
# Distributed Data Analytics
# Akka Actor Programming

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut

# 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

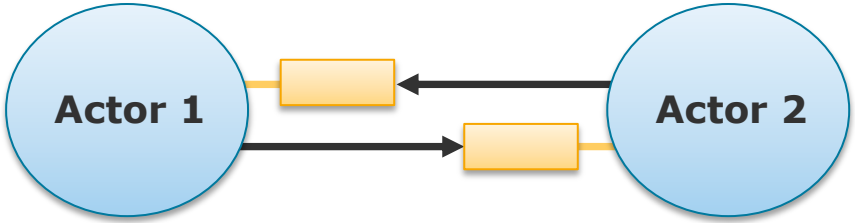
➢ Combines the advantages of object- and task-oriented programming.

## 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 (Rep)

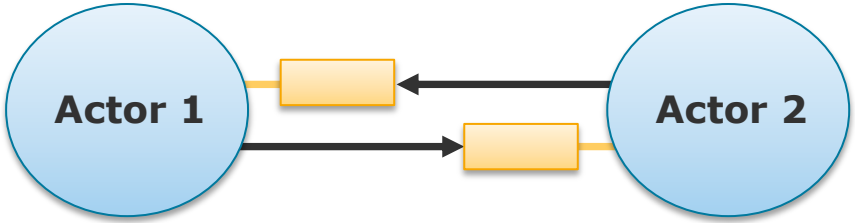



## Actor Model

- 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
  - 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 (Rep)



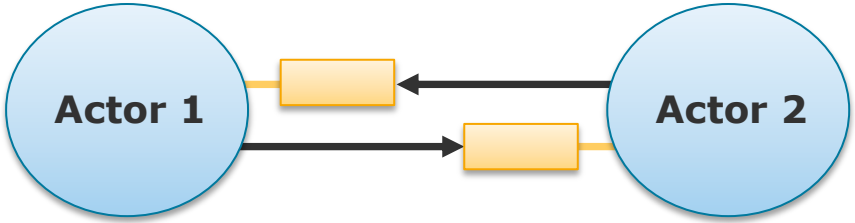

## Advantages over pure RPC

- Errors are expected to happen and implemented into the model:
  - Message loss does not starve sender, because messaging is asynchronous
  - Crashing actors are detected and (depending on configuration) automatically restarted
  - Undeliverable messages are resend/re-routed
  - …

## Dynamic Parallelization

- Actors process one message at a time
  - Parallelization between actors not within an actor
  - ➢ Spawn new (child) actors if needed

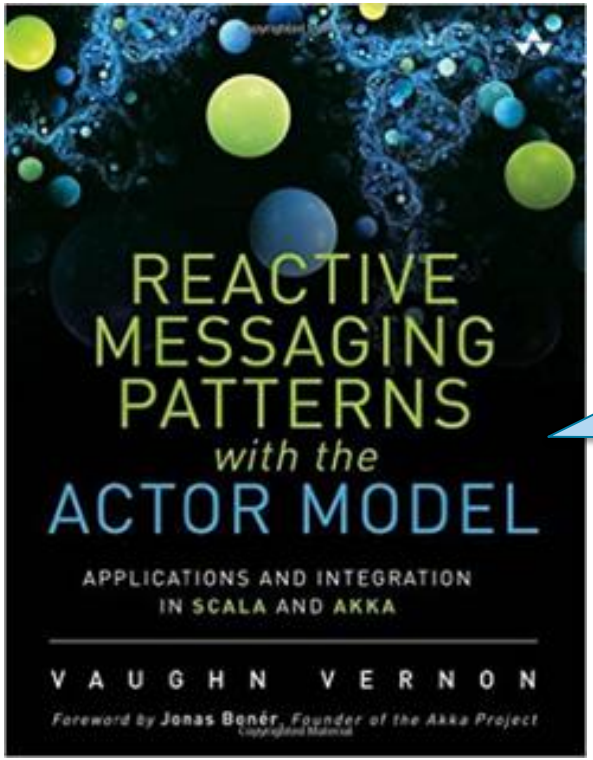
# Actor Model (Rep)




## Popular Actor Frameworks

- Erlang:
  - Actor framework already included in the language
  - First popular actor implementation
  - Most consistent actor implementation
    (best native support and strongest actor isolation)
- Akka:
  - Actor framework for the JVM (Java and Scala)
  - Most popular actor implementation (at the moment)
- Orleans:
  - Actor framework for Microsoft .NET

# 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

# Toolkit and Runtime


- A free and open-source toolkit and runtime for building concurrent and distributed applications on the JVM
- Supports multiple programming models for concurrency, but emphasizes actor-based concurrency
- Inspired by Erlang
- Written in Scala (included in the Scala standard library)
- Offers interfaces for Java and Scala

# 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 …

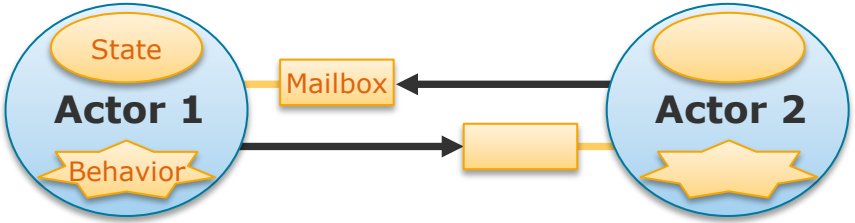
**Logger library**
logging event bus for akka actors

