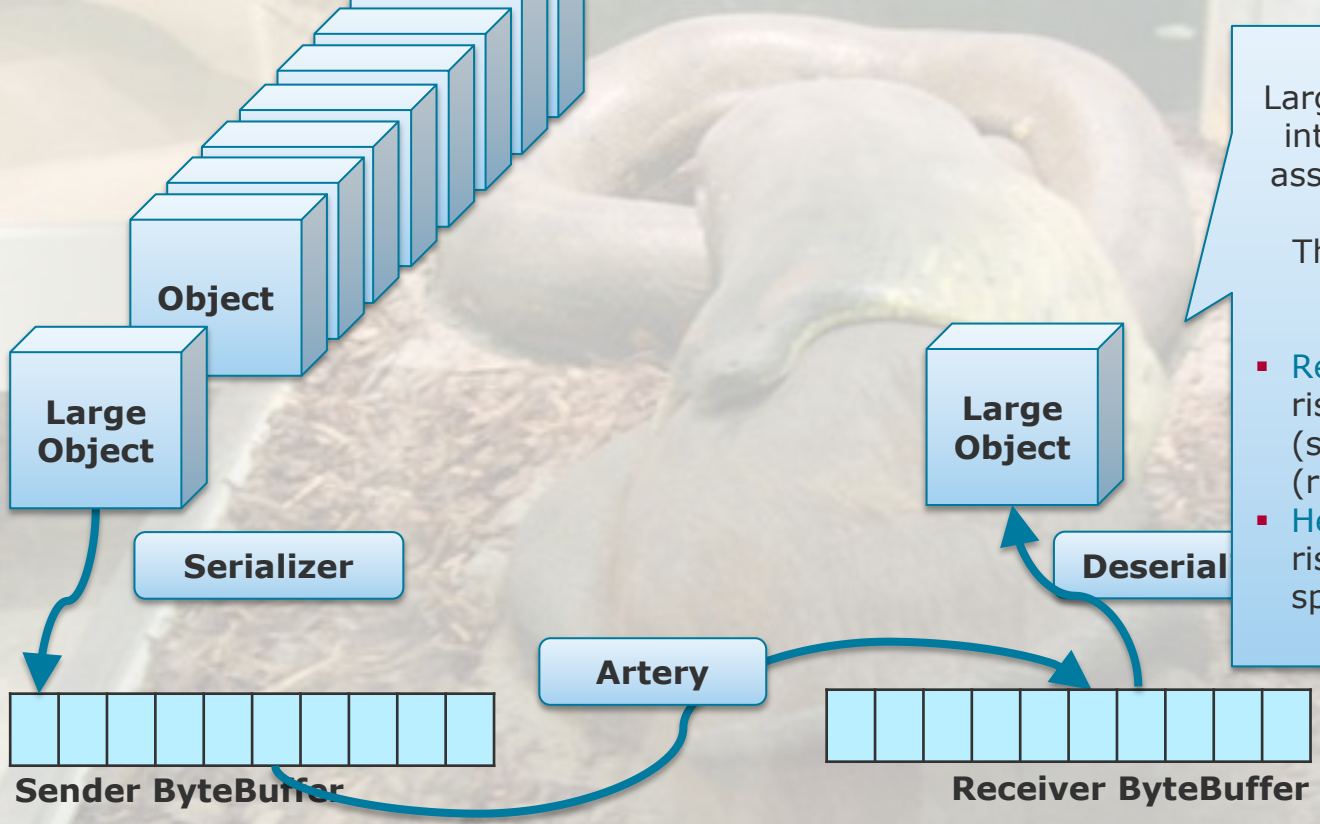
## Artery

- Focused on high-throughput, low-latency communication
- Mostly allocation-free operation
- Support for faster serialization/deserialization using ByteBuffers directly

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

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

https://petabridge.com/blog/large-messages-and-sockets-in-akkadotnet/




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**?

https://petabridge.com/blog/large-messages-and-sockets-in-akkadotnet/

# Large Messages

- 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 references in an Akka message when data is transferred.

# Artery's Large Message Channel

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
      large-message-destinations = [
        "/user/*/largeMessageProxy",
        "/user/**/largeMessageProxy"
      ]
    }
  }
}
```

or udp

hostname or IP

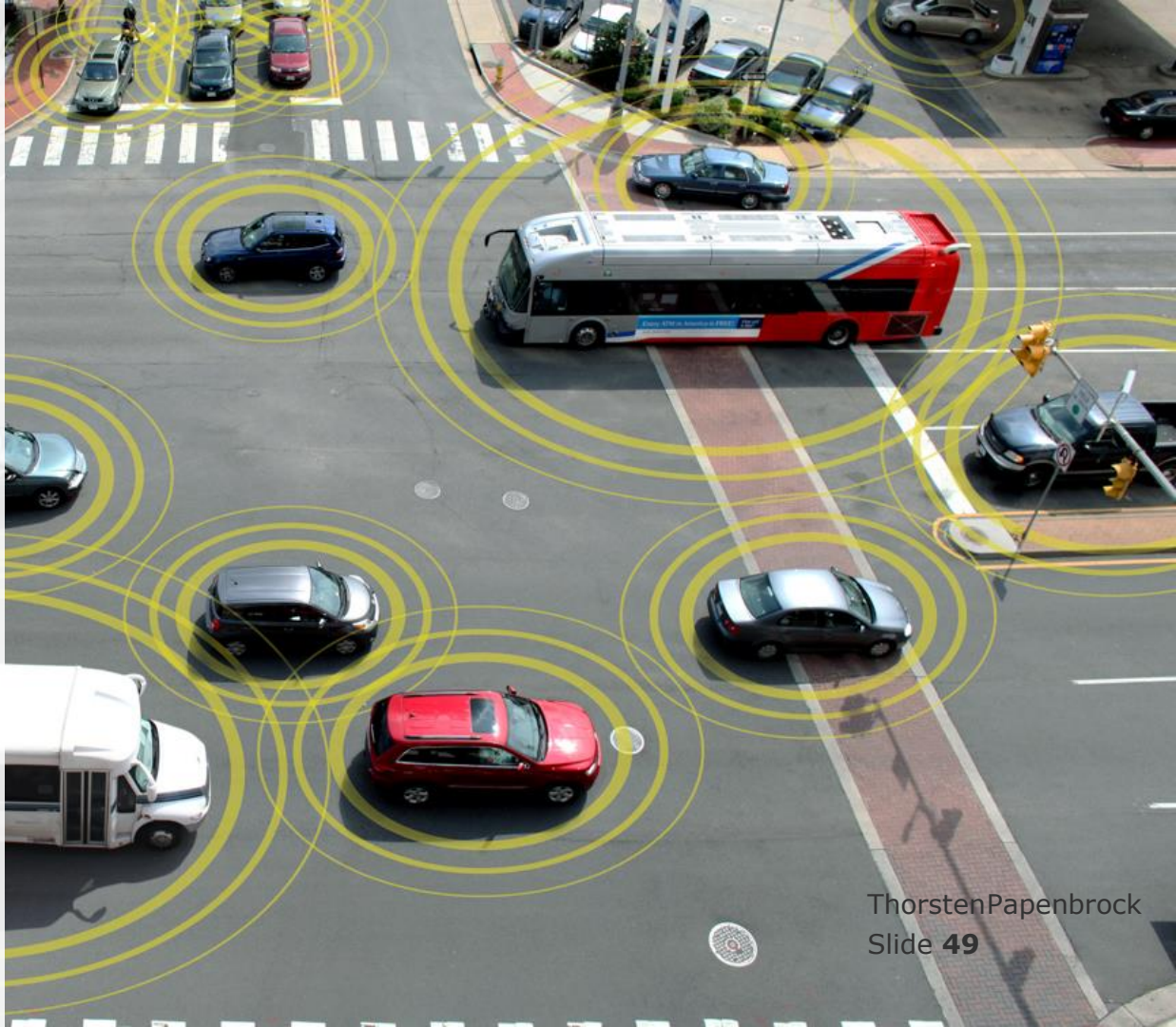
… of this actor system

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

Akka Actor Programming

# Hands-on

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

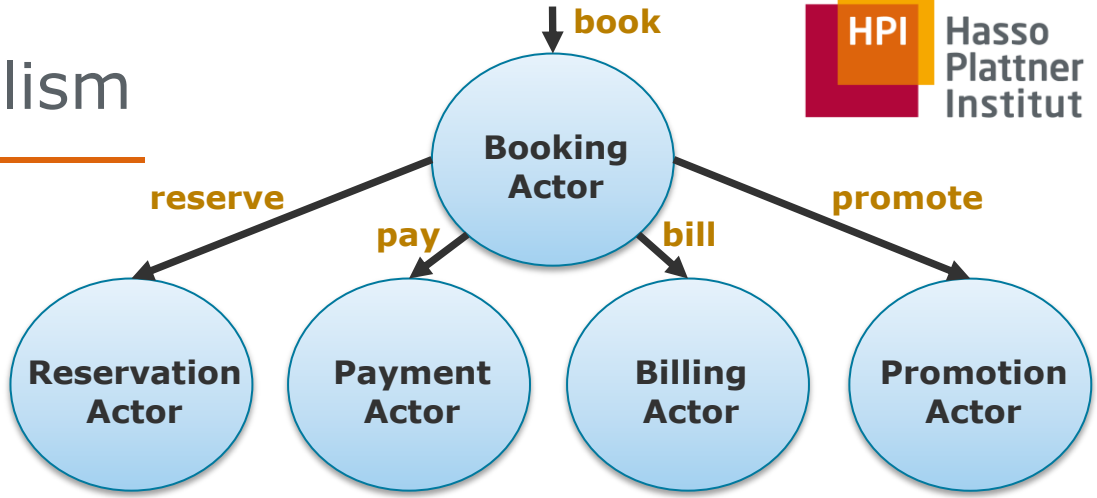


Parallelization

# Task- and Data-Parallelism

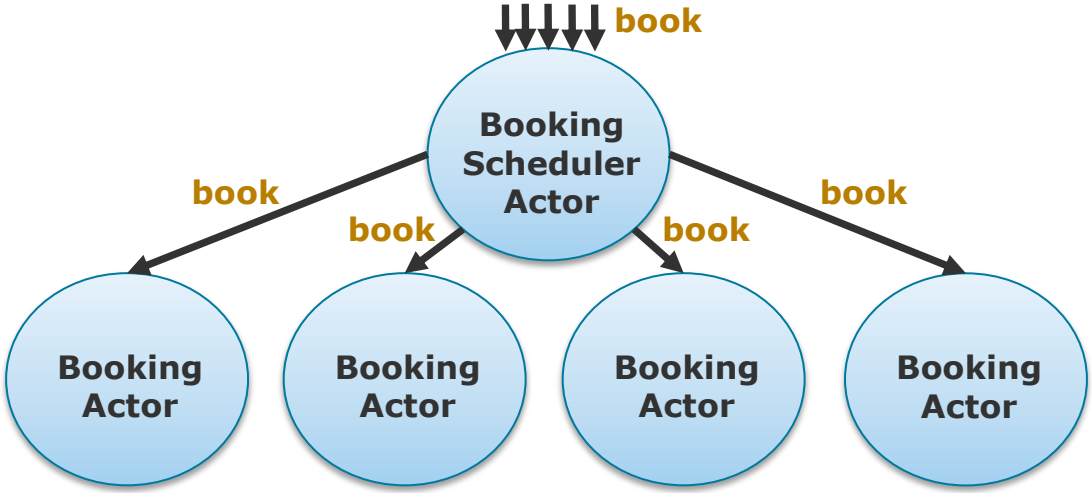
## Task-parallelism

- Distribute sub-tasks to different actors




## Data-parallelism

- Distribute chunks of data to different actors



# Scheduler

## 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

Push Propagation!

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

# Scheduler

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

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

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

Scala world: All objects are immutable!

Logic defines the worker to be chosen.

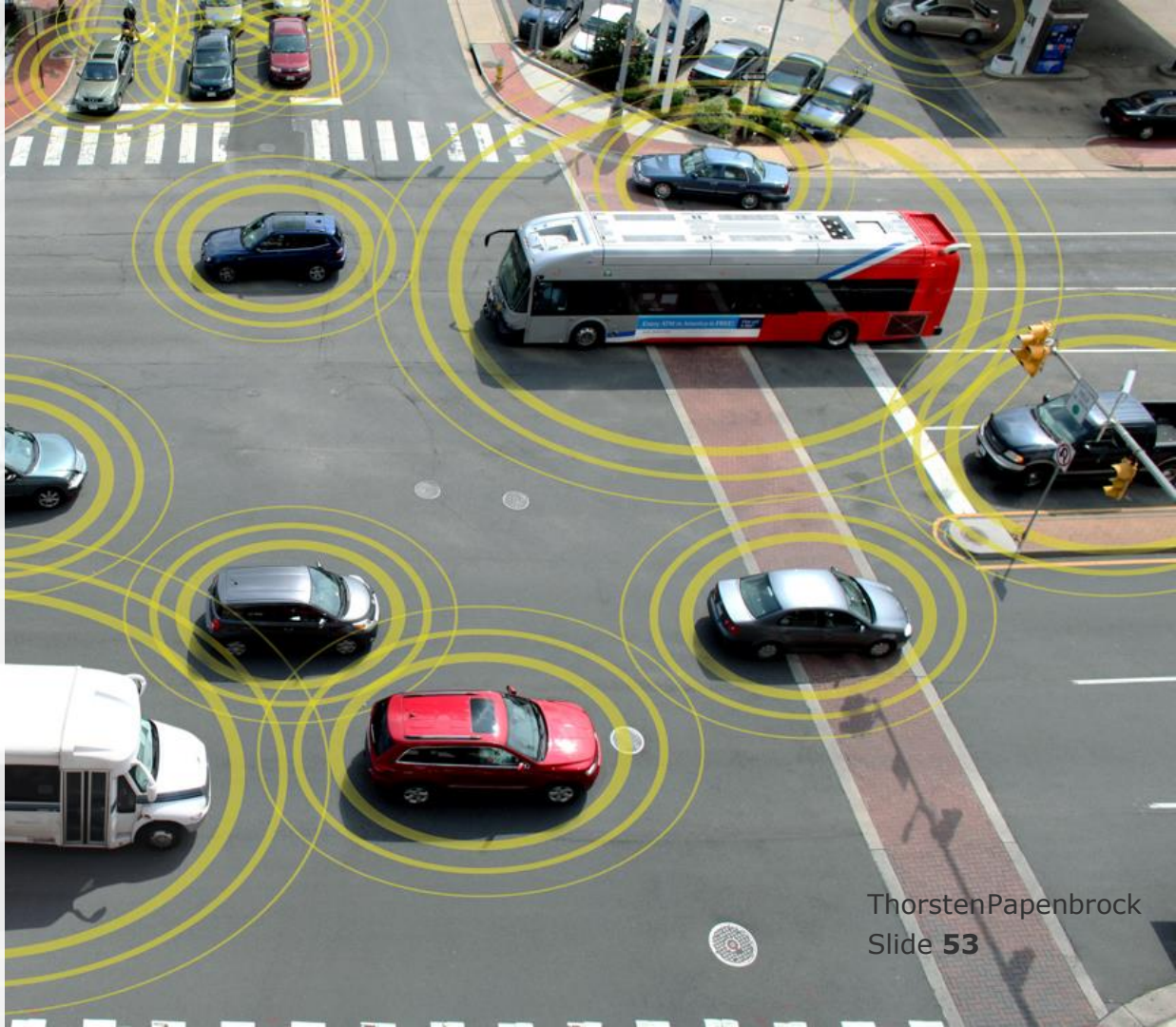
## Task Scheduling

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

Akka Actor Programming

# Hands-on

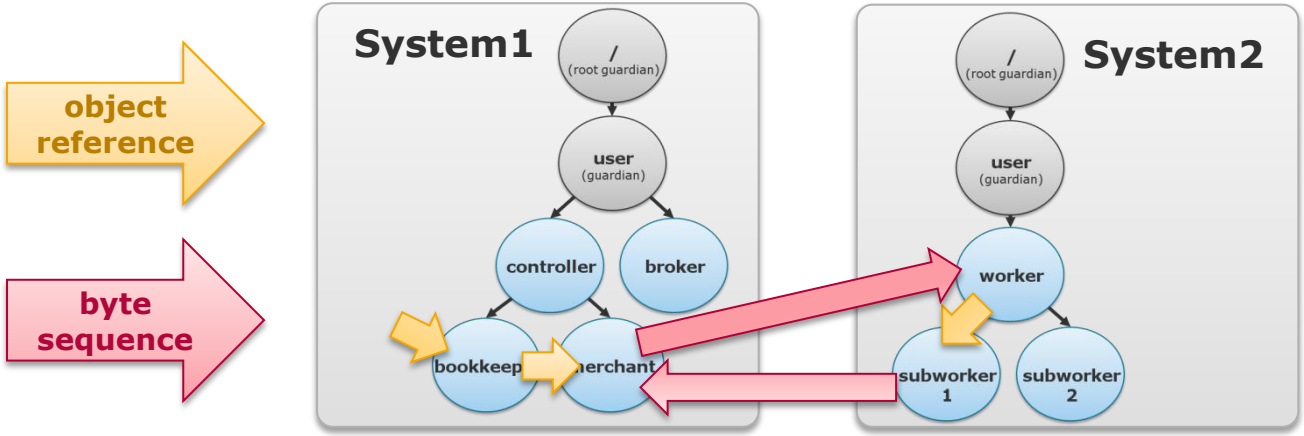
- Actor Model (Recap)
- Basic Concepts
- Runtime Architecture
- Demo
- Messaging
- Parallelization
- **Remoting**
- Clustering
- Patterns
- Homework



# Serialization

## 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)



# Serialization

application.conf

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

```
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

# Serialization

Usually no no-arg constructor needed.

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-seriablizable 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.

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

No-arg constructor needed!

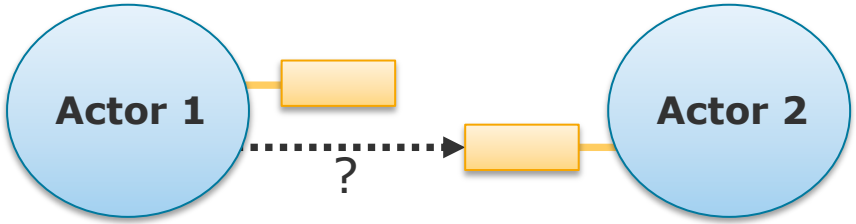
A Java **Kryo** class must do the following:

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

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

# Actor Lookup

**ActorRefs** serve as pointers to local/remote actors.



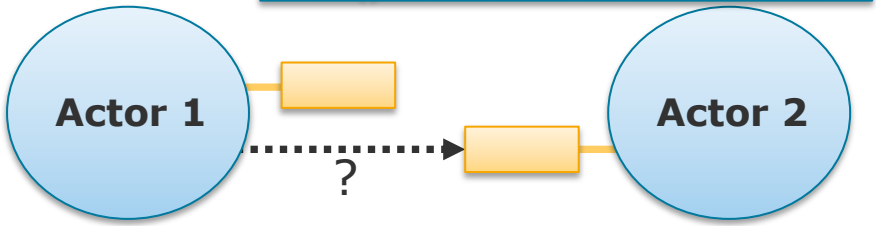

```
public class Master extends AbstractLoggingActor {
  private ActorRef worker;

  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();
  }
}
```

1. A factory pattern
3.

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.

# Actor Lookup

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




```
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.

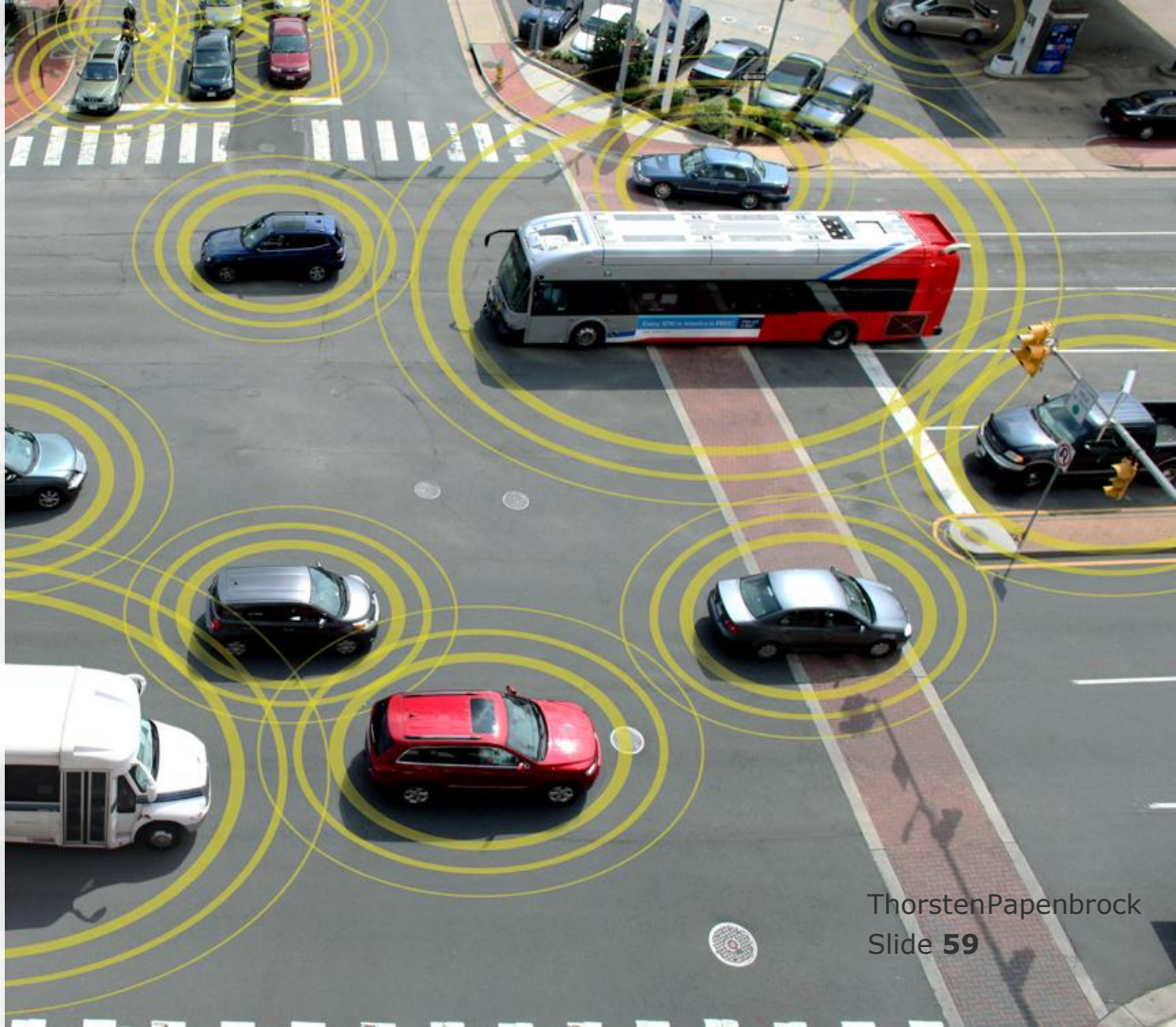
URL:
"akka.tcp://MyActorSystem@localhost:7877/user/worker"

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.

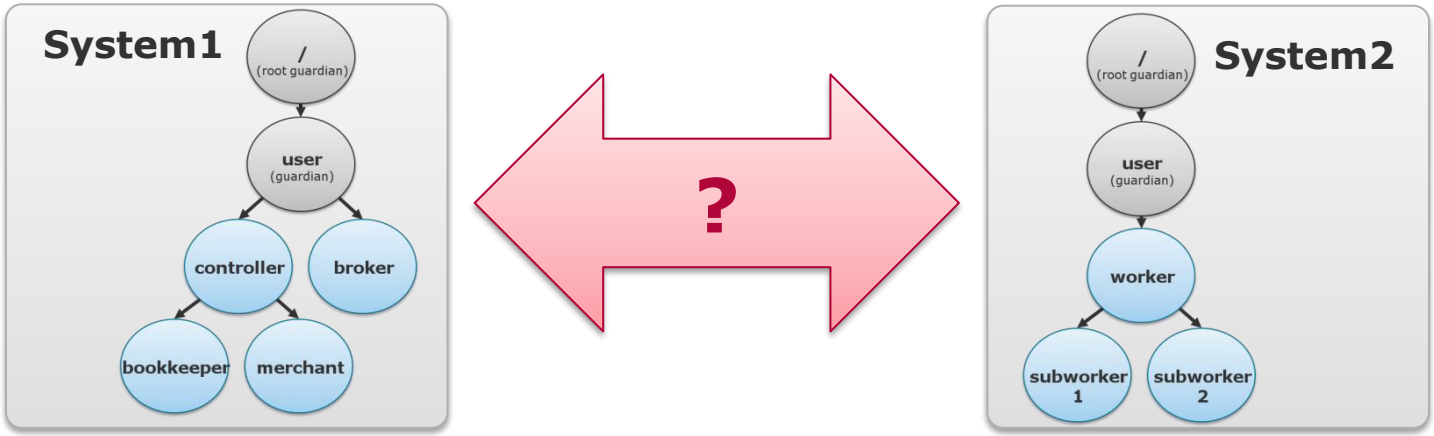
Akka Actor Programming

# Hands-on

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



# 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!)

# 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

# Dependency

**Clustering capabilities**
cluster membership, singletons, publish/subscribe, cluster client, …

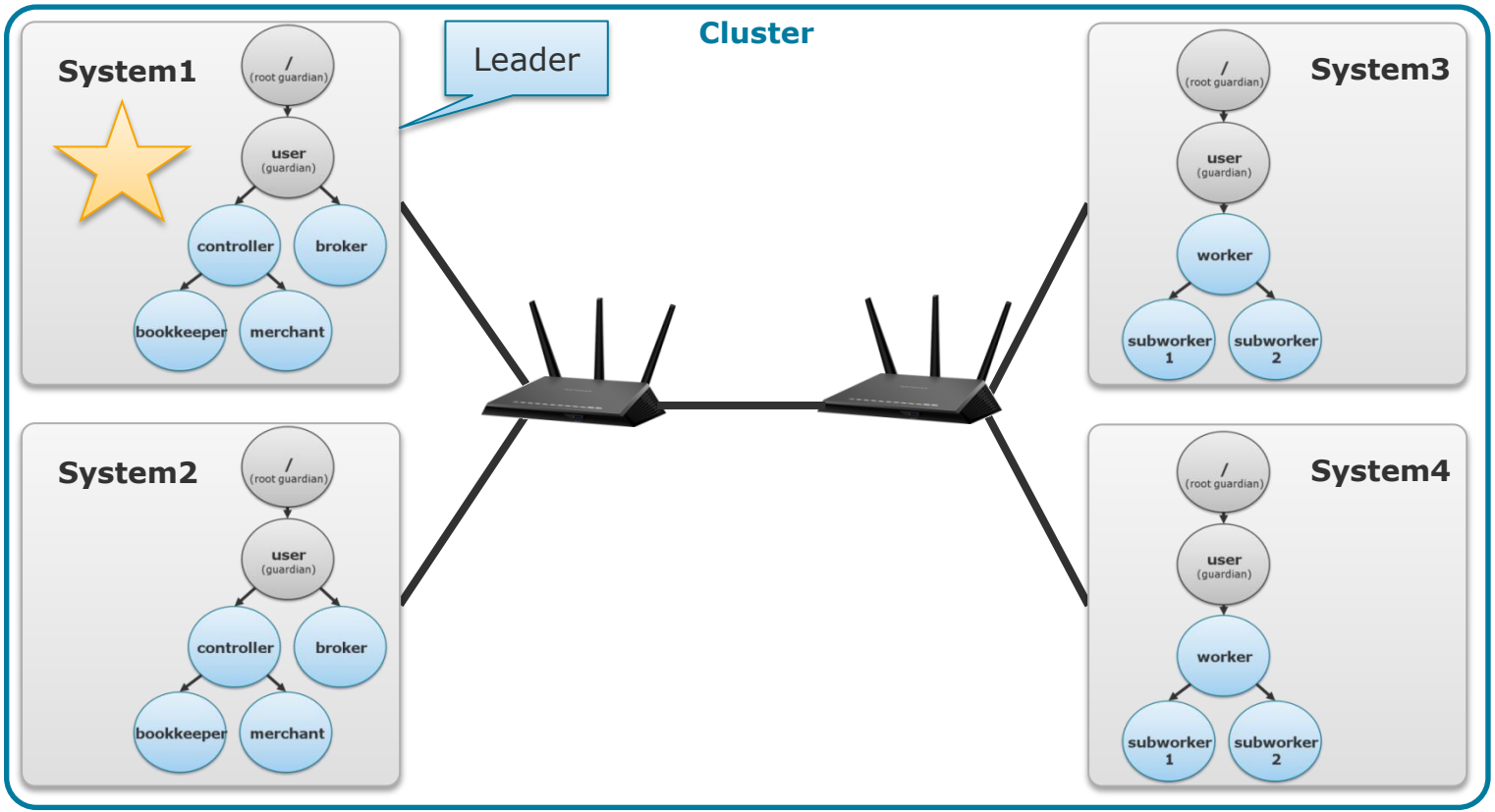
**Metrics collection**
CPU load, memory consumption, …

**Transparent actors**
logical references, distributed/persisted state, …

## Maven – pom.xml

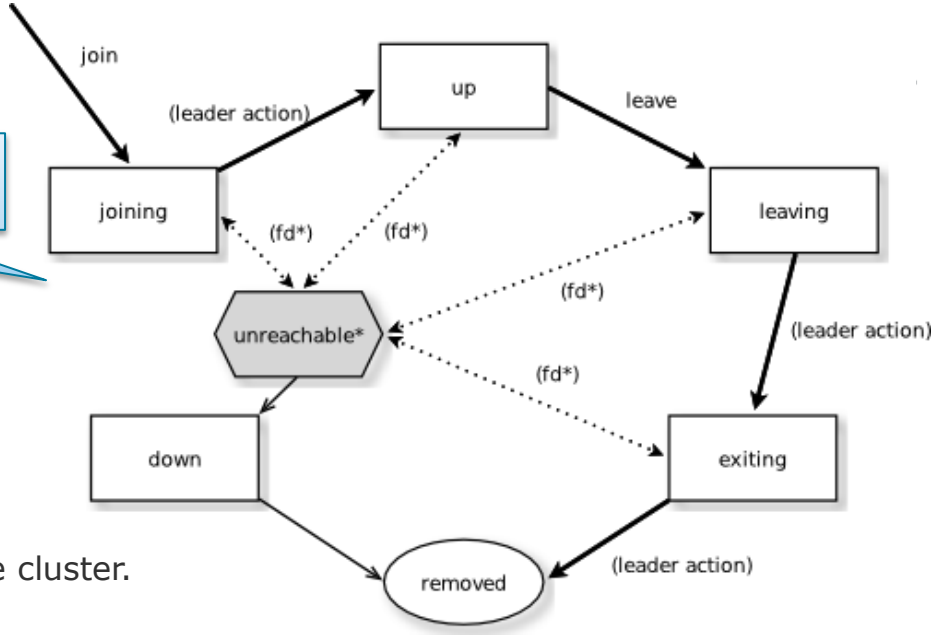
```
<dependencies>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-actor_${scala.version}</artifactId>
    <version>2.5.3</version>
  </dependency>
  <dependency>
    <groupId>com.typesafe.akka</groupId>
    <artifactId>akka-remote_${scala.version}</artifactId>
    <version>2.5.3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

# Cluster


Cluster
Leader
System1
/
(root guardian)
user
(guardian)
controller
broker
bookkeeper
merchant
System2
/
(root guardian)
user
(guardian)
controller
broker
bookkeeper
merchant
System3
/
(root guardian)
user
(guardian)
worker
subworker
1
subworker
2
System4
/
(root guardian)
user
(guardian)
worker
subworker
1
subworker
2

# 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 gossipping[1] to propagate cluster changes.
- Elects the eldest node in a cluster as "leader" of the cluster.
- Leader decisions[2]:
  - state changes to "up", "exiting", and "removed" for members of the cluster
- Member decisions[2]:
  - state changes to "joining", "leaving", and (via local failure detector) "unreachable"

[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

# Configuration

## application.conf

```
akka {
  actor {
    provider = cluster
    […]
  }
  remote {
    artery {
      […]
    }
  }
  cluster {
    min-nr-of-members = 3
    role {
      master.min-nr-of-members = 1
      slave.min-nr-of-members = 2
    }
  }
}
```

instead of local or remote

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

# Startup

```
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 = [slave]\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.

# Startup

```
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());
```

# Startup

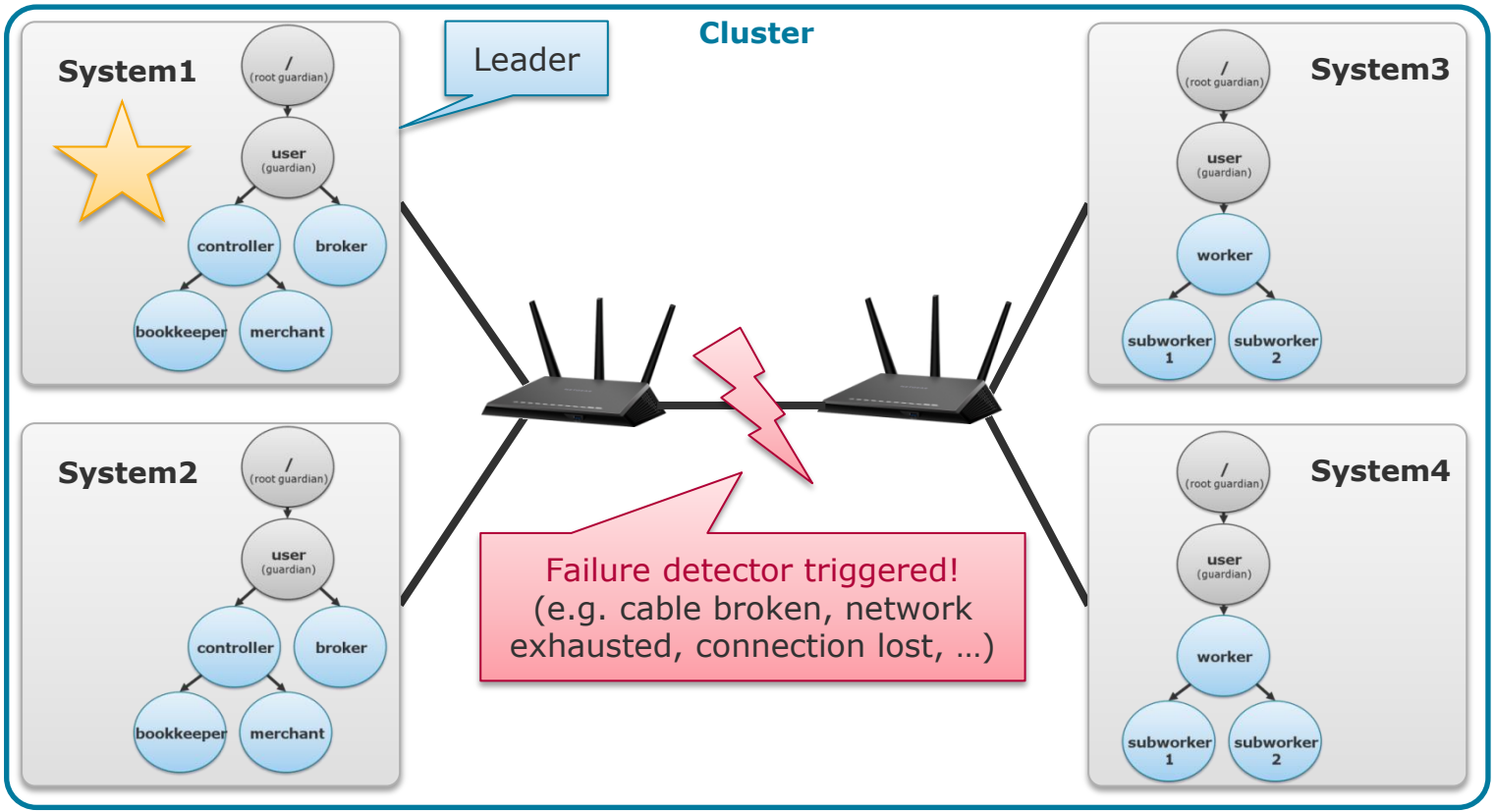
```
public static void start(String appName, String host, int port, String seedhost, int seedport) {
  […]
  Cluster.get(system).registerOnMemberRemoved(new Runnable() {
    @Override
    public void run() {
      system.terminate();
      new Thread() {
        @Override
        public void run() {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

Run when this node has been "removed".

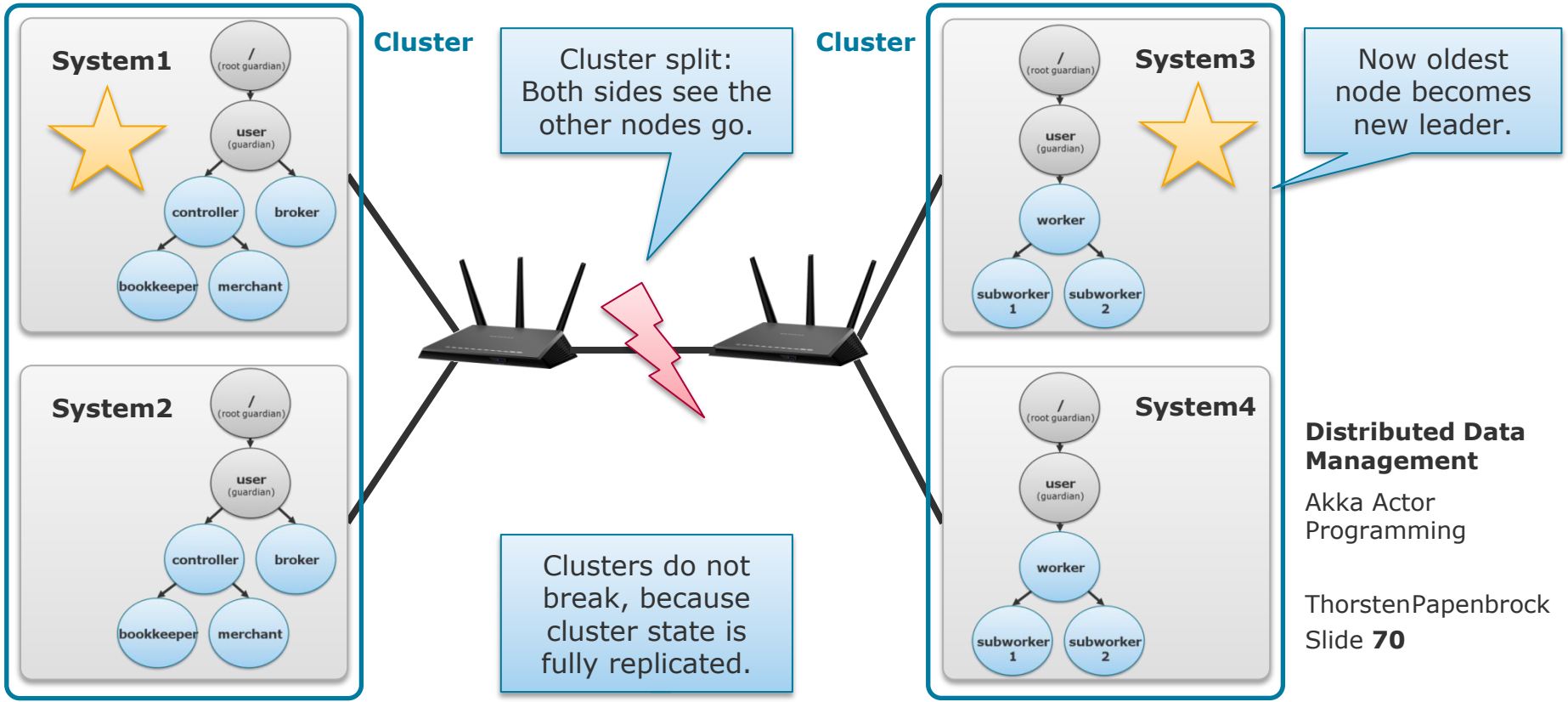
Terminate the ActorSystem.