**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>
```

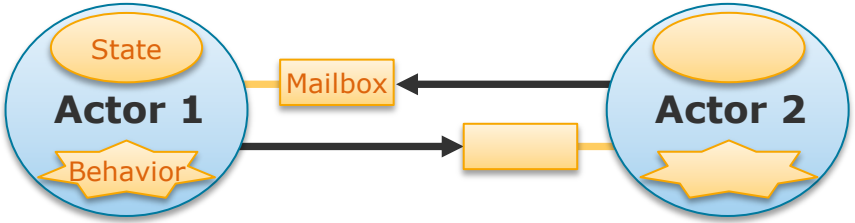
# Actors




- 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!**

# 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());
  }
}
```

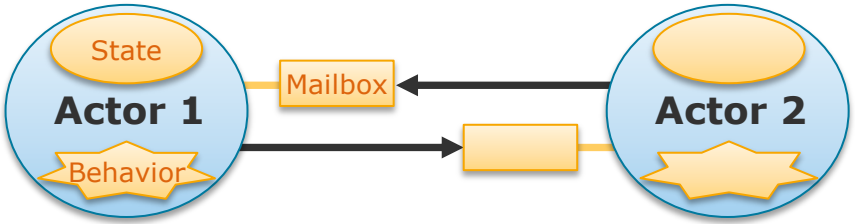
Inherit default actor behavior, state and mailbox implementation

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)

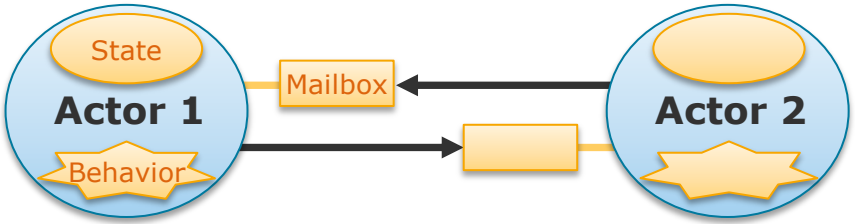
# Actors




```
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

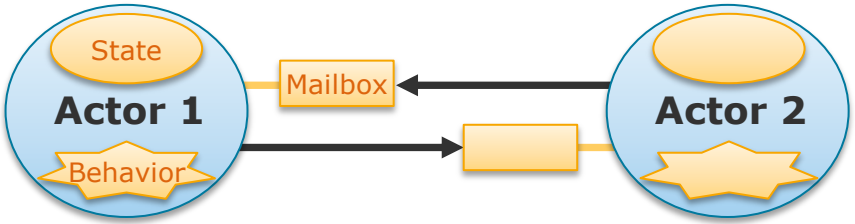
# Actors




```
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

# 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 a kind of interface description)

# Excurse: Serializable

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 Hierarchies

## Task- and data-parallelism

- Actors can dynamically create new actors

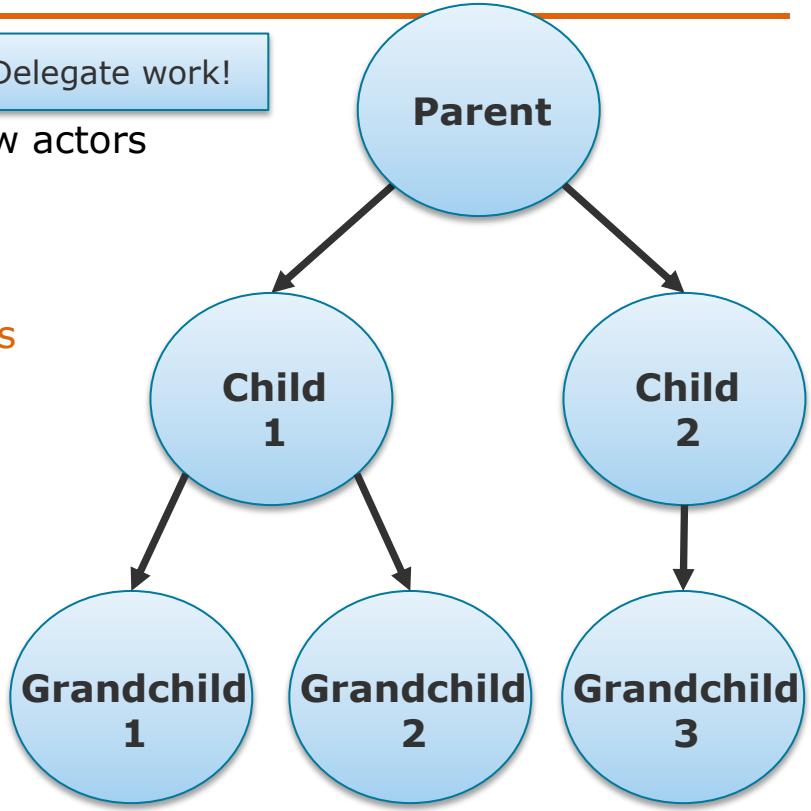
Delegate work!