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

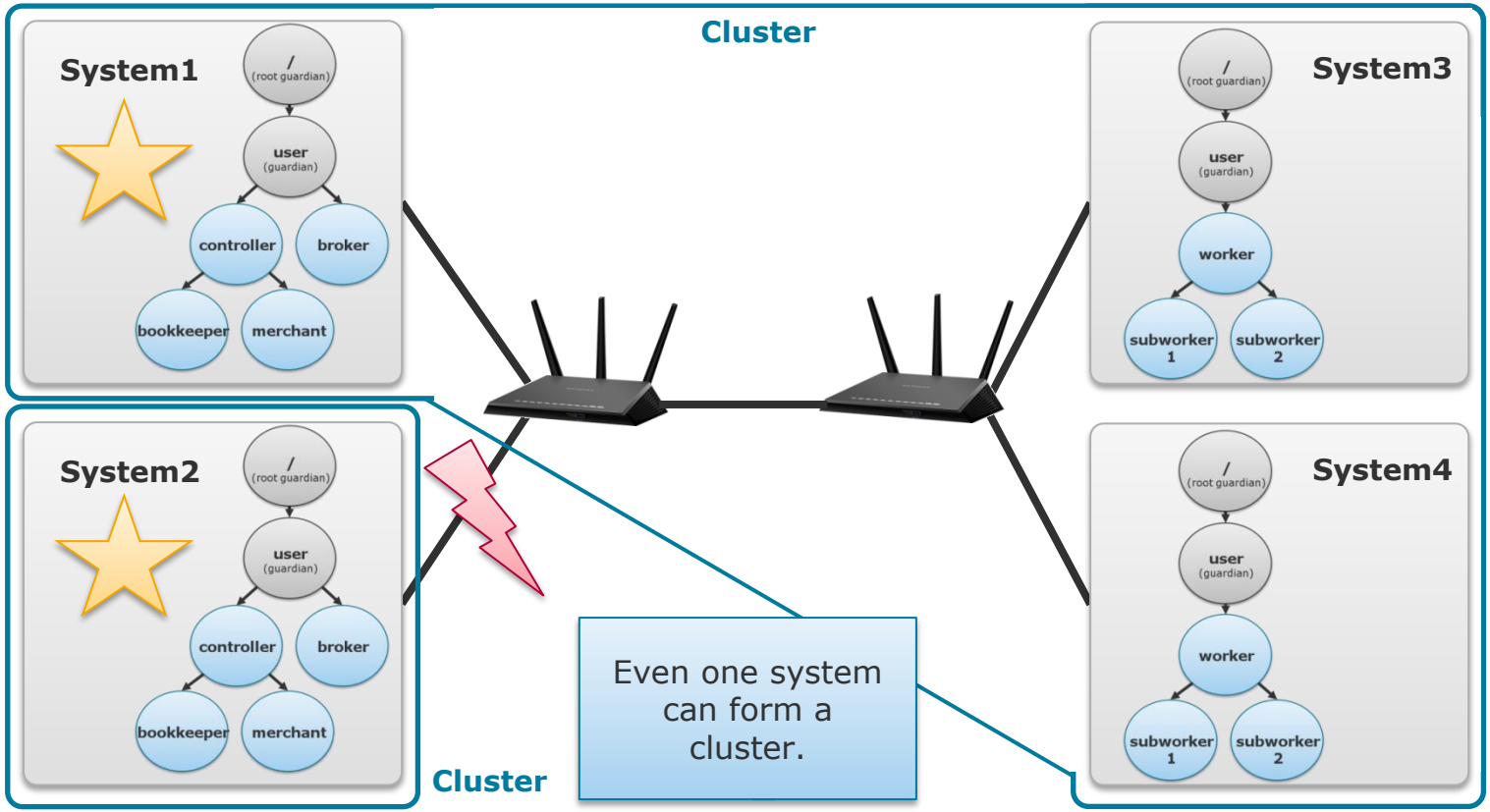
# Cluster Partitioning



# Cluster Partitioning



# Cluster Partitioning



# 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

```
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

# 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

# Cluster-Aware Scheduler

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

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

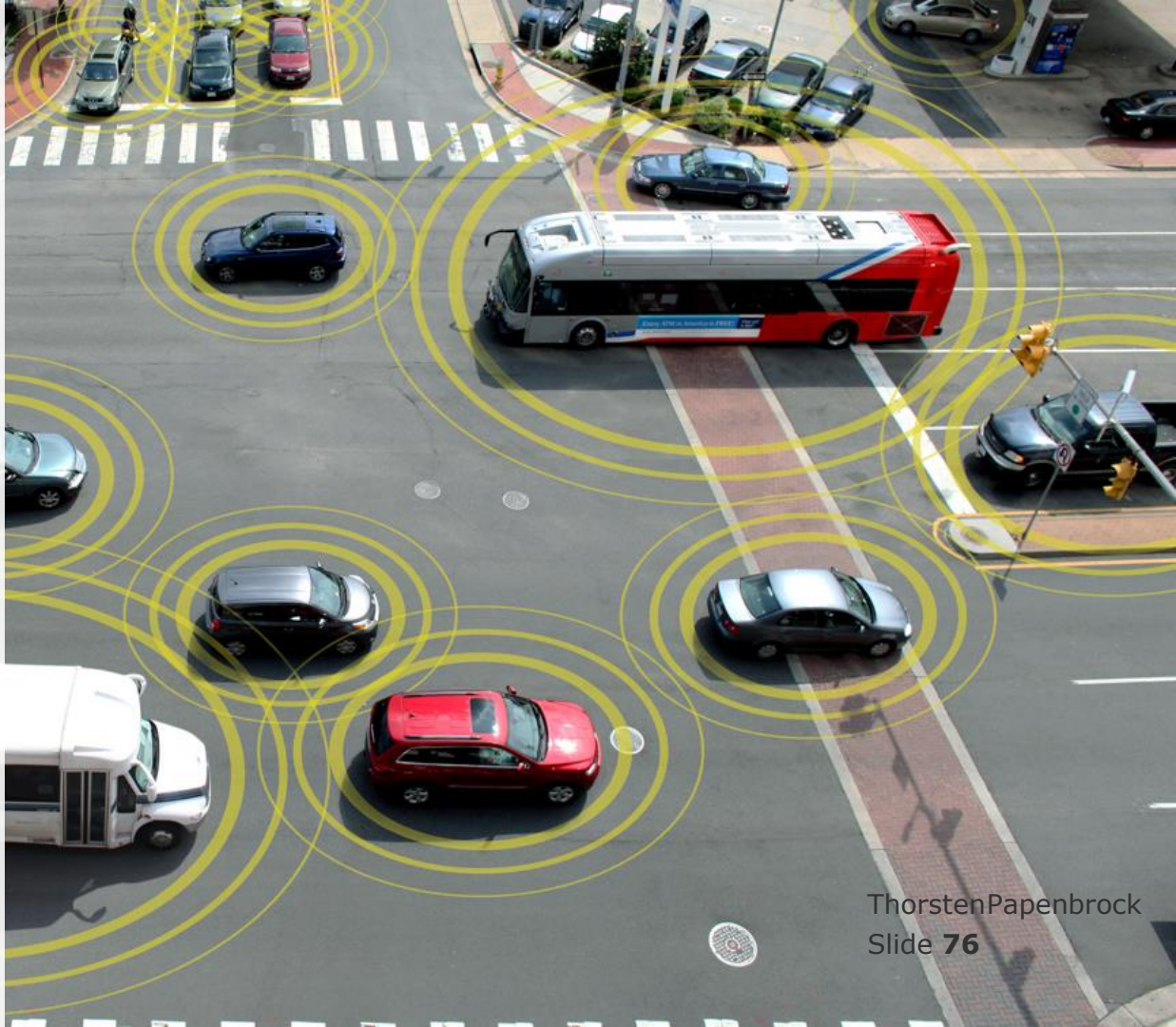
## application.conf

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

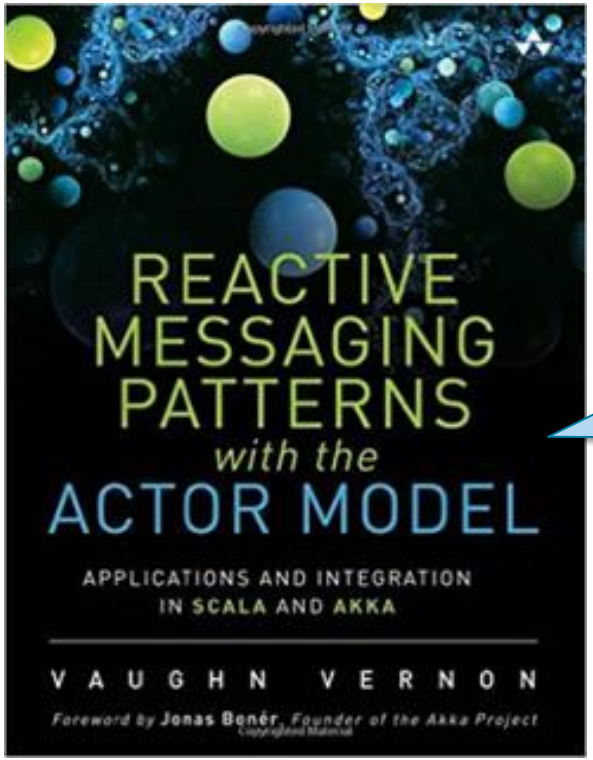
Akka Actor Programming

# Hands-on

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



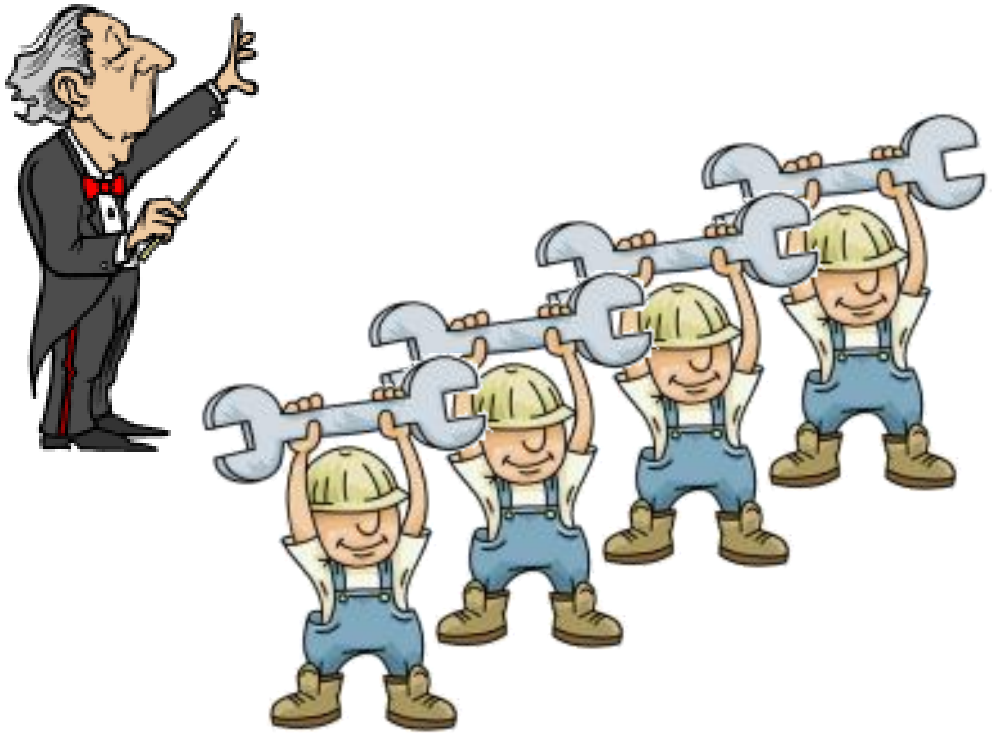
# Actor Programming Patterns




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

# Master/Worker

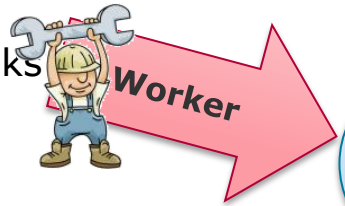


Patterns

# Master/Worker

## Task-parallelism

- Distribute sub-tasks to different actors

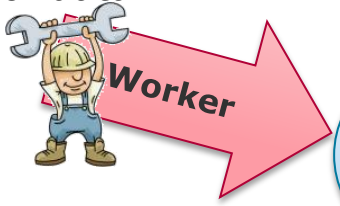







## Data-parallelism

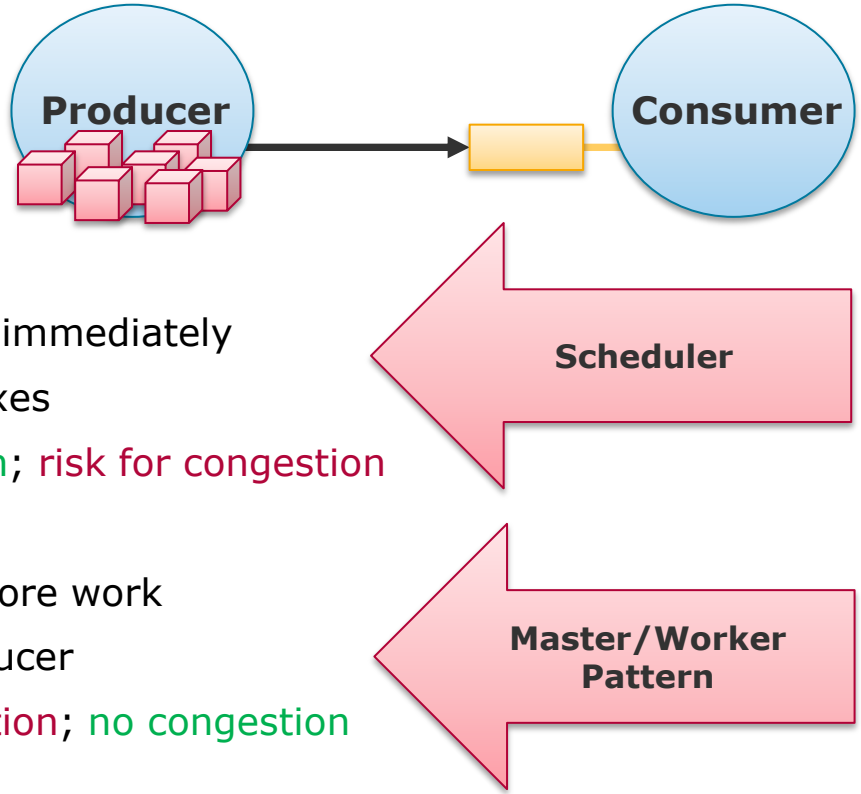
- Distribute chunks of data to different actors




# 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



# Master/Worker

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!

# 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

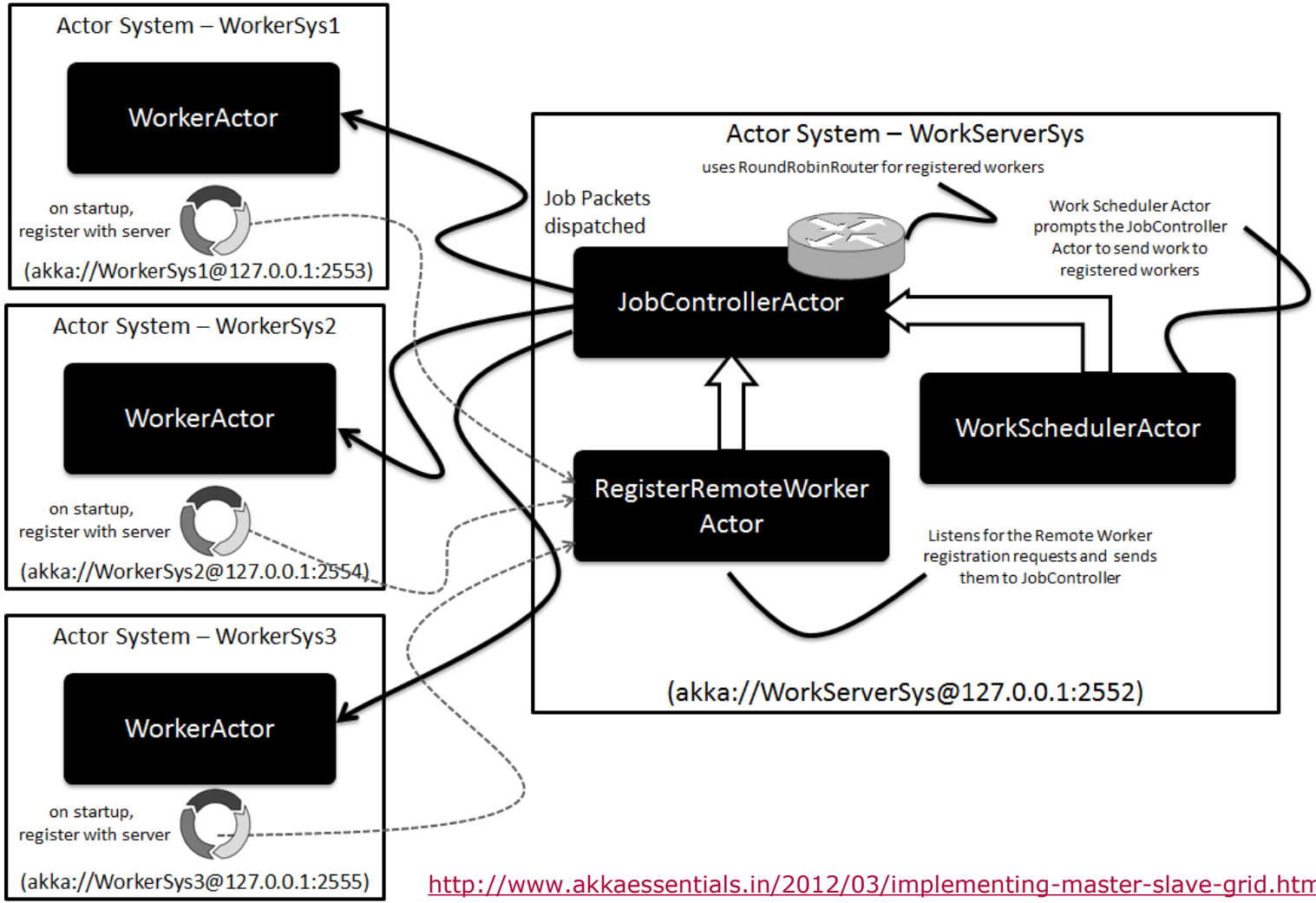
Split and assign to workers

Collect and send new work

Patterns

# Master/ Worker

## Example

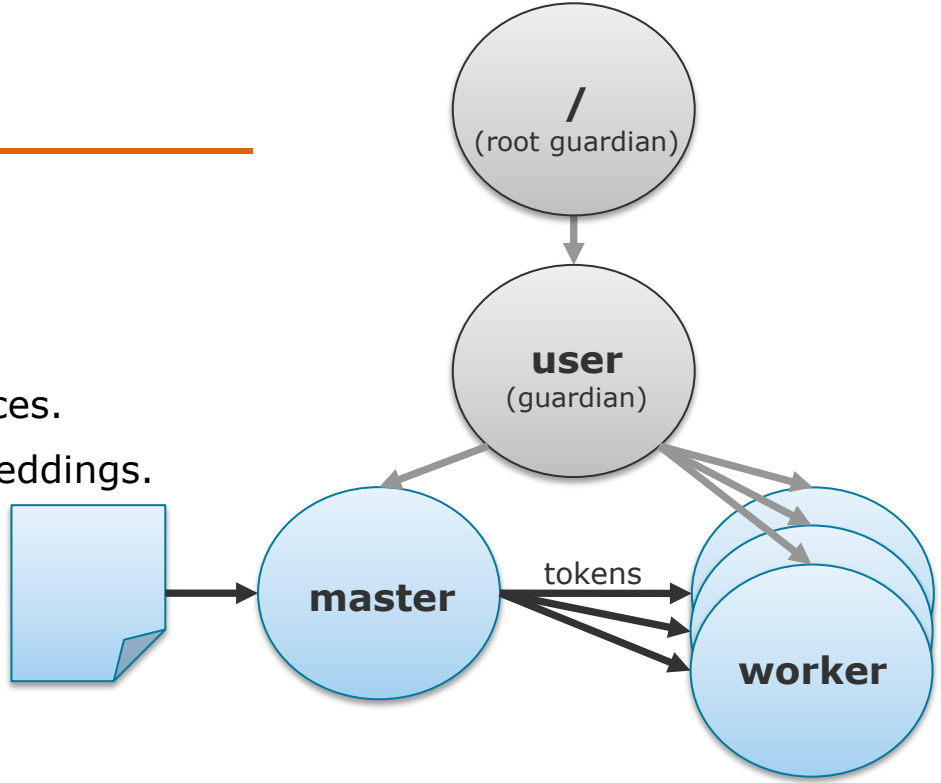



http://www.akkaessentials.in/2012/03/implementing-master-slave-grid.html