## Supervision hierarchy

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

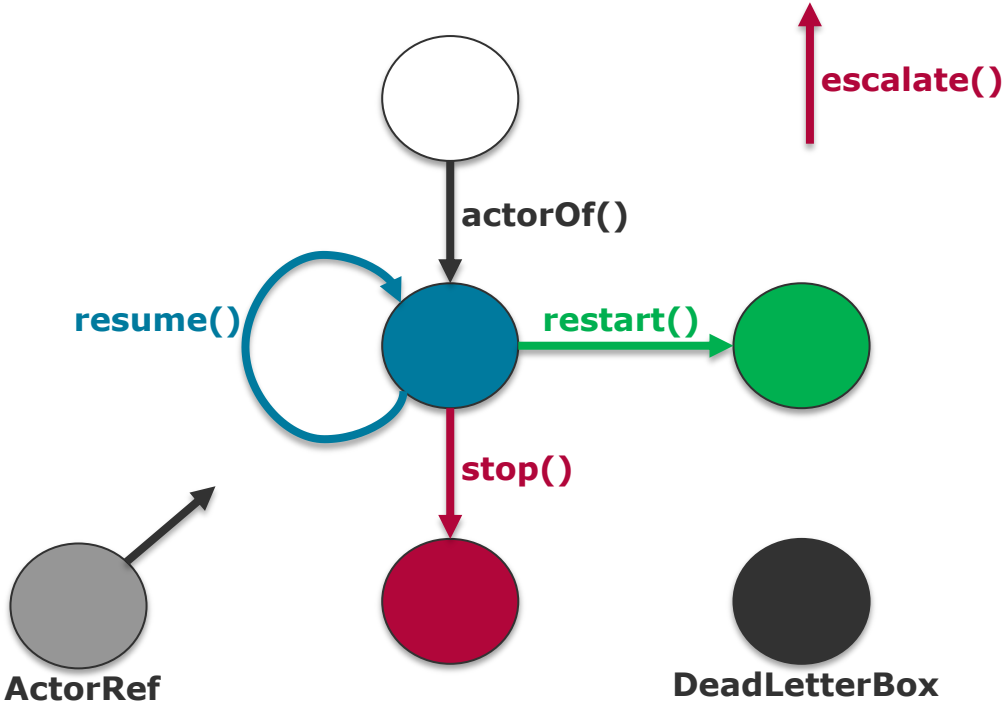
## Fault-tolerance

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



# Actor Lifecycles

## Actor Lifecycle



# 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**

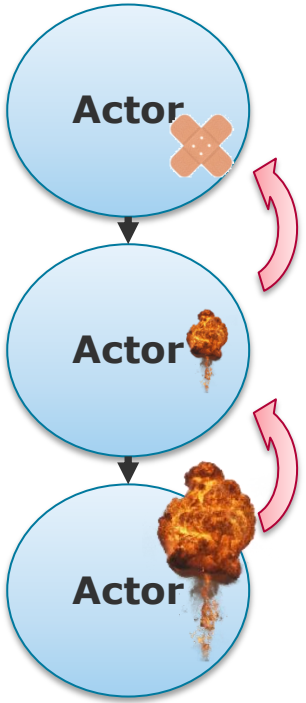
```
public class MyActor extends AbstractLoggingActor {

  @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 discovers a parsing error and crashes
  - Its supervisor restarts the actor and resends the corrupted message



# Actor Hierarchies

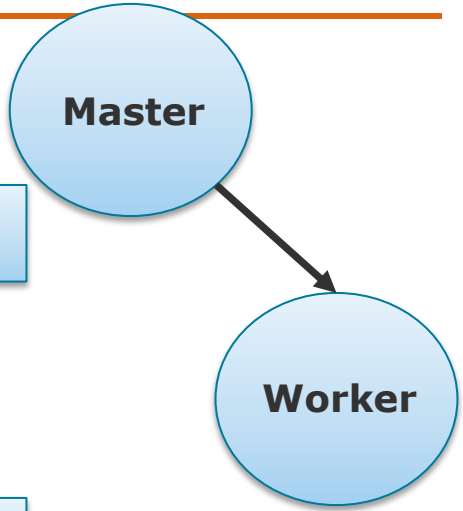
```
public class Master extends AbstractLoggingActor {
  public Master() {
    ActorRef worker = this.context().actorOf(Worker.props());
    this.context().watch(worker);
  }
  @Override
  public SupervisorStrategy supervisorStrategy() {
    return new OneForOneStrategy(3,
      Duration.create(10, TimeUnit.SECONDS),
      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