# Master/Worker

"The eager producers"

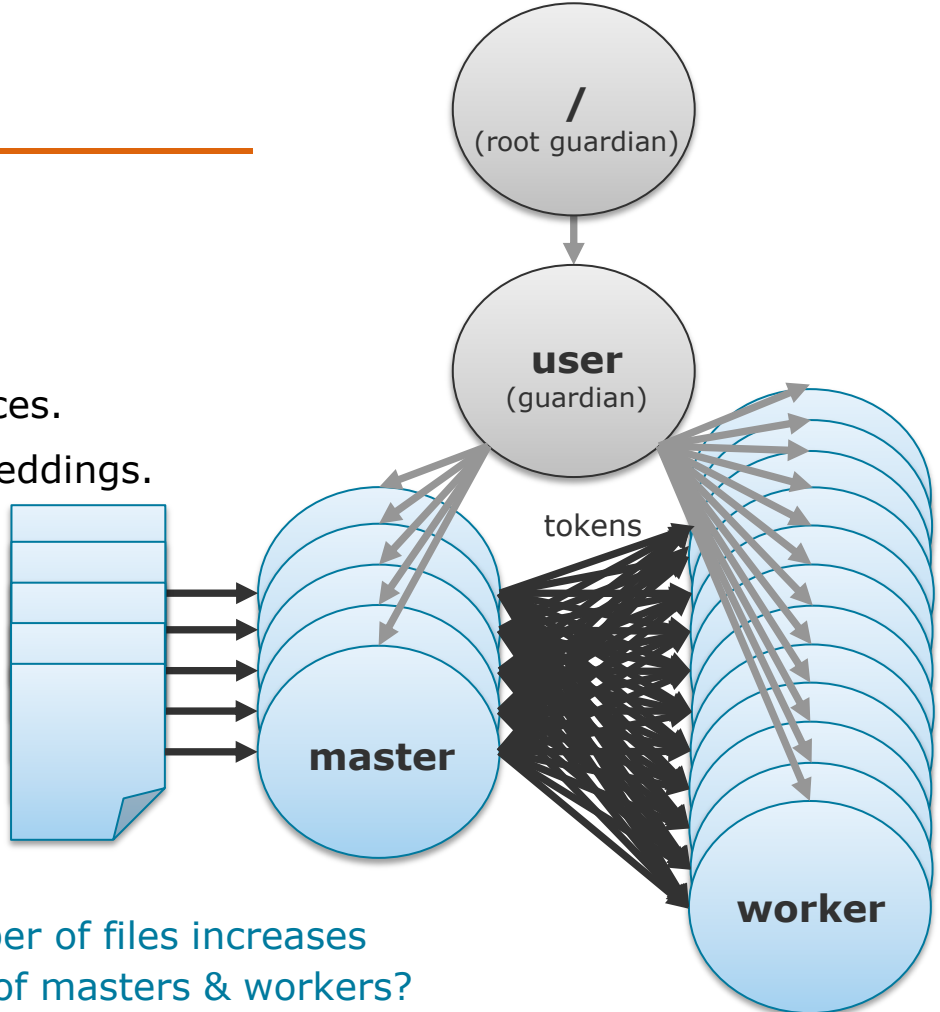
- Task:
  1. Read a file.
  2. Tokenize the sentences.
  3. Calculate token embeddings.
- Push-Propagation:
  - Each input file is read and tokenized by one master.
  - Each token range is processed by one worker.
- Works great for **one** input file!



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



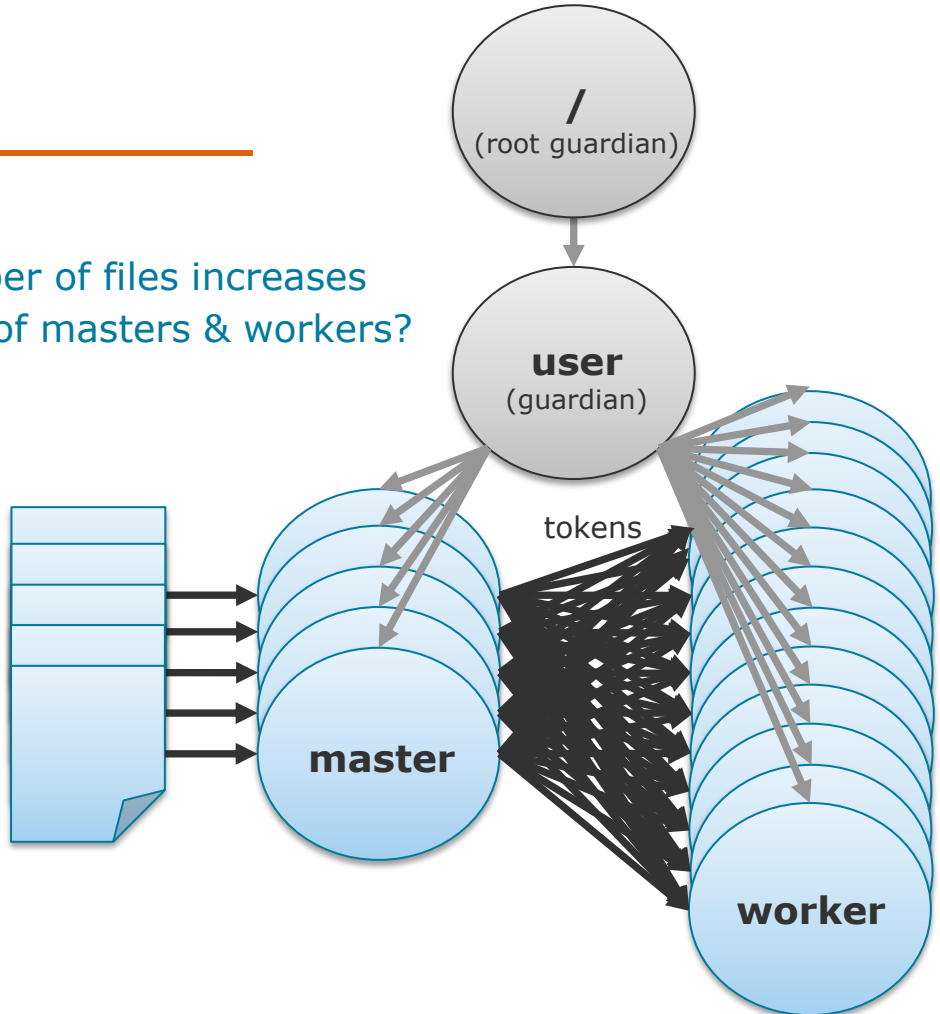
# Master/Worker

"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.



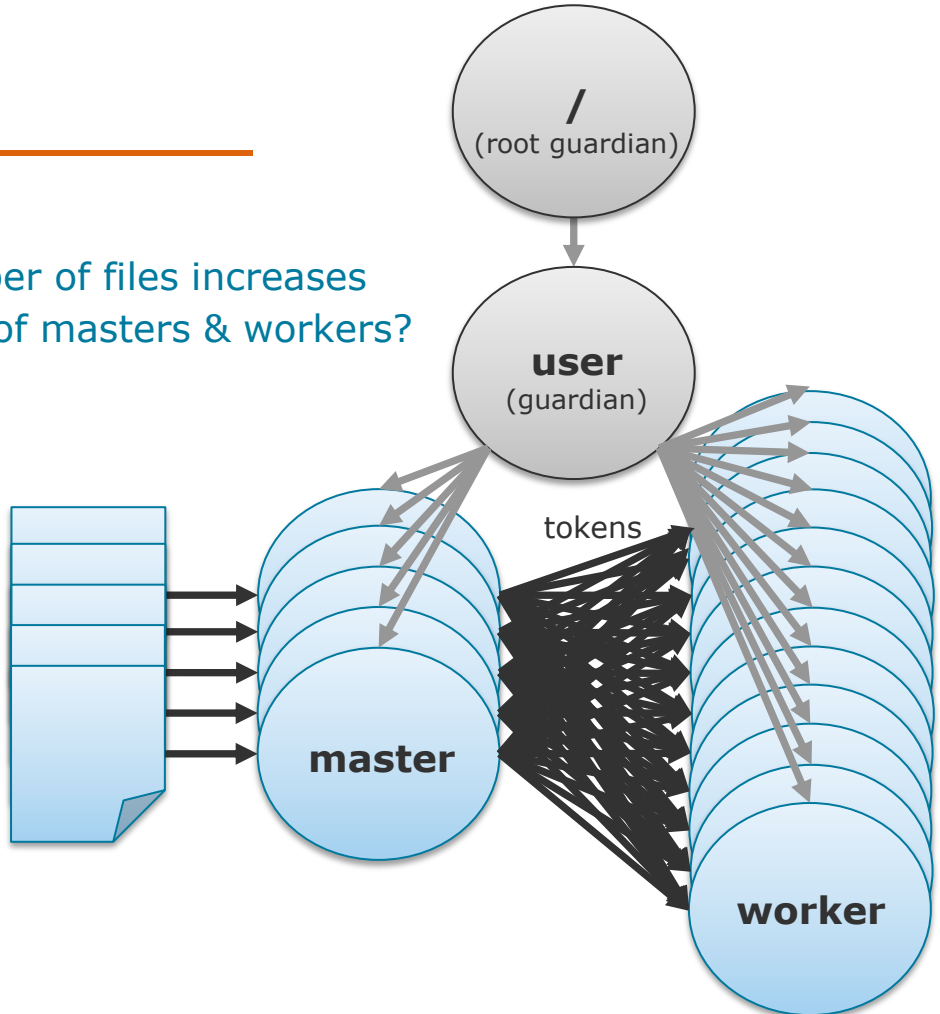
# Master/Worker

"The eager producers"

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

Solutions:

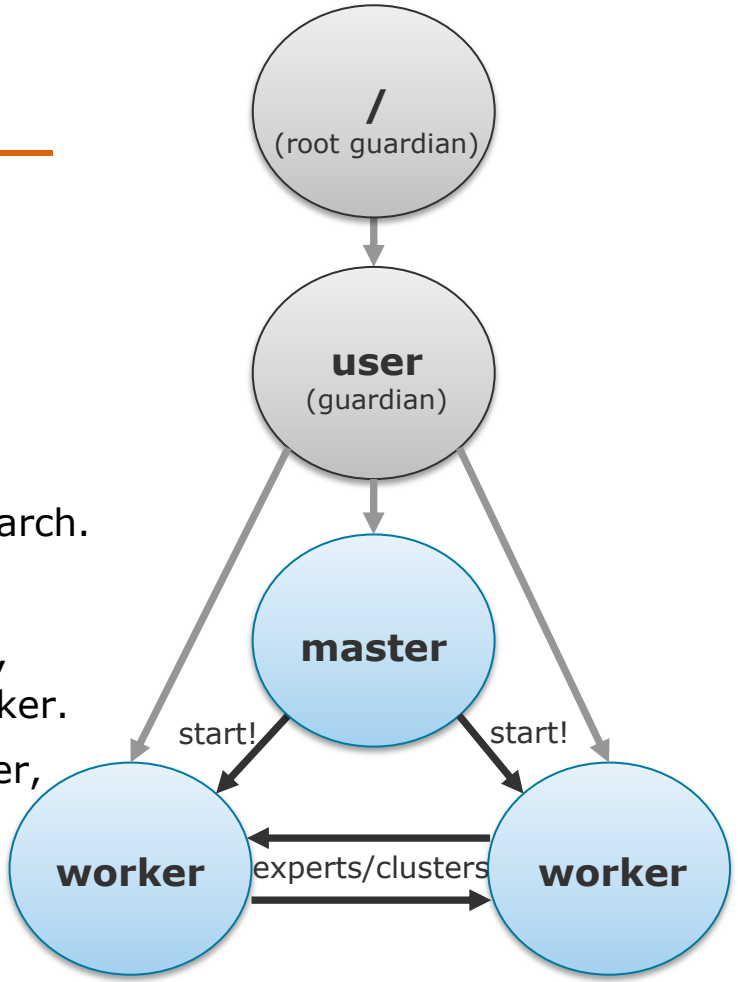
- Pull-Propagation:
  Pause long running tasks and free threads.
- File limit:
  Control the number of actors with long running tasks
  (in particular fewer than number of cores).



# 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?

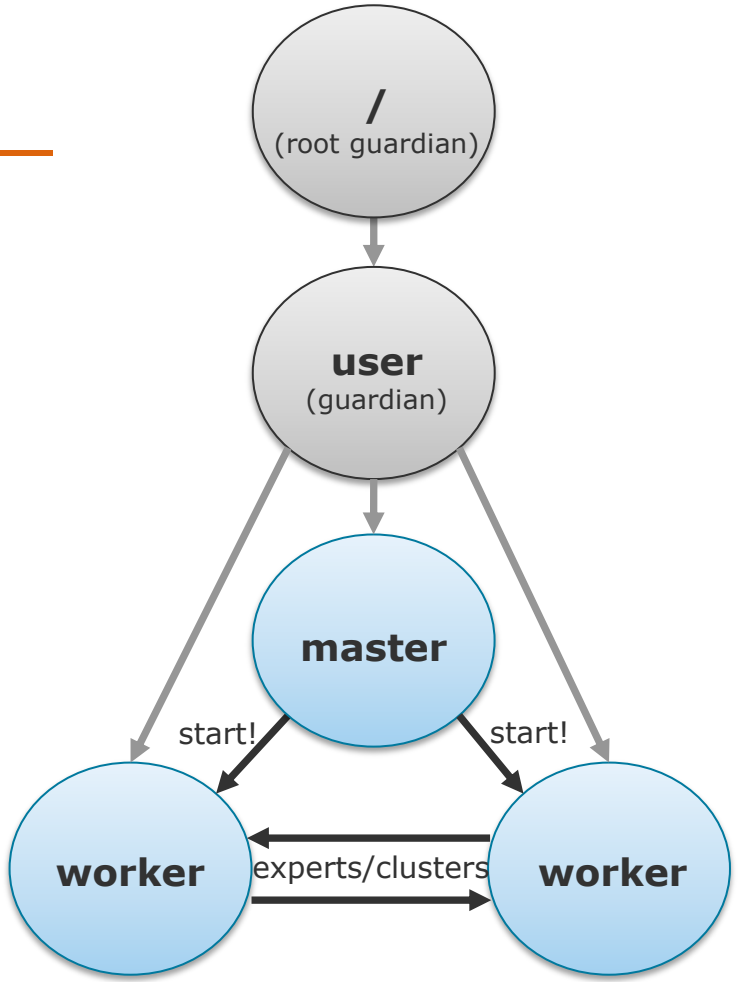


# Master/Worker

"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.

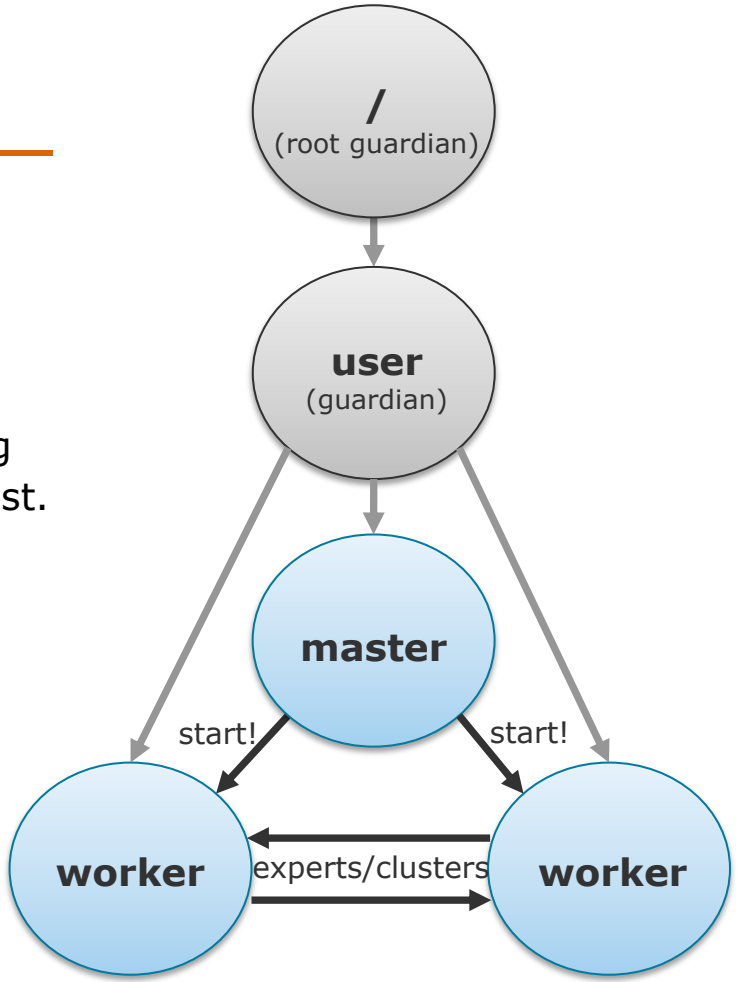


# Master/Worker

"The non-reactive workers"

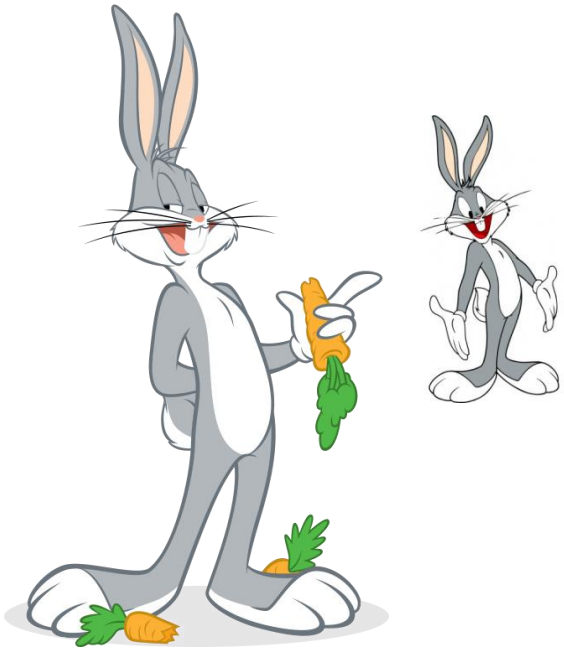
- What could be a problem here?

➢ Solution:
Proxy actors that aggregate incoming messages and deliver them on request.



# Proxy

# Proxy



## 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 where talking to the real actor!

# Proxy

## Example: Simple Proxy

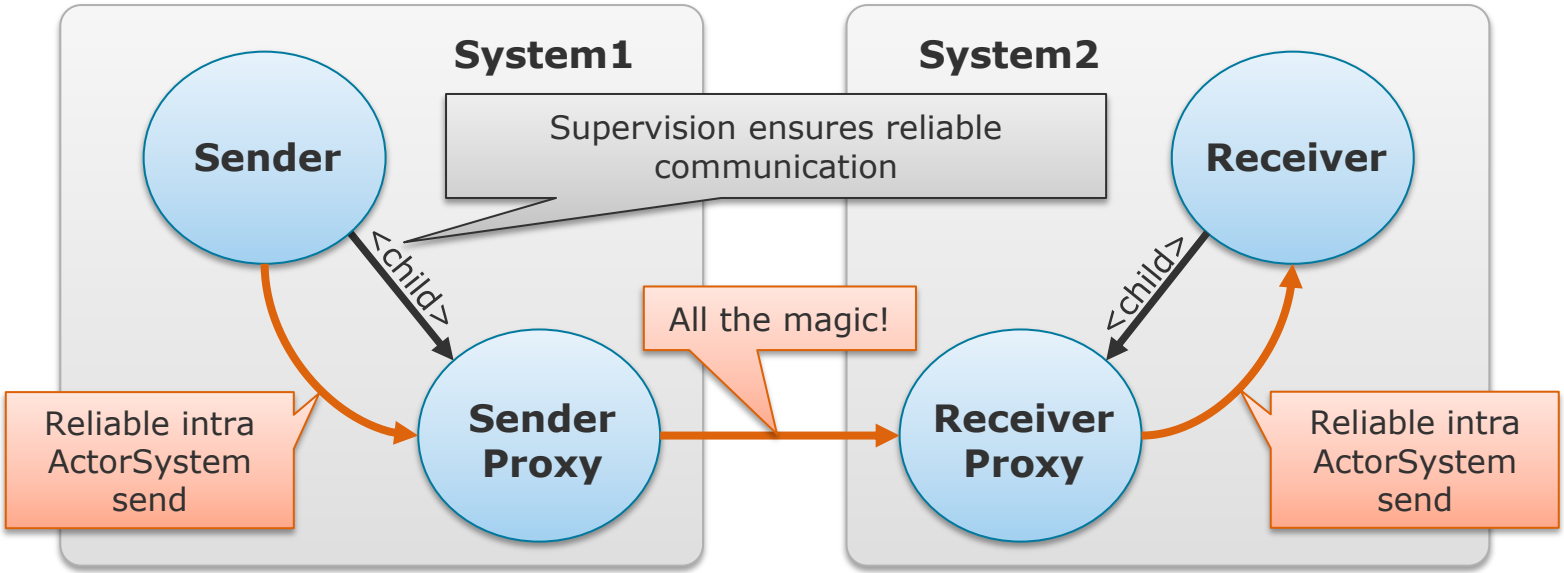
- Delegate a new communication to a proxy.
- If the communication returns a result, the proxy reports it to the real actor.