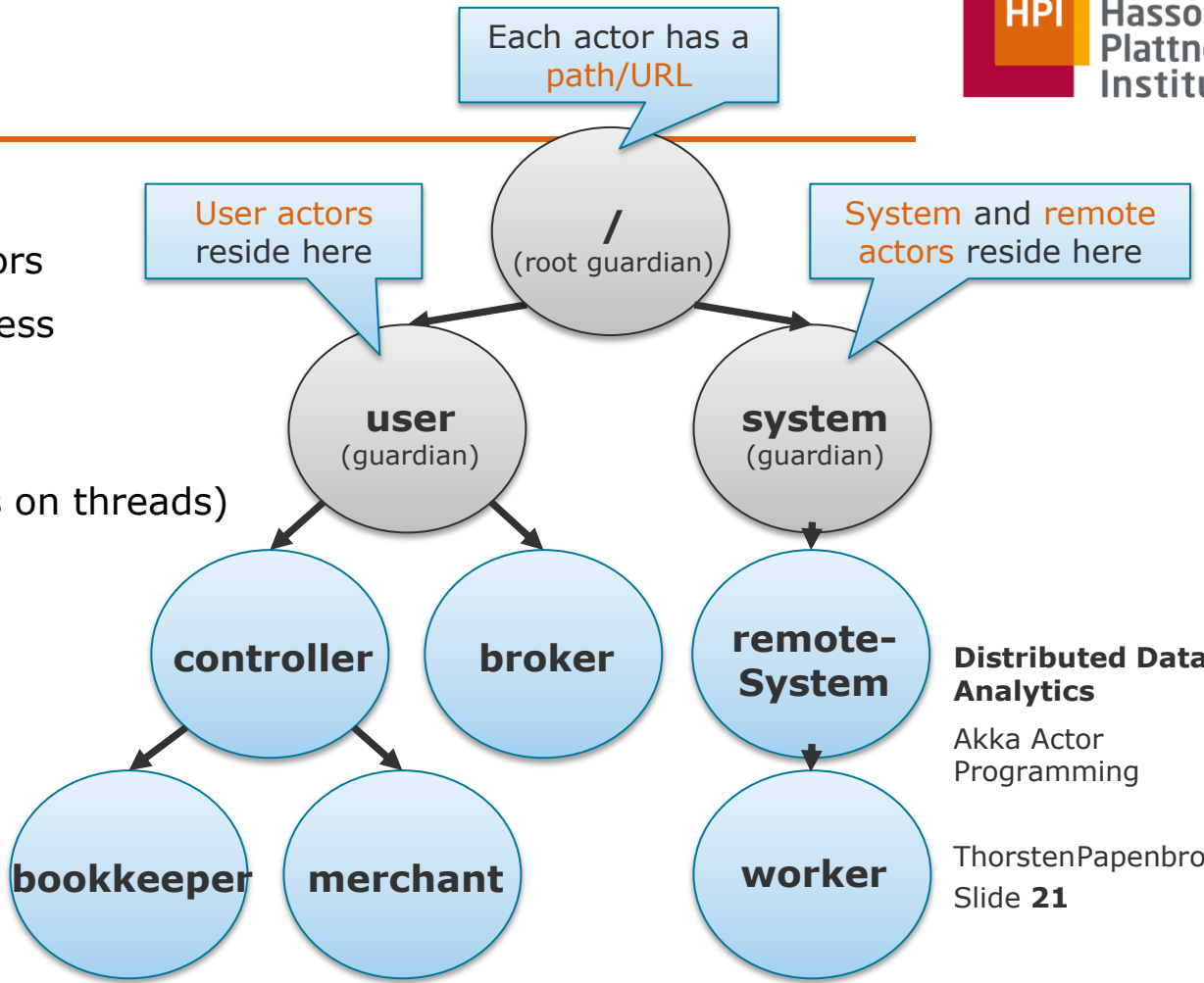
A factory pattern



# 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

Hasso Plattner Institut

Event stream

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

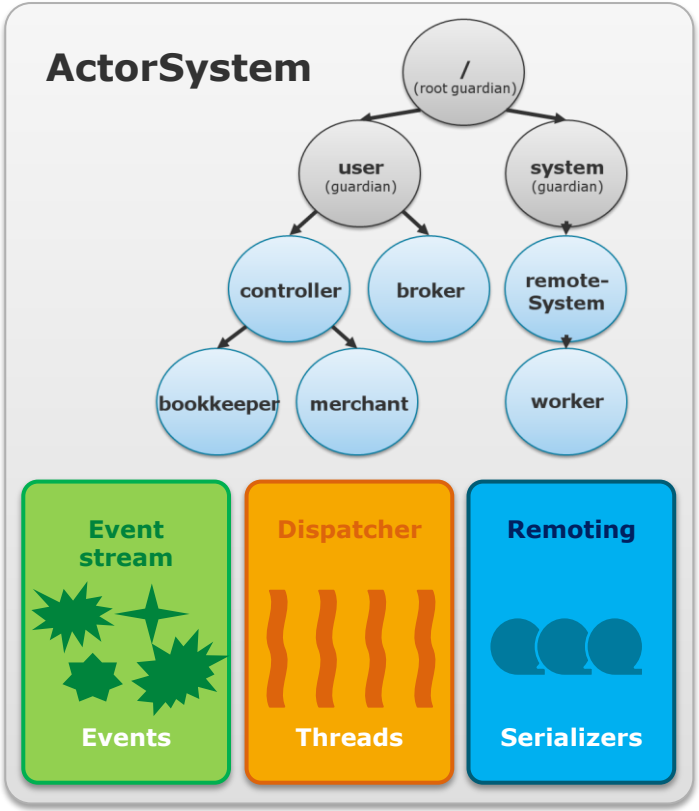
Dispatcher

- 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

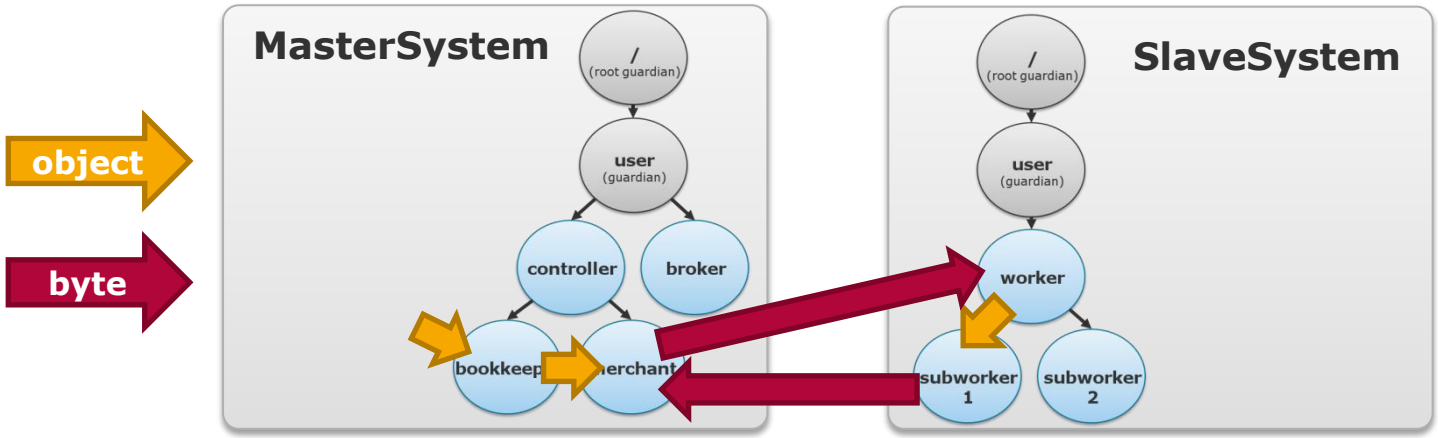
- Resolves remote actor adresses
- Sends messages over network
  - serialization + de-serialization



# Remoting

## Serialization

- Only messages to **remote actors** are serialized
  - Communication within one system: language-specific format
    - Java classes
  - Communication via process boundaries: transparent serialization
    - Serializable, Kryo, Protocol Buffers, … (configurable)



# Remoting

## remote.conf

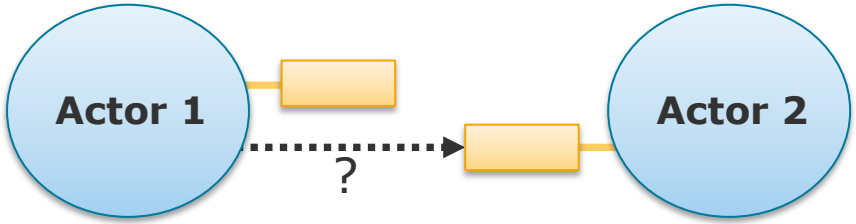