```
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();
  }
}
```

# Proxy

## Example: Reliable Communication Proxy

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




https://doc.akka.io/docs/akka/2.2.4/contrib/reliable-proxy.html#reliable-proxy

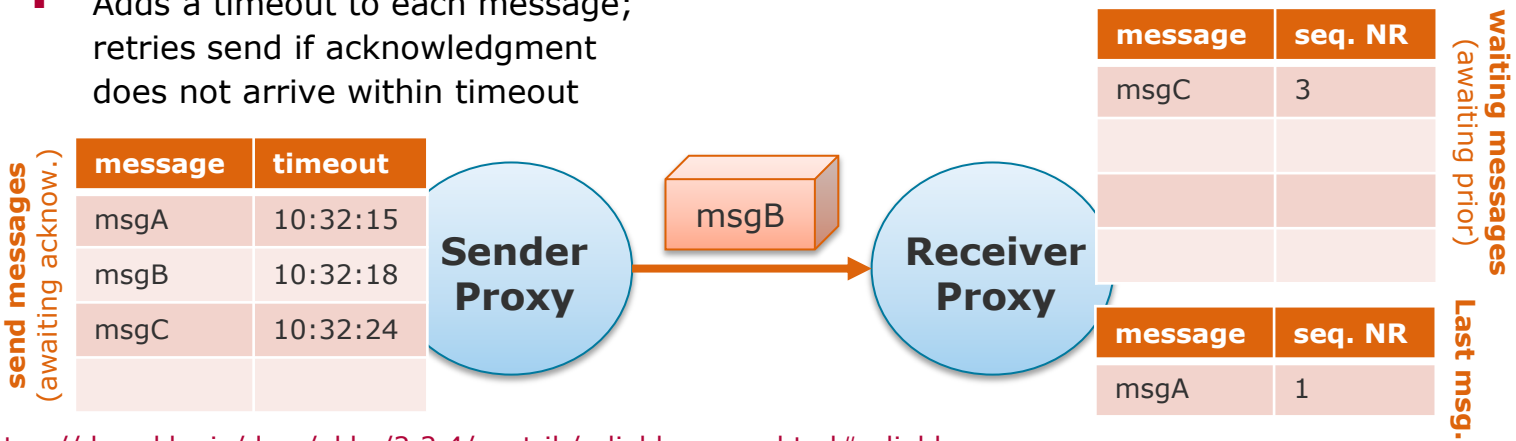
## Example: Reliable Communication Proxy

### 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




https://doc.akka.io/docs/akka/2.2.4/contrib/reliable-proxy.html#reliable-proxy

# Proxy

## Example: Reliable Communication Proxy

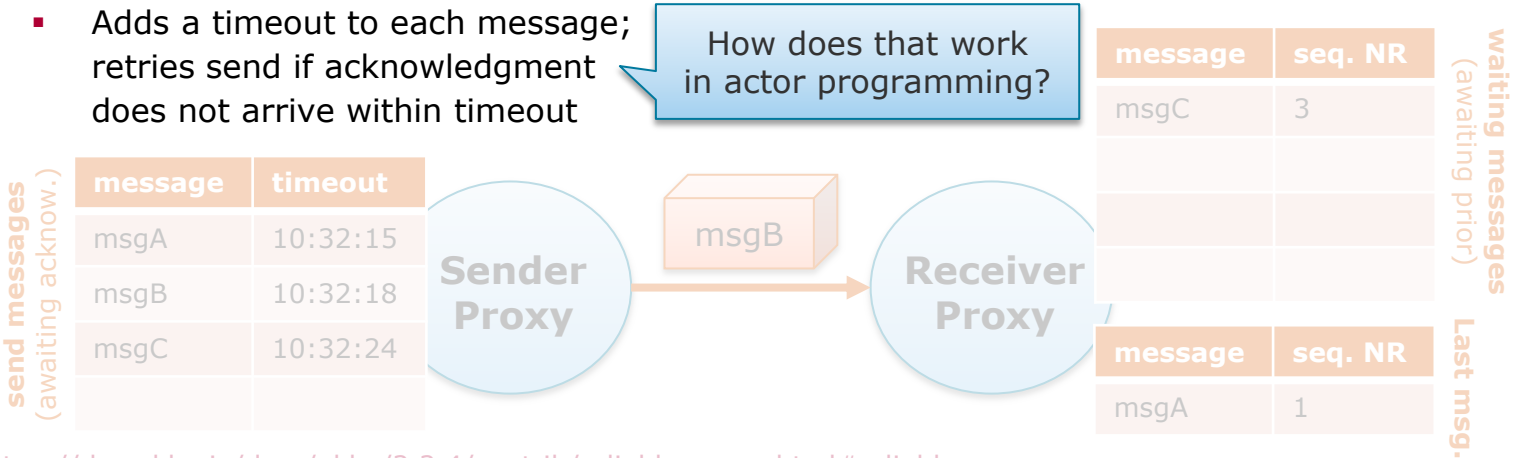
### Sender Proxy

- 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

How does that work in actor programming?




https://doc.akka.io/docs/akka/2.2.4/contrib/reliable-proxy.html#reliable-proxy

# Proxy

## 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.cancle();
  […]
}
```

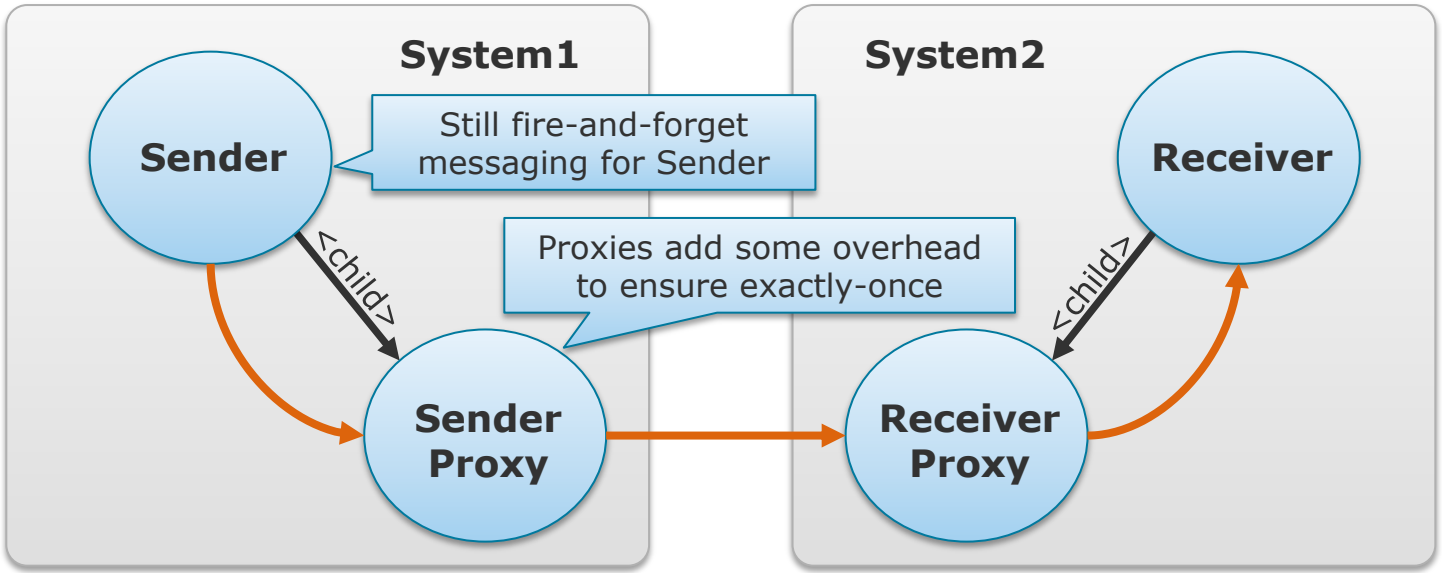
(Re-)send messageA to receiverProxy every 3 seconds

Stop resending messageA

# Proxy

## Example: Reliable Communication Proxy

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




https://doc.akka.io/docs/akka/2.2.4/contrib/reliable-proxy.html#reliable-proxy

Patterns

# Ask

# Ask

## Tell messaging

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

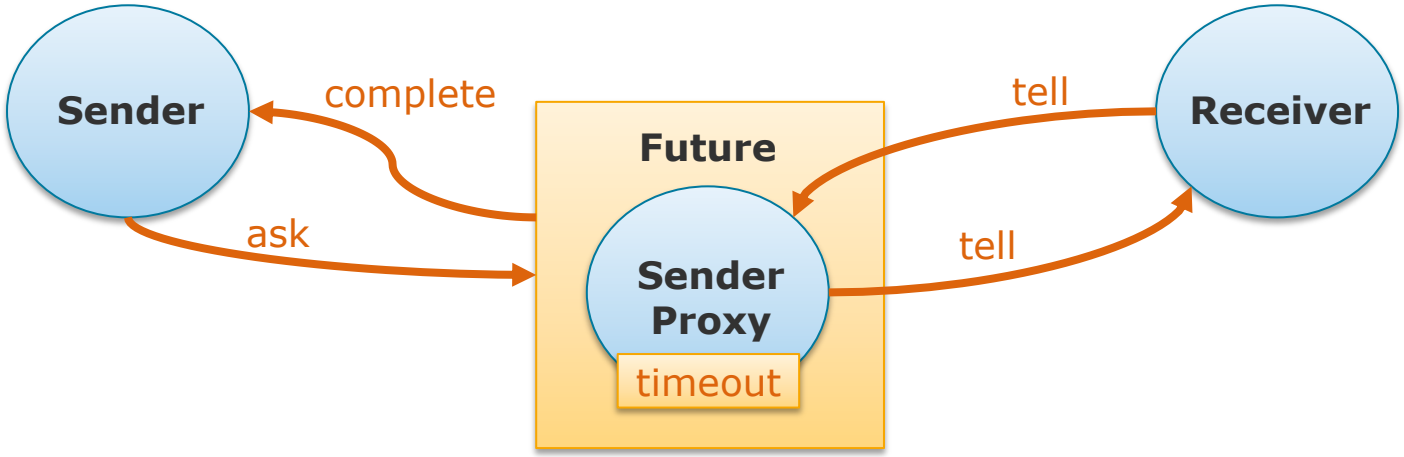
## Ask pattern

- 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

## 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

## Ask pattern

- Useful if …
  - the outside, non-actor world needs to communicate with an actor.
  - an actor must not continue working 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).

# Ask

Why to avoid ask:

- Paradigm violation
  - Synchronous calls break the strict decoupling of actors.
- Resource blocking
  - Actively waiting for other actors locks resources (in particular threads).
- Inefficient messaging
  - Asking requires more effort than telling messages (e.g. timeouts).

**Avoid "ask" if possible!**

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

# 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);
```

# Ask

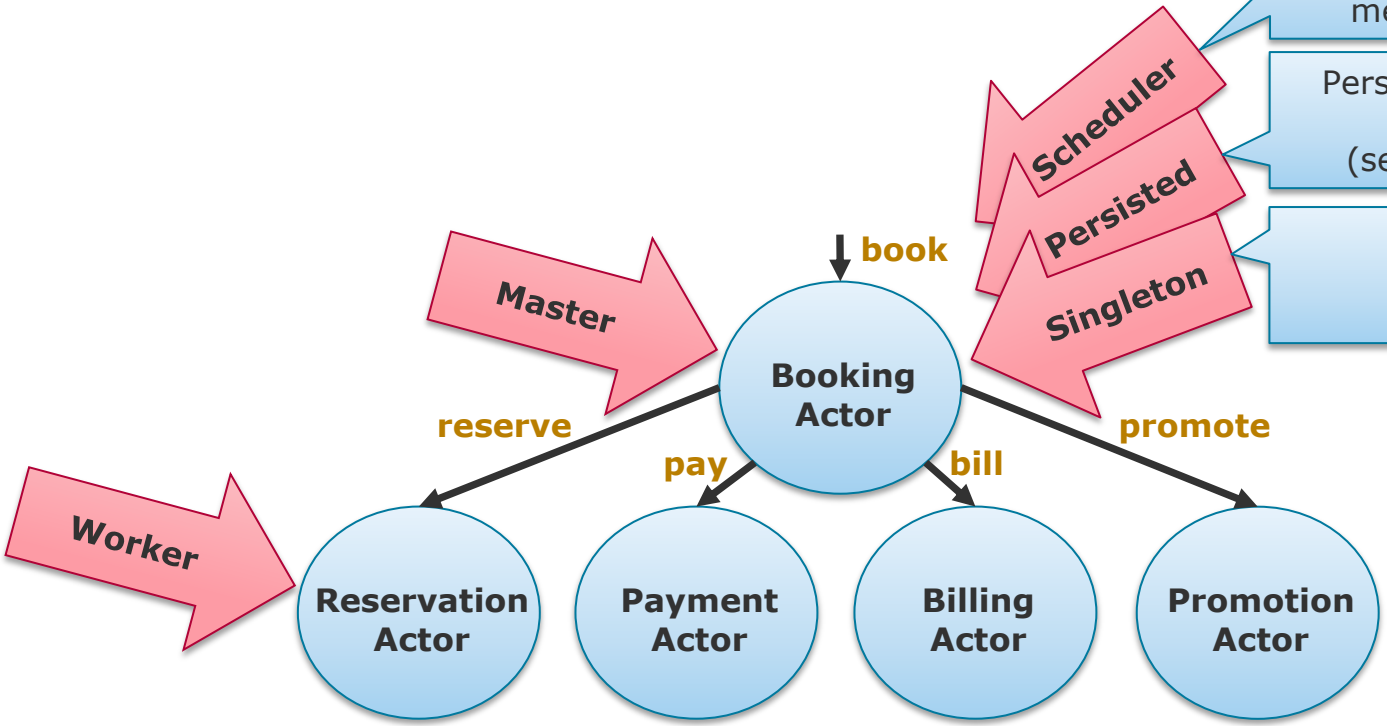
## Actors vs. Futures (+ Pipes)

- 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.
  - callbacks are needed to avoid blocking.
- Why are callbacks bad in actor programming?
  - 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).
    - Bad for debugging, parallelization control, resource management, failure handling, …
    - Prone to introduce shared mutable state and, hence, to destroy encapsulation.
  - Failure handling gets messed up, because the asked actor needs to reply with certain ask-specific error messages to influence its completion.

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

# Singleton

# Singleton


Scheduler
Persisted
Singleton
Register new worker, watch on failed worker, read cluster metrics, balance load, …
Persist or distribute state to restore after failure (see docu on "sharding")
Exist exactly once in the cluster!
Master
book
Booking Actor
reserve
pay
bill
promote
Worker
Reservation Actor
Payment Actor
Billing Actor
Promotion Actor

# Singleton

## 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:

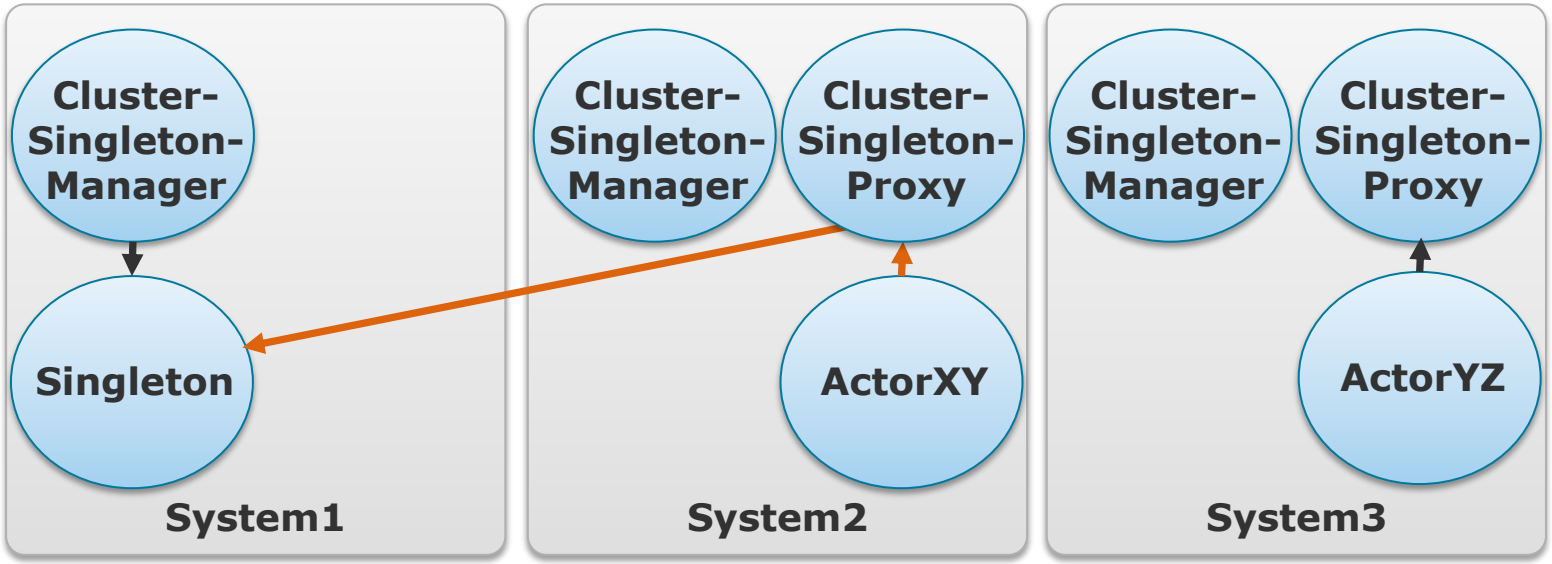
- Simply create that actor once in the cluster.
- Problems:
  1. Requires a dedicated ActorSystem that is responsible for creating the singleton.
  2. If that ActorSystem dies, the singleton is unavailable until the ActorSystem is back.
  3. Starting the same dedicated ActorSystem twice might cause split 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)

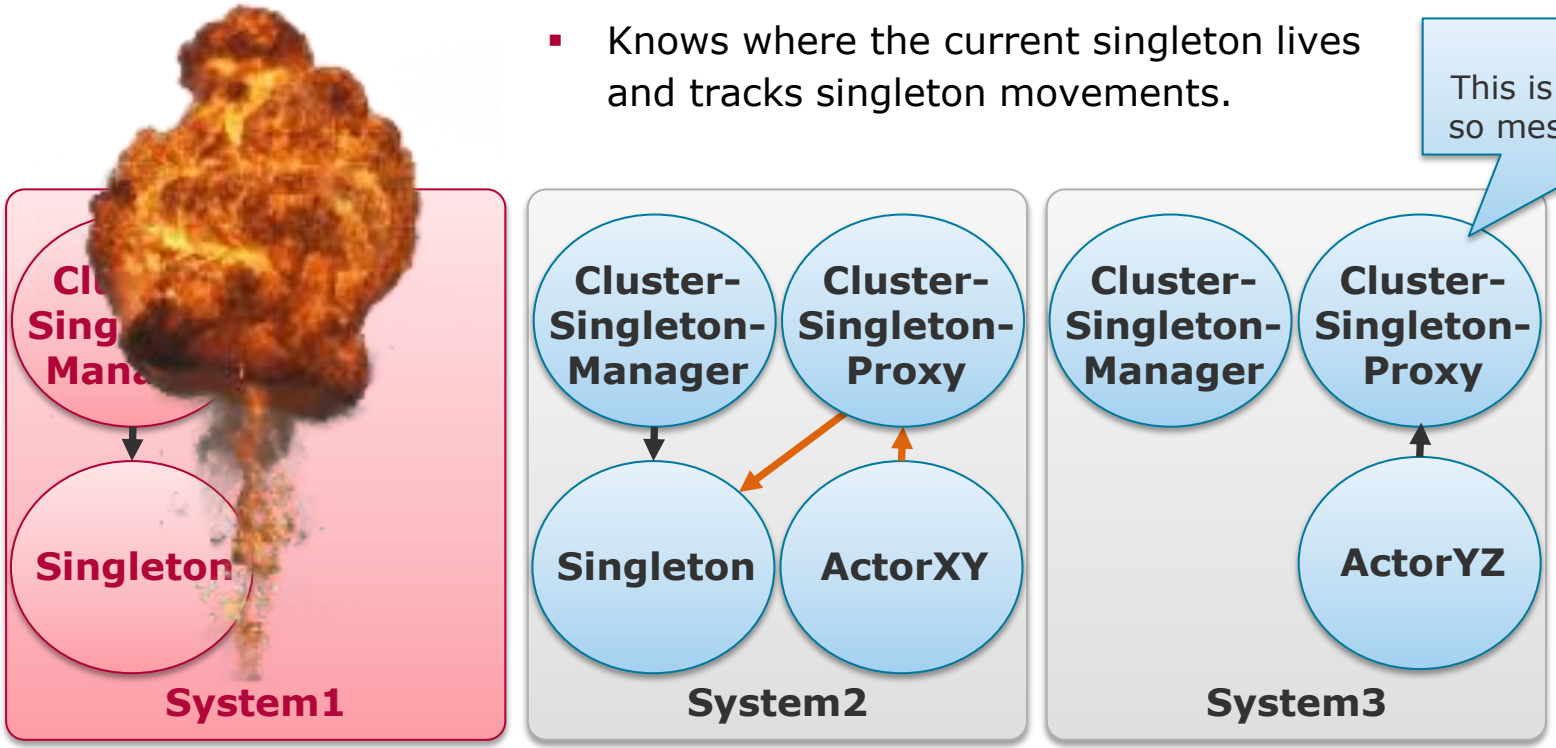


Patterns

# 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.



# Singleton

## 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());
```

# Reaper

# 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 being 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()

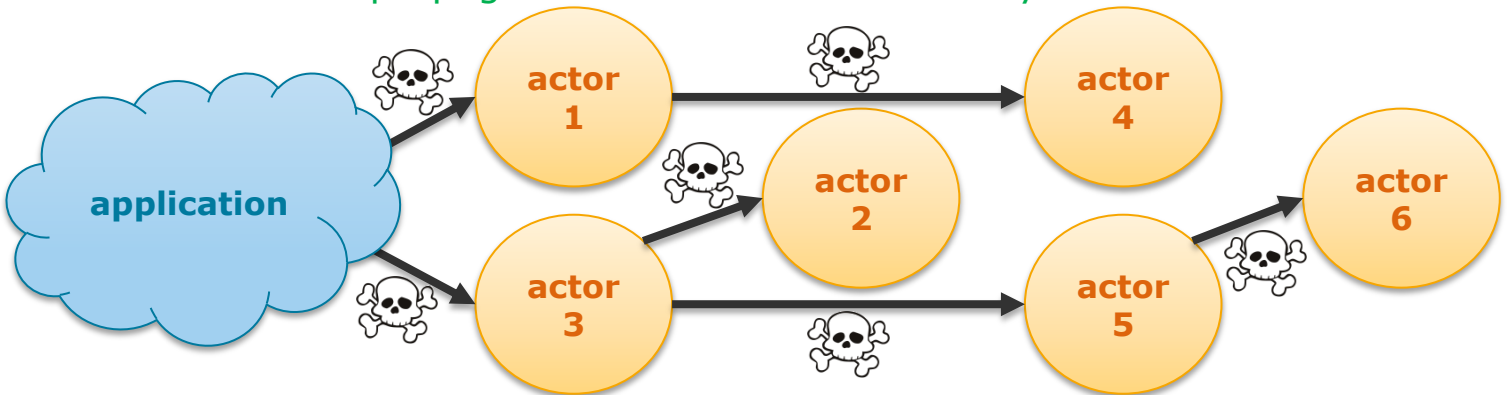
- Calling terminate() on an ActorSystem will stop() all its actors.
- Problem:
  - ActorSystems on remote nodes are still alive!

# Reaper

## 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 PoisenPill are properly processed.
  - PoisonPill propagates into all remote Actor Systems.

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



# Reaper

## PoisonPill Shutdown

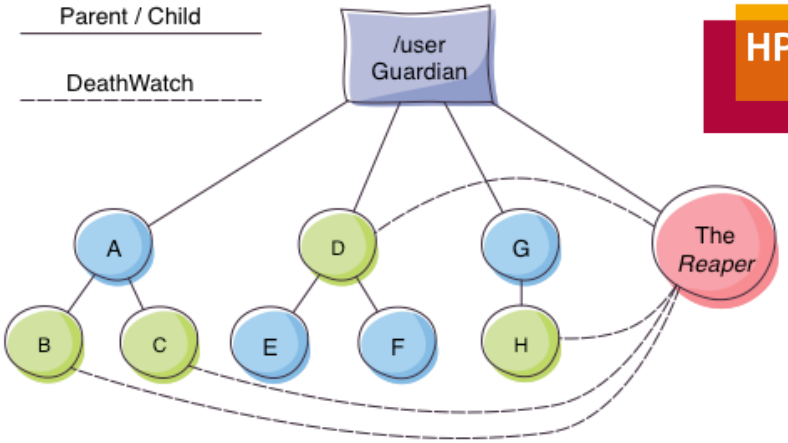
- If an application is done, **send a PoisonPill message to all actors.**
- Actors automatically **forward the PoisonPill** to all children.
- The PoisonPill finally **stops an actor.**
- Advantages:
  - Pending messages prior to the PoisenPill are properly processed.
  - PoisonPill propagates into all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

# 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)



See: http://letitcrash.com/post/30165507578/shutdown-patterns-in-akka-2

```
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());
  }

  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();
  }
}
```

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

Watch a new actor

Witness actor dying

End the world

# Reaper

## 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.

# Reaper

## 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.

# Further Reading

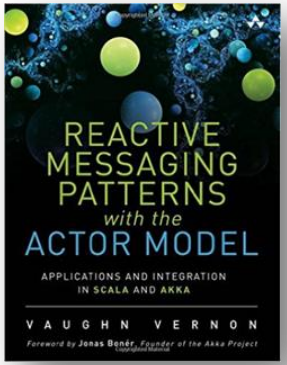
Akka documentation

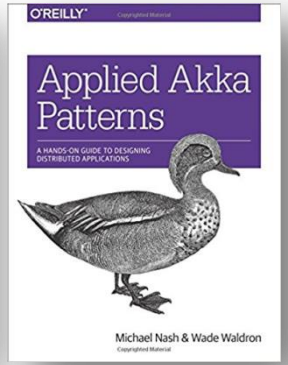
- http://doc.akka.io/docs/akka/current/java/index.html
- http://doc.akka.io/docs/akka/current/scala/index.html

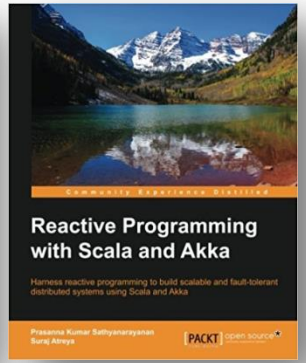
Example code @ GitHub:
https://github.com/akka/akka-samples

Experiences, best practices, and patterns

- http://letitcrash.com
- http://akka.io/blog
- https://github.com/sksamuel/akka-patterns
- Akka actor programming literature:



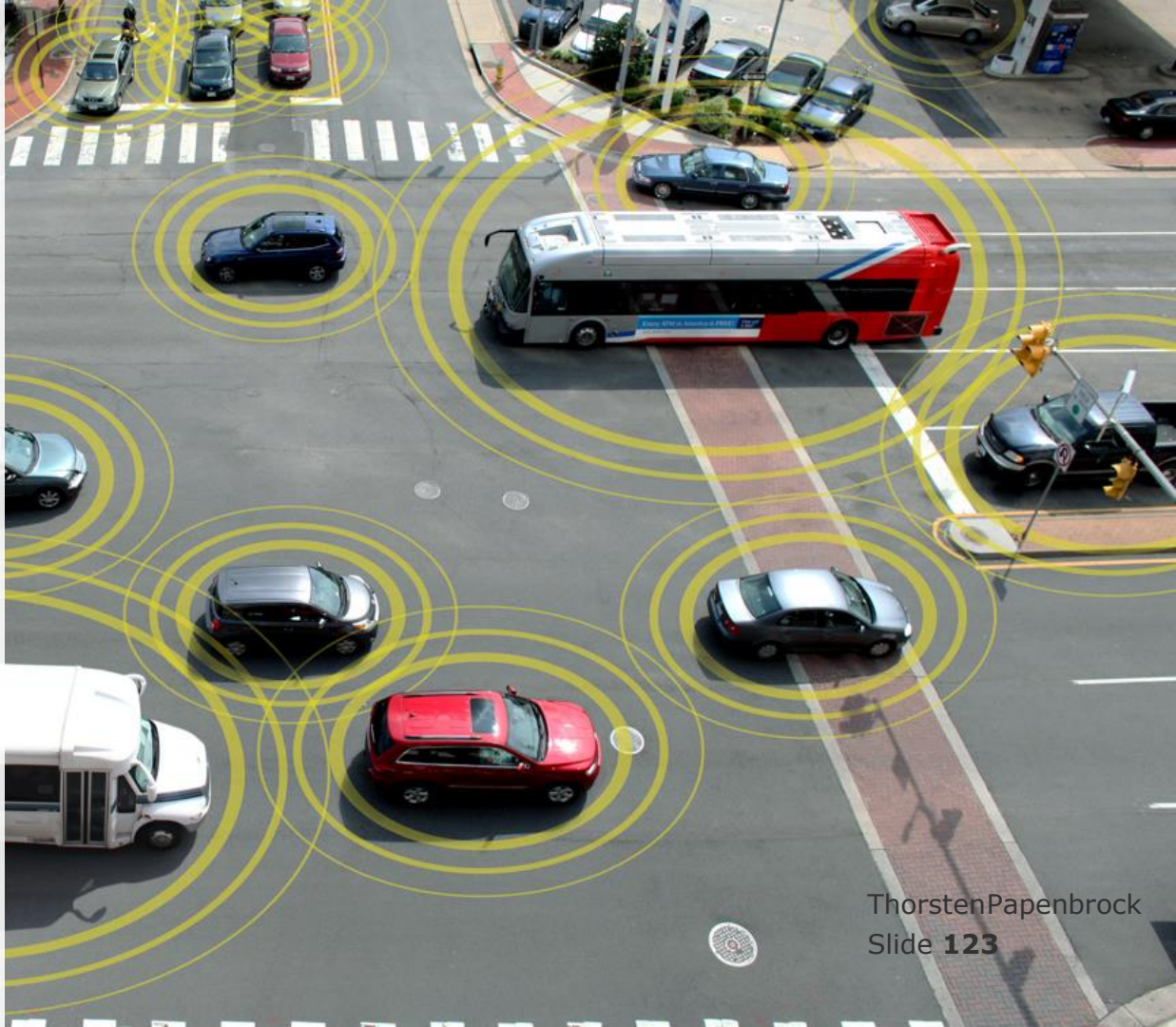
| | | |
|---|---|---|
| **johanandren** Merge pull request #80 from ygree/fsm-typed-java ··· | | Latest commit ccd4323 8 hours ago |
| akka-sample-camel-java | Upgrade to 2.5.17 (#75) | 14 days ago |
| akka-sample-camel-scala | Upgrade to 2.5.17 (#75) | 14 days ago |
| akka-sample-cluster-java | Upgrade to 2.5.17 (#75) | 14 days ago |
| akka-sample-cluster-scala | Upgrade to 2.5.17 (#75) | |
| akka-sample-distributed-data-java | Upgrade to 2.5.17 (#75) | |
| akka-sample-distributed-data-scala | Upgrade to 2.5.17 (#75) | |
| 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

Akka Actor Programming

# Hands-on

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



HPI-Information-Systems / **akka-tutorial**

Watch 7 Star 5 Fork 7

Code Issues 0 Pull requests 0 Projects 0 Security Insights

Code for the Akka tutorial

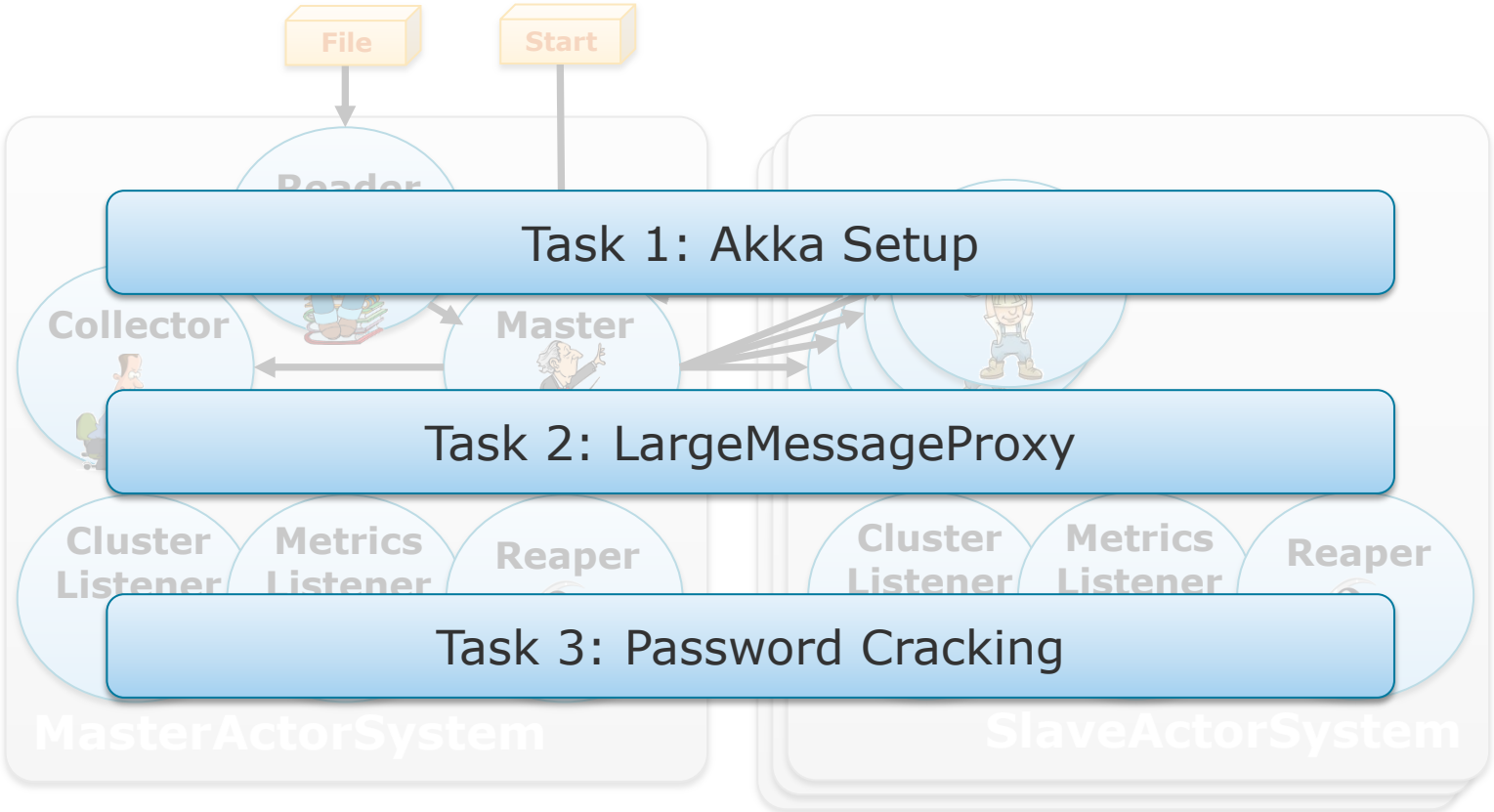
**53** commits **1** branch **0** releases **5** contributors Apache-2.0

Branch: **master** New pull request Find file Clone or download

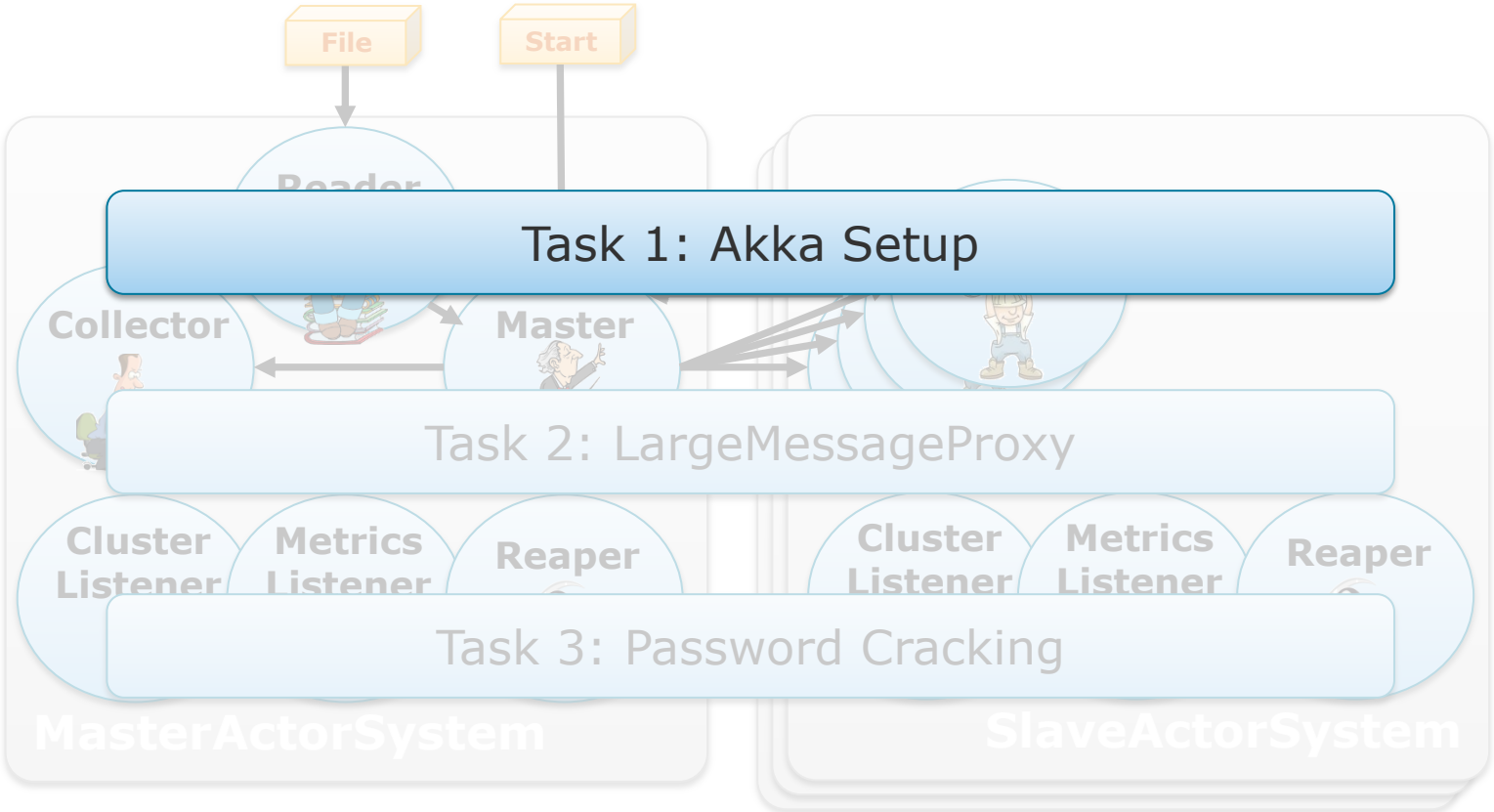
**thorsten-papenbrock** Split the ddm project into ddm-lmp and ddm-pc. Latest commit `bfe21b6` 5 minutes ago

| | | |
|---|---|---|
| akka-tutorial | 1 is not a prime | 11 months ago |
| ddm-lmp | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| ddm-pc | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| octopus | 1 is not a prime | 11 months ago |
| .gitignore | Split the ddm project into ddm-lmp and ddm-pc. | 5 minutes ago |
| LICENSE | Added the octopus project to the repository. | last year |
| README.md | Added the ddm project. | 21 days ago |

# ddm-pc


File
Start
Reader
Collector
Master
Cluster Listener
Metrics Listener
Reaper
MasterActorSystem
Cluster Listener
Metrics Listener
Reaper
SlaveActorSystem
Task 1: Akka Setup
Task 2: LargeMessageProxy
Task 3: Password Cracking

# ddm-pc


File
Start
Reader
Collector
Master
Cluster Listener
Metrics Listener
Reaper
Task 1: Akka Setup
Task 2: LargeMessageProxy
Task 3: Password Cracking
MasterActorSystem
SlaveActorSystem

Homework

# Task 1 – Akka Setup

1. Form teams of two students.
2. Create a public GitHub repository.
3. Copy the ddm-lmp and ddm-pc projects from the exercise repository https://github.com/HPI-Information-Systems/akka-tutorial into your repository.
4. Build, understand and test the two ddm projects.
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