```
akka {
  actor {
    provider = remote
    serializers {
      java = "akka.serialization.JavaSerializer"
      kryo = "com.twitter.chill.akka.ConfiguredAkkaSerializer"
    }
    serialization-bindings {
      "java.io.Serializable" = kryo
    }
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "192.168.0.5"
      port = 7787
    }
  }
}
```

Sets the serialization class for remote messages

Configures transport protocol, hostname, and port for remote actor systems

# Actor Lookup

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



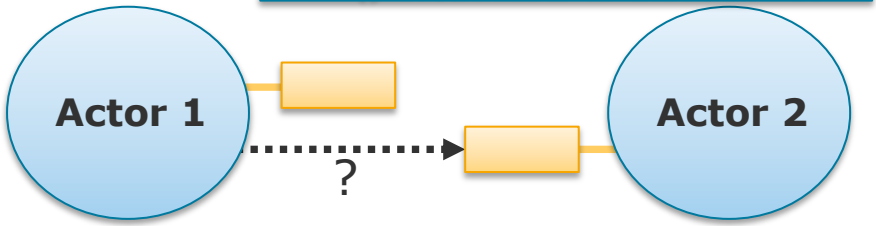

```
public class Master extends AbstractLoggingActor {
  private ActorRef worker;

  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

```
public class WorkerTest {

  private ActorSystem actorSystem;