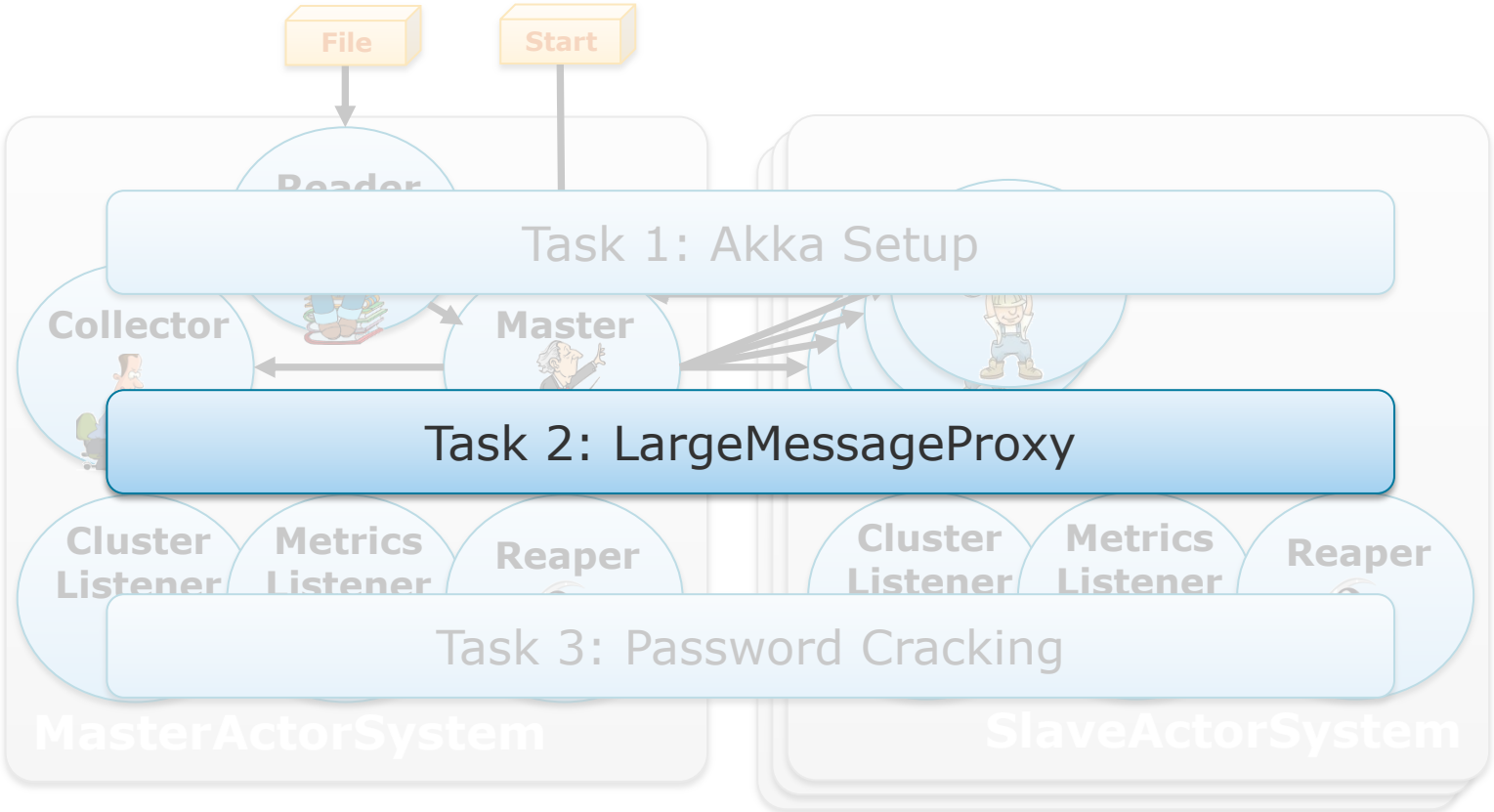
Homework

# Task 1 – Akka Setup

Submission

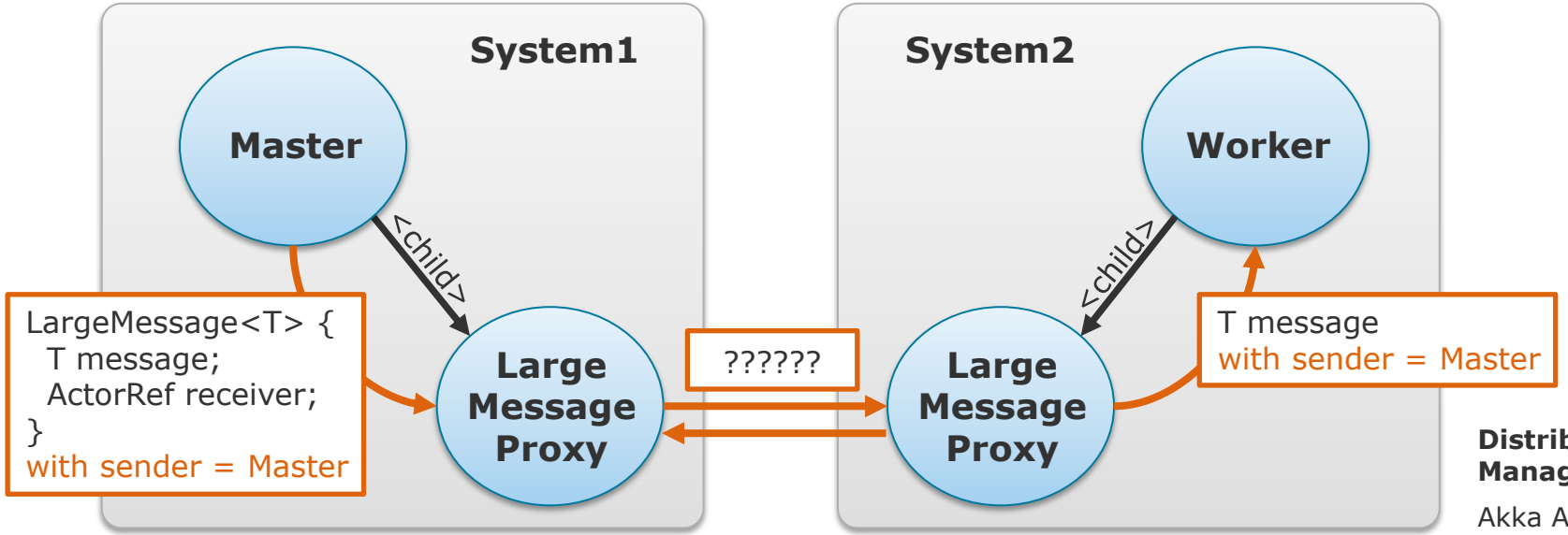
- Deadline
  - 08.11.2019 09:00:00 (next Friday!)
- Artifacts
  1. Email with content:
     <firstname1> <lastname1>
     <firstname2> <lastname2>
     <groupname>
     <GitHub-URL>

# ddm-lmp


File
Start
Reader
Collector
Master
Cluster Listener
Metrics Listener
Reaper
MasterActorSystem
Cluster Listener
Metrics Listener
Reaper
SlaveActorSystem
Task 1: Akka Setup
Task 2: LargeMessageProxy
Task 3: Password Cracking

Homework

# Task 2 – LargeMessageProxy




Task

- Implement the LargeMessageProxy actor!

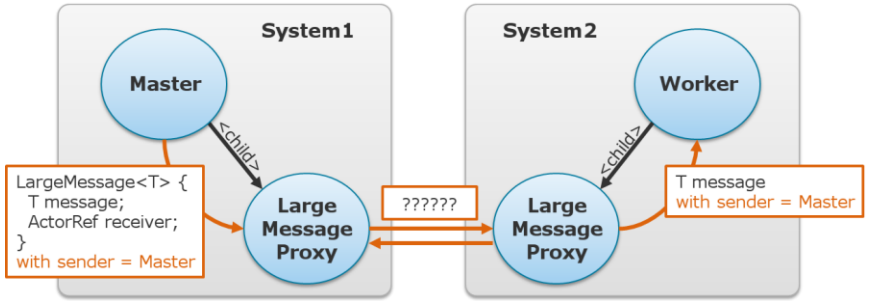
```java
@Override
public Receive createReceive() {
        return receiveBuilder()
                        .match(LargeMessage.class, this::handle)
                        .match(BytesMessage.class, this::handle)
                        .matchAny(object -> this.log().info("Received unknown message: \"{}\"",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

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

        // This will definitely fail in a distributed setting if the serialized message is large!
        // Solution options:
        // 1. Serialize the object and send its bytes batch-wise (make sure to use artery's side channel then).
        // 2. Serialize the object and send its bytes via Akka streaming.
        // 3. Send the object via Akka's http client-server component.
        // 4. Other ideas ...
        receiverProxy.tell(new BytesMessage<>(message.getMessage(), this.sender(), message.getReceiver()), this.self());
}

private void handle(BytesMessage<?> message) {
        // Reassemble the message content, deserialize it and/or load the content from some local location before forwarding its content.
        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!

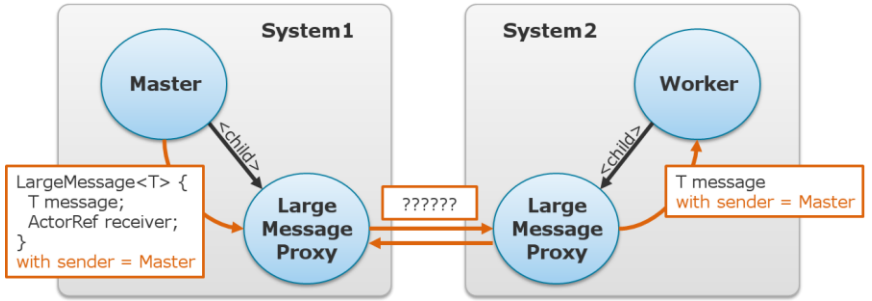


Homework

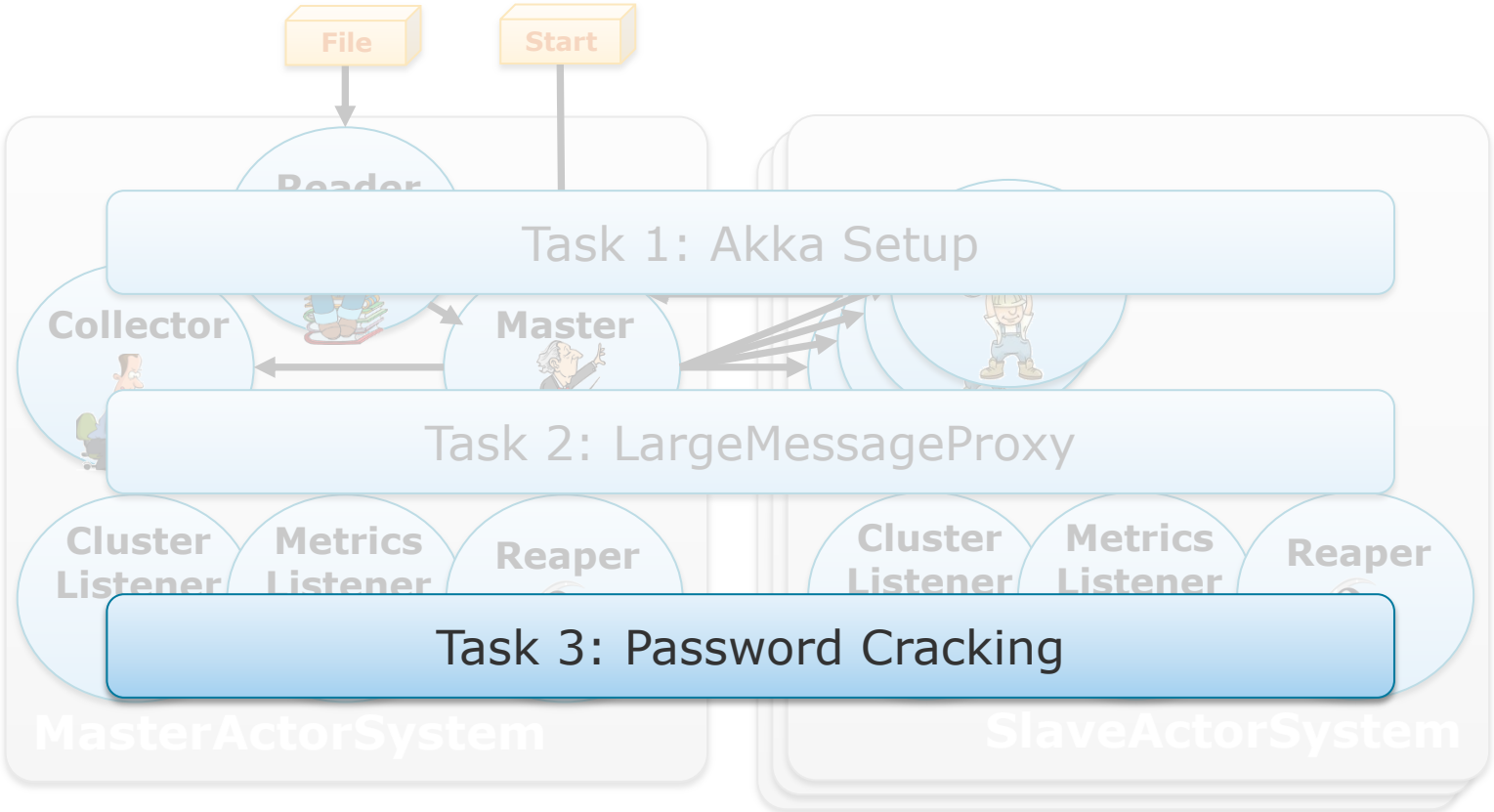
# Task 2 – LargeMessageProxy

Submission

- Deadline
  - 15.11.2019 09:00:00
- 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.



# ddm-pc


File
Start
Reader
Collector
Master
Cluster Listener
Metrics Listener
Reaper
MasterActorSystem
Cluster Listener
Metrics Listener
Reaper
SlaveActorSystem
Task 1: Akka Setup
Task 2: LargeMessageProxy
Task 3: Password Cracking

Homework

# Task 3 – Password Cracking

Hasso Plattner Institut

| ID | Name | PasswordChars | PasswordLength | Password | Hint1 | Hint2 | Hint3 | Hint4 | Hint5 | Hint6 | Hint7 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Sophia | ABCDEFGHIJK | 10 | GGGFGFFFFG | HJKGDEFBIC | FCJADEKGHI | FAJBDIEKGH | AGCJEHFKIB | BHKICGFADJ | JIFAGKDBCE | GAHDKJBCEF |
| 2 | Jackson | ABCDEFGHIJK | 10 | EFFF | | AEHJIDGFKC | IDAHFGEKBJ | EHFIJKBGAC | HFJIEDACBK | FGKIDJCEAB | KDHGCAEJFB |
| 3 | Olivia | ABCDEFGHIJK | 10 | KDD | | CAKEIFHGJD | JBFEDHIKAG | IDAKGHBFJC | KGBAEICHDJ | DKHFBEJIAC | EABJGFIKDC |
| 4 | Liam | ABCDEFGHIJK | 10 | CCCCGGC | | FAICGJDHEK | CHBKIGEJAF | AICDKGHJBF | EDAGKBJHIC | JDKIFACEGB | BGKJDAHCFE |
| 5 | Emma | ABCDEFGHIJK | 10 | BDDBDDDDDB | EGICDFKHBJ | HEAJIBDGFK | BAHCKDFIJG | HBEDKAGCIJ | IBHCEFJADK | FAGDEJICKB | GFHEAKCDBJ |
| 6 | Noah | ABCDEFGHIJK | 10 | GHGGHGGHHH | CFKBJGDIEH | CAIGHEJFDK | GJBEKIADFH | AIBCJHGEKF | GDIBCKFHAJ | CGJHDEAIBK | DGKFBEACJH |
| 7 | Ava | ABCDEFGHIJK | 10 | DFFFDFDFDD | FHIKEBGDJC | KHFICAJGED | KIAHDFEJGB | CGFAKIBDHJ | ACEHFKBIDJ | GBADIJEKFC | AFCKGHBDJE |
| 8 | Aiden | ABCDEFGHIJK | 10 | HHIHI | GCIFEHDKBJ | JDHIEGKACF | FJHBEGAKDI | AIBJEHKGFC | CGJAFBIHDK | JECAIDGHBK | IJBDCKEAFH |
| 9 | Isabella | ABCDEFGHIJK | 10 | CJCJCCJ | EHDCGIKBJF | IFJCAEHGKD | AFBHEGKIJC | KGFBIADJHC | JKAGEDHIBC | CBKIDEAHFJ | CJAFKEIBDG |
| 10 | Lucas | ABCDEFGHIJK | 10 | BBCBCCC | KGJHIDECFB | BHFACKEGIJ | ICJGHFKBAD | KEICHGAJDB | BCDKEJIFAH | IDGEBAJKCF | FCBDKGHJAE |
| 11 | Mia | ABCDEFGHIJK | 10 | IDDDDDIDDI | FIBC | | | | | | BKIGAEDFCJ |
| 12 | Caden | ABCDEFGHIJK | 10 | DDDAADDDDD | AEIC | | | | | | EKFGAJCBHD |
| 13 | Aria | ABCDEFGHIJK | 10 | CCCFCCFFCC | GKH | | | | | | HBJEAFDCGK |
| 14 | Grayson | ABCDEFGHIJK | 10 | JJBBJJ | GEH | | | | | | BKGEJFACDI |
| 15 | Riley | ABCDEFGHIJK | 10 | BGGGBBB | GHE | | | | | | JBCHAGEKDF |
| 16 | Mason | ABCDEFGHIJK | 10 | AAJAJA | EKJ | | | | | | BCGEFAKJID |
| 17 | Zoe | ABCDEFGHIJK | 10 | JJJJJJ | JHC | | | | | | CBEFIDGKAJ |
| 18 | Elijah | ABCDEFGHIJK | 10 | EJJEJEJJEE | KDC | | | | | | JBCGADIEKF |
| 19 | Amelia | ABCDEFGHIJK | 10 | GDDGGGDGDG | GCI | | | | | | HEGDCAKJFB |
| 20 | Logan | ABCDEFGHIJK | 10 | FFEFEEEFFF | KHB | | | | | | JFHDCKEAGB |
| 21 | Layla | ABCDEFGHIJK | 10 | CCCHCHCCCC | GIFH | | | | | | JBGAHFDCEK |
| 22 | Oliver | ABCDEFGHIJK | 10 | ABBBAABAAA | AFK | | | | | | FJGKHEADBC |
| 23 | Charlotte | ABCDEFGHIJK | 10 | BGBGBGBBBG | ECK | | | | | | GJBECKAFHD |
| 24 | Ethan | ABCDEFGHIJK | 10 | HHBBHHH | DHJ | | | | | | BCHJEAKFDG |
| 25 | Aubrey | ABCDEFGHIJK | 10 | JJEEJE | HDFEJBKICG | IF | | GEBIK | KCEFBAIGHJ | IDCJBAGEHK | IKHJCDBAEF |
| 26 | Jayden | ABCDEFGHIJK | 10 | CCCGGGG | DJEHCBKFIC | | AIDHE | ECFKBIAHJG | GDCIFKBJAH | HJBGAIKCED | DICKFBGAEJ |
| 27 | Lily | ABCDEFGHIJK | 10 | DHHDHDD | KCJFIBHEDG | FJDKCAIEGH | DABGJEFKIH | DCGIKHFABJ | KDGBEHIACJ | IBCKDFHJAE | EHABDKCFJG |
| 28 | Muham | ABCDEFGHIJK | 10 | CBCBBBC | EDFGHKIBJC | HEICAKBJFG | CABIDFGHKJ | DHAKICBGJE | IHAKJCEBFD | CEABFJGKID | ABGKFDHCEJ |
| 29 | Chloe | ABCDEFGHIJK | 10 | CEECECCCEC | KEGDHFCBIJ | GCIHAEDKFJ | HFGKIBACEJ | CJHGKBDAIE | FECIBJKADH | GAFCIBEKJD | CJBAKEGDFH |
| 30 | Carter | ABCDEFGHIJK | 10 | BBIIBIBIIB | CKFGBIHDEJ | EJDKIHGABF | ECHIJGFBAK | AFHIBCKGDJ | IHCBKGEJAD | AFHIDJKBCE | IBGCJKFAED |
| 31 | Harper | ABCDEFGHIJK | 10 | EAAAEAEEAE | IAFKCHJGDE | AKFDJIHGBE | BGECFIJKAH | BJDAIGKEHC | IHEBKACJDF | BJDIGAKFCE | EFADJKCBGH |
| 32 | Michael | ABCDEFGHIJK | 10 | DCDDCCDDDD | JGFEICBKHD | CKJDHGIEAF | KIGDHABJCF | GDEIHACBJK | BICEAFDHKJ | JGFCKBDEAI | HCBGDKFAJE |
| 33 | Evelyn | ABCDEFGHIJK | 10 | IICICCICCC | CDBJHIEGKF | FIKGEHCAJD | BIJAGEKFHC | CADBIHFJKG | GHJDBKAEIC | KJEIDHABCF | ACBGKFDIJE |
| 34 | Sebastian | ABCDEFGHIJK | 10 | EIIIEIIIIE | EDFKHGJBIC | KFHIDJGACE | KHAIDGBJFE | CIABGJKEHF | JKEIDCBAHG | KIDFJABHEC | FDBGJCAEIK |
| 35 | Adalyn | ABCDEFGHIJK | 10 | JJJJDJDDJJ | IKEJFCBHDG | AHDCFKEJGI | DBIKAHJGEF | DKAIHGFJCB | CABKHEDJIG | ABJEFCHDKI | GJAFEBCDIK |
| 36 | Alexander | ABCDEFGHIJK | 10 | HKHKHKKHKK | CEBHKDFIJG | FGJADKCIEH | AGDIFHKJBE | FHCEKABGIJ | BFJAIDCGHK | BCAHEJDGIK | JFCDIHKAEB |

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., |Hint|=|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.

Homework

# Task 3 – Password Cracking

| ID | Name | PasswordChars | PasswordLength | Password | Hint1 | Hint2 | Hint3 | Hint4 | Hint5 | Hint6 | Hint7 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Sophia | ABCDEFGHIJK | 10 | c4712866799881ac48ca55bf78a9540 | 1582824a01c4 | e91aca467f5a | 52be0093f91b | 8052d9420a20 | ca70f765d8c1b | 570d3ada41de | f224061bd035 |
| 2 | Jackson | ABCDEFGHIJK | 10 | c178ef3bd2dbf4e92291a9b563c0ae2c | 7624e76e72b52 | 834d255d0276 | b2e939a89b78 | 0f0c2aefcfcf4b3 | d22b58963201e | 0066eb98a0f3 | 21b5a6f0b9c15 |
| 3 | Olivia | ABCDEFGHIJK | 10 | b6d | | | | | 0dd9e26059942 | f70853fec1c11 | b0f110e28c9d9 |
| 4 | Liam | ABCDEFGHIJK | 10 | 109 | | | | | 90d6920945a7 | b857b99db7ab | 503d34487226 |
| 5 | Emma | ABCDEFGHIJK | 10 | 607 | | | | | 552ba27c5ae4c | 60b64d370b60 | b7fdd9f77b932 |
| 6 | Noah | ABCDEFGHIJK | 10 | 6d4 | | | | | a1601cb73654c | 62ecbbd80652 | a1ba7bb71eb9 |
| 7 | Ava | ABCDEFGHIJK | 10 | 4121ab0055971 | | e00595b2cab3c | 90d6247cbef52 | 72c52457352b | 7ff4495040445c | 628dfdd46cd2f8 | c901b559232e | b8209fa62631 |
| 8 | Aiden | ABCDEFGHIJK | 10 | fbe3613750f71d7996e9d63601dc7fd4 | de2617fb757fc0 | 06bb6d175e5d | 03ee78244a72 | 87316b71fbfc44 | 9aab84d04556e | 87a65ceb83b5 | 589c35f40243e |
| 9 | Isabella | ABCDEFGHIJK | 10 | 5a22e3bdef6c85307b361f2e1758f461 | 23d6de9da425b | 7af3c5c070a12 | 824137665f56c | 71deb0e1e18a | 49535ddb45d7a | 9271a854a0e6 | 6b2b2dbf84e0a |
| 10 | Lucas | ABCDEFGHIJK | 10 | 49afdd0a20ae497060405ec7b557faa6 | 041734164643 | 0df7feecbe4bb | 046c1ffec90e0 | a221a7c41ebf4 | dcbe04357c159 | ae51984b3c8c | e5b090db2396 |
| 11 | Mia | ABCDEFGHIJK | 10 | 77026d73fb8c33e0f45c3f6bc3 | | | | | | 30469e377b | 5b451e4478c4 |
| 12 | Caden | ABCDEFGHIJK | 10 | 484616315092a69ebd7cf4c1b | | | | | | 570b879d43 | eb83b37c3c50 |
| 13 | Aria | ABCDEFGHIJK | 10 | 3fff9b667a867fccaada0d823d0 | | | | | | 0d3bbb35405 | 39b7290d29bd |
| 14 | Grayson | ABCDEFGHIJK | 10 | ac923aa891c087fad57b02de9 | | | | | | bb614d9f1c9 | 485a4dbf7cc98 |
| 15 | Riley | ABCDEFGHIJK | 10 | 57203d2db503c69464900aed | | | | | | 1595ef921e | 6350339f168b8 |
| 16 | Mason | ABCDEFGHIJK | 10 | 4d873360dd931098ead7d692 | | | | | | 7433be73e8a | 7aa536698df56 |
| 17 | Zoe | ABCDEFGHIJK | 10 | f2095d3f48f6c0366423436865 | | | | | | 023d0fbf0d4a | f9f4dfc2082c69 |
| 18 | Elijah | ABCDEFGHIJK | 10 | 25e975a018dd7265dcb44a17 | | | | | | e3d813569ce | 698303662587 |
| 19 | Amelia | ABCDEFGHIJK | 10 | 6fb693ee39e015290f087a0ca | | | | | | 9182a060c9 | 584a163c9c80 |
| 20 | Logan | ABCDEFGHIJK | 10 | 1d43da0376f725fff867e1096e3635c9a | 8fb | | 41c672365d64 | dfcd28d1604f0a | 45759a68fbe9c | 1c8b4ed5fccc | b58d913c2aac |
| 21 | Layla | ABCDEFGHIJK | 10 | c3647d6d4f8e8136cf7640d1976d234 | | 494879c4612f | 70d759f30b22 | 74e53e2720fee | 10a55cb79be0e | 56459103a26b | efaee8c12e011 |
| 22 | Oliver | ABCDEFGHIJK | 10 | d2488287e89e2bb00bff6c4e767fe5 | c4c588e7b790b | 911eca14216d | f52b85e1bb8c | 3ba17b3532e8 | 8848a569dcd0 | f9b2ecef6af24 | 9ab4b25771e3 |
| 23 | Charlotte | ABCDEFGHIJK | 10 | 0e481c55eea1567bf4a5434cc0d713d | d6426c5a36fa4 | 7578c180f1b19 | 46588a7ef05e0 | 5daadc4464d5c | bf1244e8e8879 | fe47bd94da1e | 7a8c1b2e2827 |
| 24 | Ethan | ABCDEFGHIJK | 10 | d08ce9b35434a29b6d34ae4df99114e | 537ceb64562eb | ac4c0b2db991 | b43e80f0be33 | cccd5b0f386b8 | 868e118ee7dd | eed78dc7c439 | e30e4a6278bf |
| 25 | Aubrey | ABCDEFGHIJK | 10 | a54 | | | | | dbef55053480e | 39769b86753a | ec515cf86d99 |
| 26 | Jayden | ABCDEFGHIJK | 10 | 482 | | | | | 0b28c5a0bdae | d43e7731d10e | 09c6383d87f59 |
| 27 | Lily | ABCDEFGHIJK | 10 | 64e | | | | | 71545ceb163c0 | 0b5a12e03c63 | 36b071418c49 |
| 28 | Muhammad | ABCDEFGHIJK | 10 | b24 | | | | | c72f7ec4768b1 | ff8ffabe05c576 | 73170f8660b2b |
| 29 | Chloe | ABCDEFGHIJK | 10 | 314885f3b250cfad9a08ab7c6a0b7125 | ba60bc240c6f8 | b32fc6c704f82 | | e9d2d120af79 | e2a2fb62382d6 | 1e52d3a60770 | 9e6240b987eb |
| 30 | Carter | ABCDEFGHIJK | 10 | 507b389927e0aa92bdf50e7ffe0c119c | 2221935370639 | ec62e5d714ffd | f4ce80b687 | 3e0e6e85ce | 7c1163a66461c | 18e32024bb20 | 4d35243c99da |
| 31 | Harper | ABCDEFGHIJK | 10 | 17649029a718c93179e9da331e78012 | f4aa95f0083c60 | b0c11abac12a | 98878f2bea2d9 | 61101eb1bfae6 | d694ff1668eca5 | 2a2e2681f0d0 | 5194a8888927 |
| 32 | Michael | ABCDEFGHIJK | 10 | a926deae7e334a3992fbfa30d4d75823 | 8b63be6310da9 | 03fbdc4f9b69fe | d96188307f70c | 6bdcb76976d88 | e54cb6eabaa20 | 9c2f8383f1aa5 | 04bd3a11b5e4 |
| 33 | Evelyn | ABCDEFGHIJK | 10 | 43079487b664ebafba46e77698d58a4 | 7b43a0546a75f | c6c9a5d45cf19 | 67cbc51d481e | b1a79b2429508 | 59099a87582e | ac6479c44e48 | 7d490919fceb7 |
| 34 | Sebastian | ABCDEFGHIJK | 10 | 0306aed6a72de9d32e0b9d9ec430e92 | 8e5837886ae8a | 2a9f2b7b2e974 | 5effda9ae8fd9 | 037bcf1d83a00 | 3d4b9e8ba7bca | 9379b2c12b91 | f1f3b79be05dc |
| 35 | Adalyn | ABCDEFGHIJK | 10 | bef1a0cc6ba9868fe2071e80b7069f24 | 622ba2b0c4557 | 1087ebc69f50b | c28553d4a058 | 74802c5978ebf | 50e770146296 | ba4c14f03ca99 | d2bda4956404 |
| 36 | Alexander | ABCDEFGHIJK | 10 | f14a798017874d94e78421db5a126e6 | 50e4f0b88e214 | b5502b12a7d7 | d897e5993c0d | 11547ce885e7c | aa8e5f28e181e | cd8b68b0cdba | da8408b0a088 |

Both password and hints are SHA-256 encrypted.

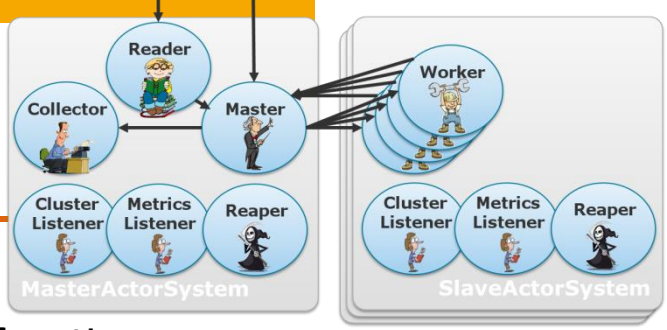
Encryption cracking via brute force approach:

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

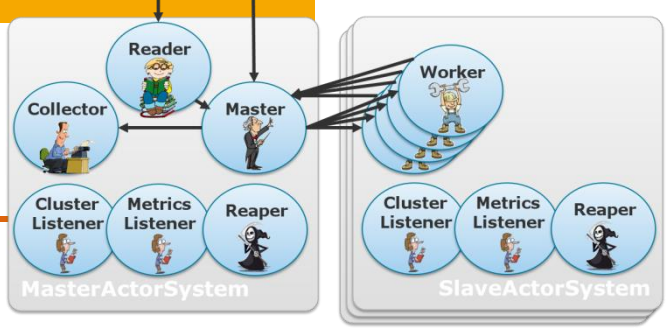
- The passwords and hints are encrypted with the following function:

```
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/

Homework

# Task 3 – Password Cracking

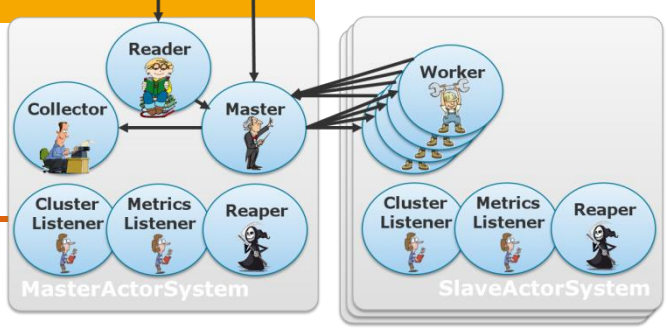



Hints

- 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)

Homework

# Task 3 – Password Cracking

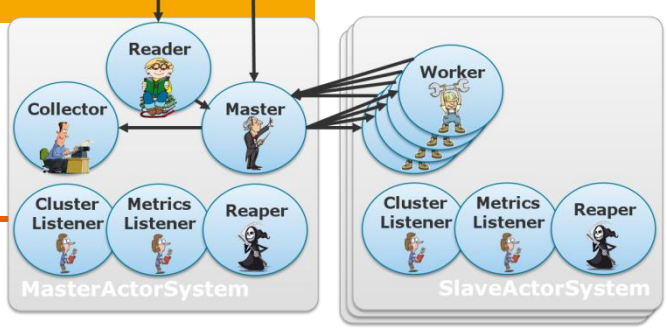



Notes

- 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

Homework

# Task 3 – Password Cracking

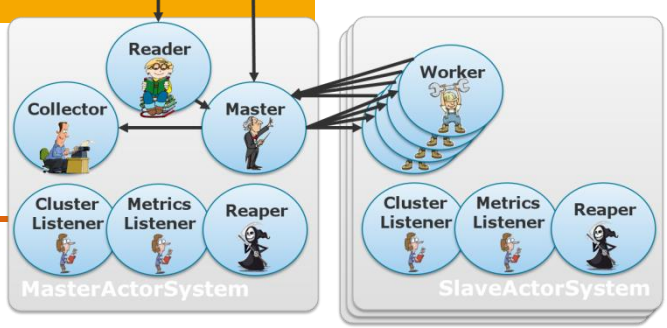



Rules

- 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.

Homework

# Task 3 – Password Cracking

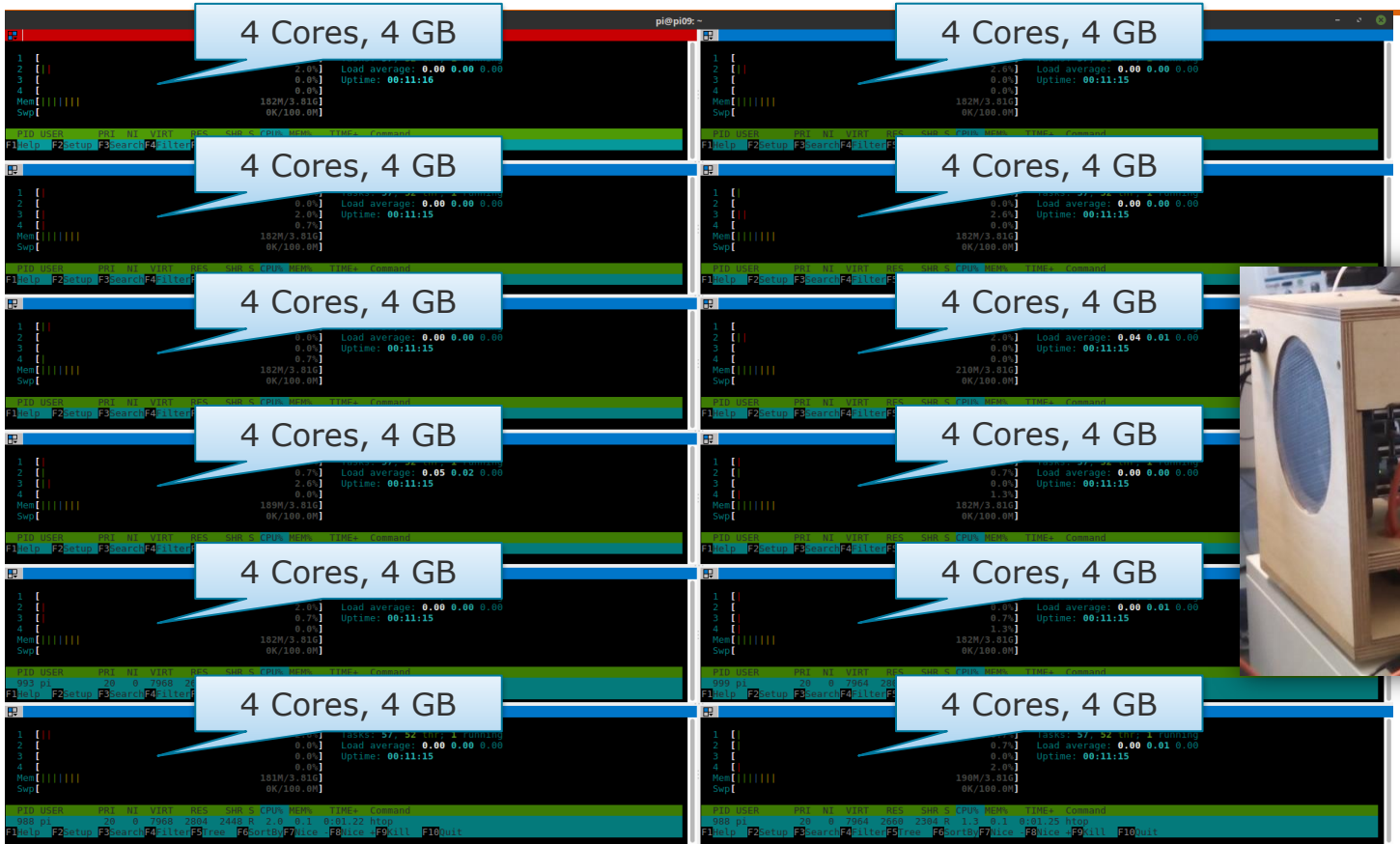


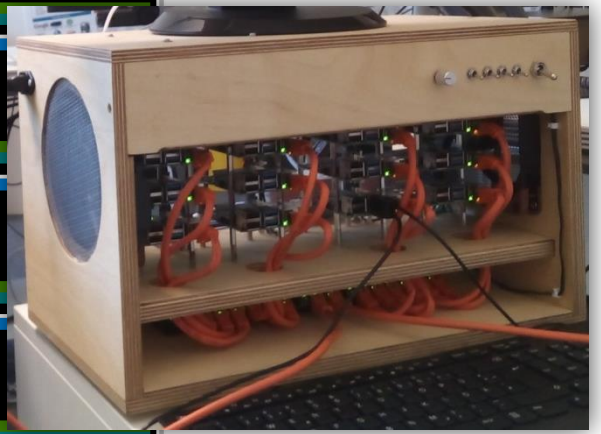

Submission

- Deadline
  - 22.11.2019 09:00:00
- 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.

Homework

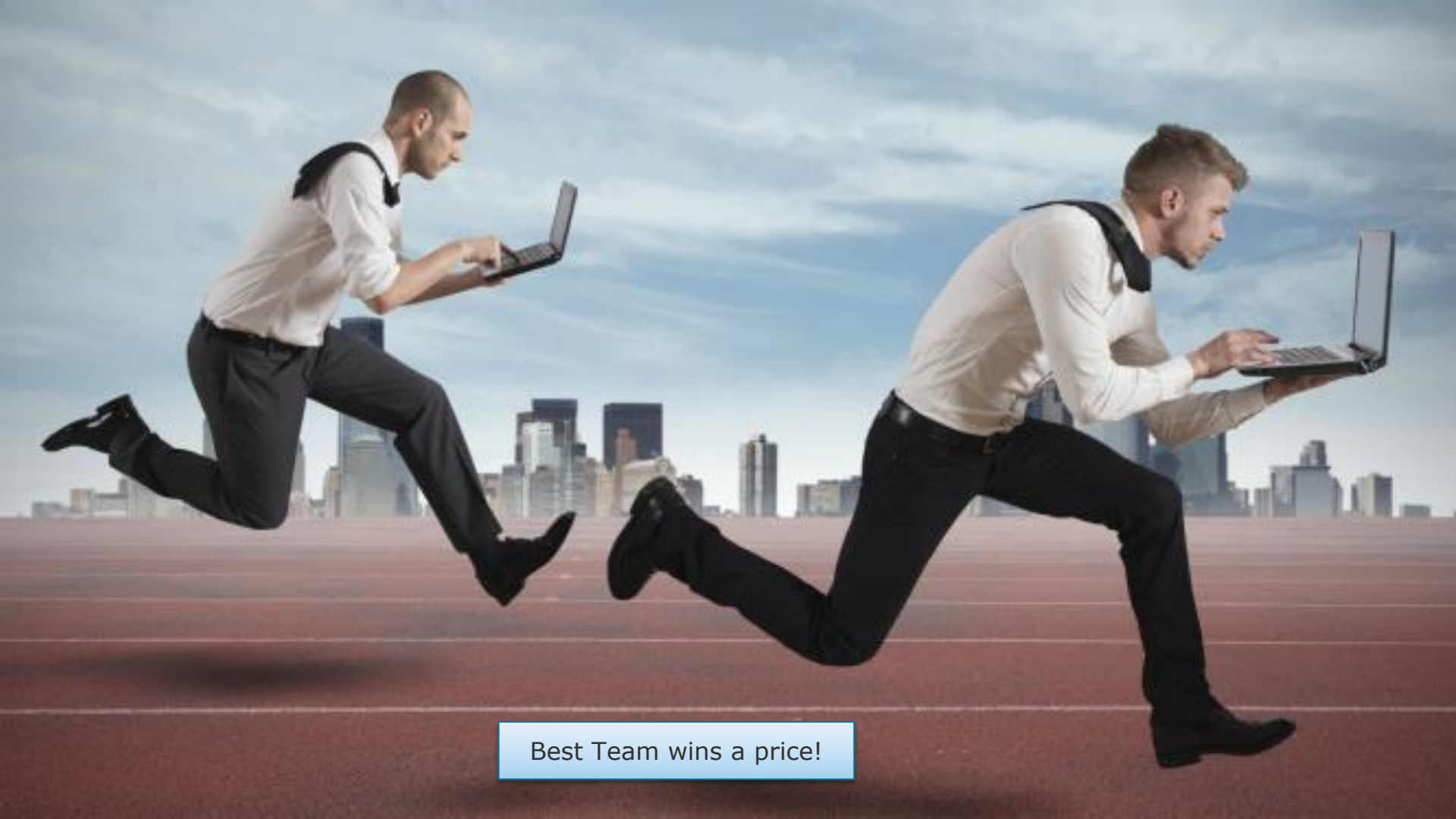
# Evaluation – Pi Cluster



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!"