  @Before
  public void setUp() {
    this.actorSystem = ActorSystem.create();
  }

  @Test
  public void shouldWorkAsExpected() {
    new TestKit(this.actorSystem) {{
      ActorRef worker = 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() {
    this.actorSystem.terminate();
  }
}
```

# 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()

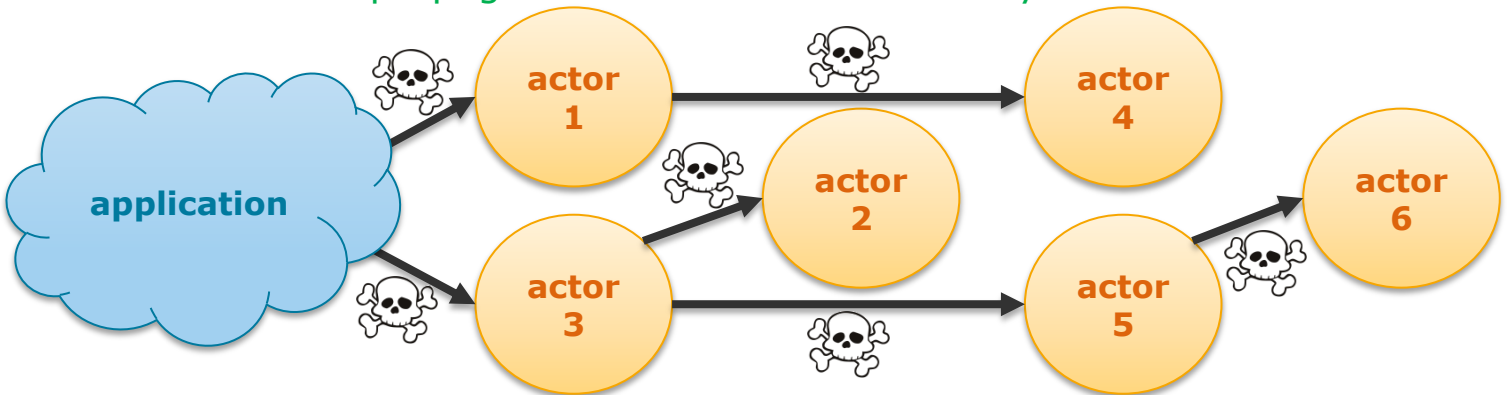
- Calling terminate() on an Actor System will stop() all its actors
- Problem:
  - Remote Actor Systems are still alive

# Application Shutdown

## 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



# Application Shutdown

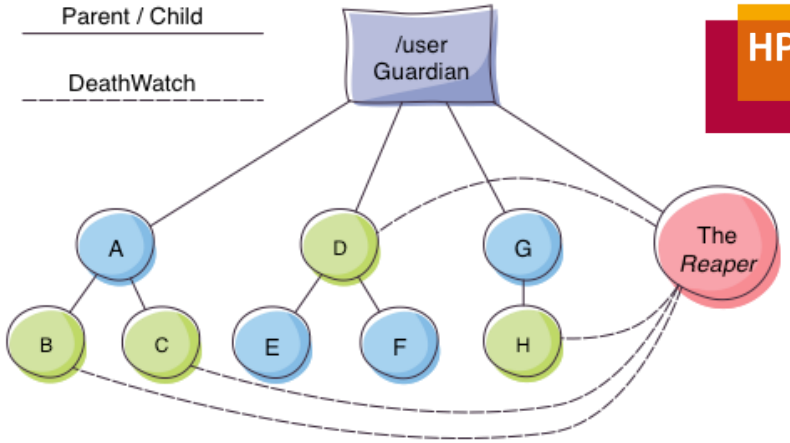
## PoisonPill Shutdown

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

```
import akka.actor.PoisonPill;

[…]

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

**PoisonPill** is an Akka message that is handled by all actors

# Application Shutdown




## 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 dieing

End the world

# Application Shutdown

## Reasons to die without a PoisonPill

- If an actor's parent dies, the orphaned actor dies too
- If a client looses its master Actor System, it might decide to die
- If an error occurs, the supervisor might choose to let the failing actor die
  - **Let it crash philosophy**
    - Failure recovery is usually easier for actors on higher abstraction levels

# Application Shutdown

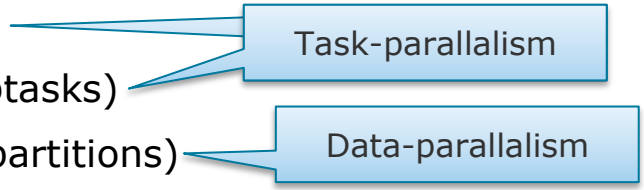
## Stop a running system

- 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

# 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 is delegated to a dynamically managed pool of worker actors




## Task Scheduling

- Strategies:
  - RoundRobinRoutingLogic
  - BroadcastRoutingLogic
  - RandomRoutingLogic
  - …
  - SmallestMailboxRoutingLogic
  - ConsistentHashingRoutingLogic
  - BalancingRoutingLogic

# Scheduler

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

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!

Strategy defines the worker to be chosen

## Task Scheduling

- Strategies:
  - RoundRobinRoutingLogic
  - BroadcastRoutingLogic
  - RandomRoutingLogic
  - …
  - SmallestMailboxRoutingLogic
  - ConsistentHashingRoutingLogic
  - BalancingRoutingLogic

# Large Messages

## Akka Messages

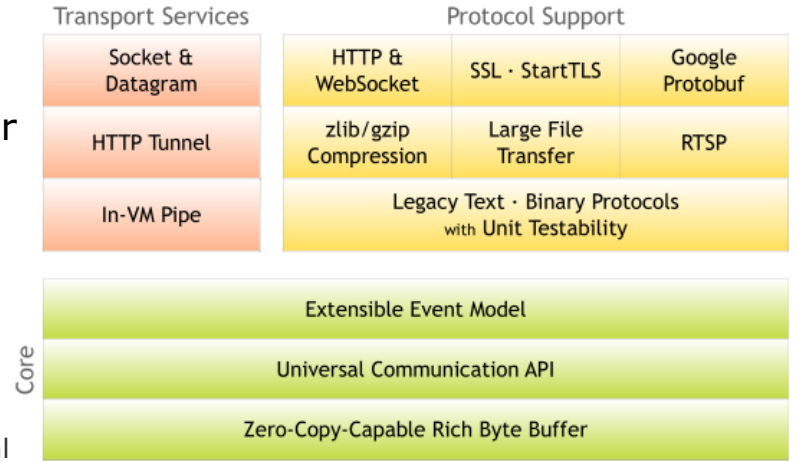
- Small, serializable objects
- What if we need to send large amounts of data over the network?
  - Use Netty for data transfer (one abstraction level deeper than Akka)
  - Send data references via Akka messages

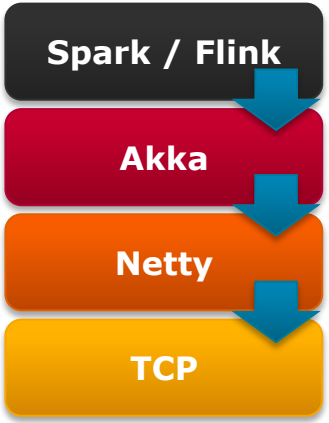
Alternative:
Use the Akka Streaming module!

## Netty

- A (non-)blocking I/O client-server framework for large file transfer

See: https://netty.io/wiki/user-guide-for-4.x.html

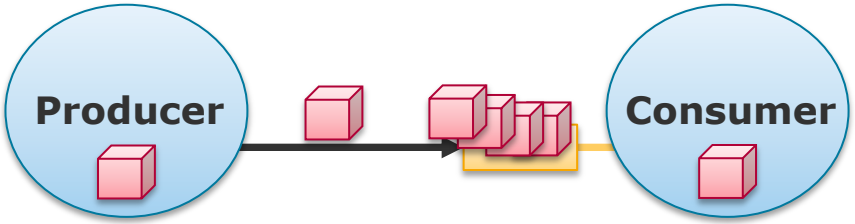


# Pull vs. Push

## 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

# 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
- 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 Scala Interface

## A Master-Worker Example

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

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

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

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

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

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

# Some Further Nodes

## Redundant API Calls

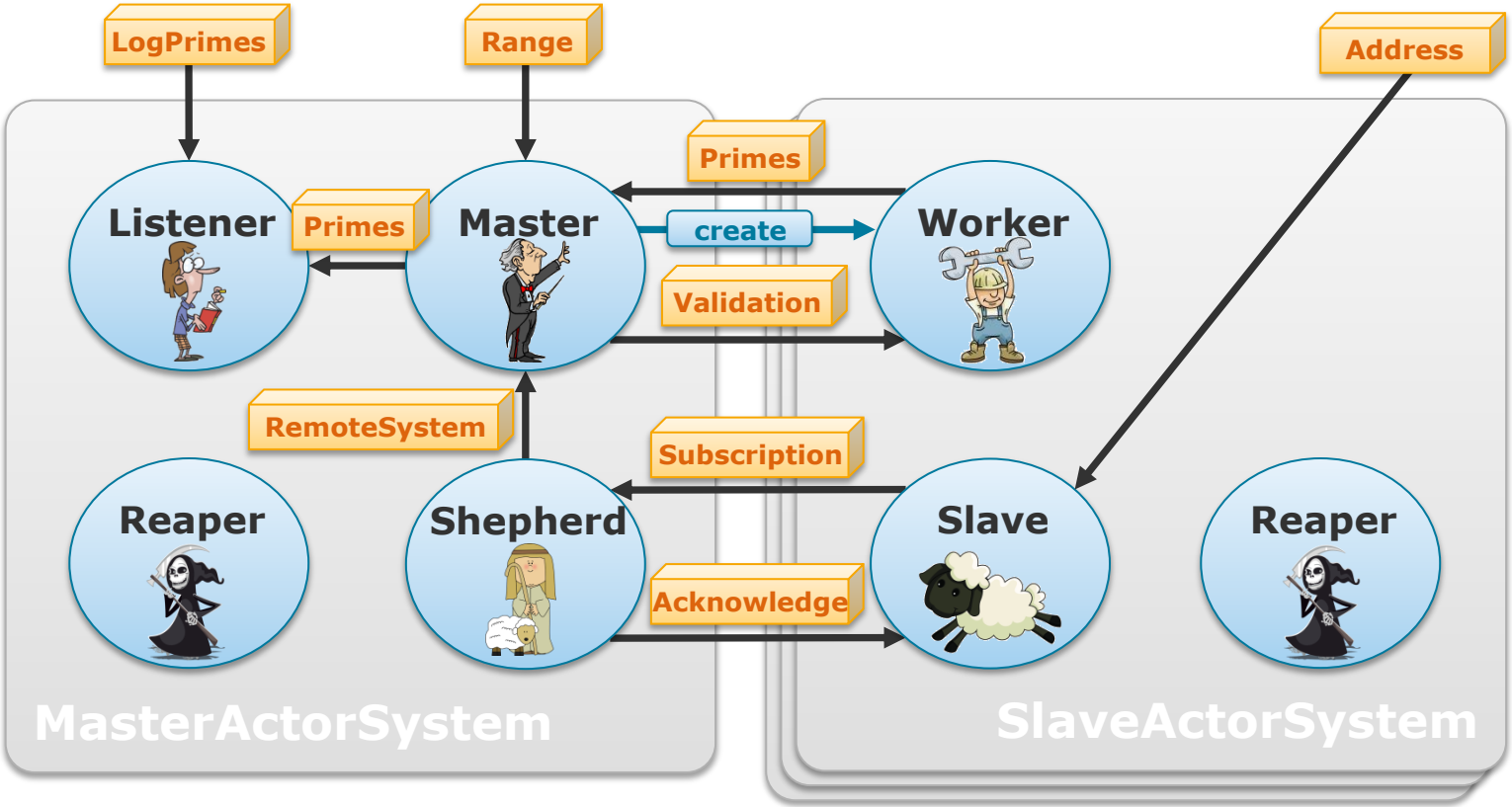
- Due to Java-Scala interface mix
  - this.getContext() = this.context()
  - this.getSender() = this.sender()
  - …

## Blocking Calls

- Ask pattern (depricated – do not use!)
  - Java: someActor.ask(message)
  - Scala: someActor ? message

# Live Demo

```
master --host <IP-Master>
slave --host <IP-Slave> --master <IP-Master>
```



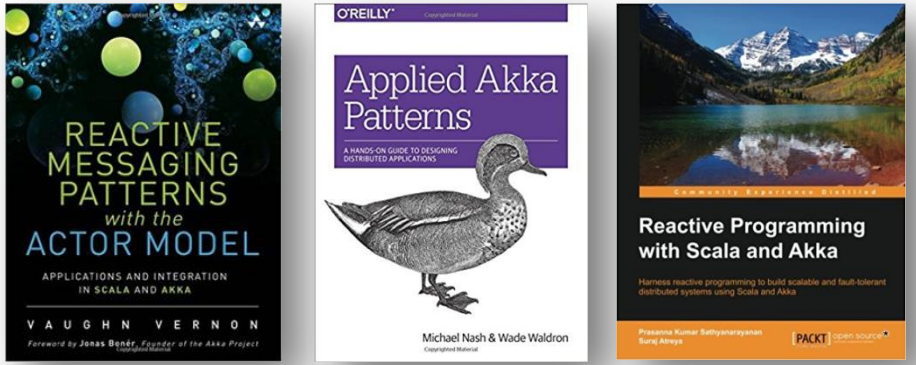
# Further Reading

Akka documentation

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

Experiences, best practices, and patterns

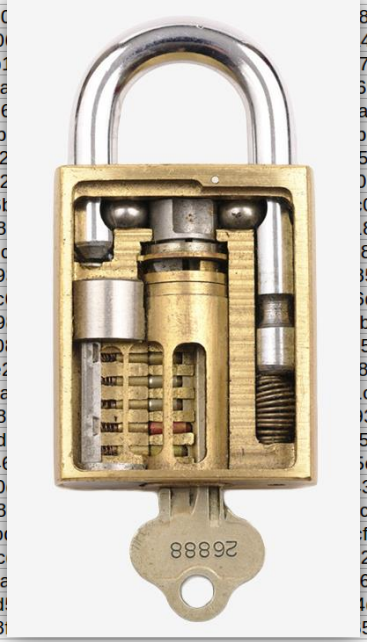
- http://letitcrash.com
- http://akka.io/blog
- https://github.com/sksamuel/akka-patterns
- Akka actor programming literature:

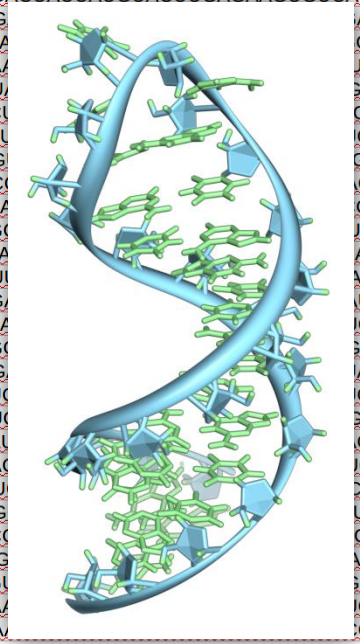


Akka

# Homework – Our Dataset

| ID | Name | Password | Gene |
|---|---|---|---|
| 0 | Adrian Christoph | 73fdb9b9ce907e6c3629e2da8d245667d6bfc096b778fa29f8c82492212c27ad | CAAGCGACCUCAGAUCGUUGGCCUUCACCCGCACAGCGGUUGCCGCGUAUAAGGGGUA |
| 1 | Adrian Werner Gerhard | 4d17f1ff2e5b4e727c1cb8ee6bac35e75d53c57ba4d443623279d4043c334af1 | UUCGGACGAGUUCUAUUCCUCGAAUUUUCCAACGGAUACAAGCAUGAAUGUAGGCACUA |
| 2 | Alexander | b0394d9a1d1a6f4dd365f3c96a342eb3578d1220621614d38105076eaf61c4b3 | CCAGGGAUUAUCUUCCGGAGCGUCGACACUCAUUGUCUCGCCAUUAACUACACGAUUUA |
| 3 | Axel | f9bdd9647477313O7f05c8c368fbd67c6470b045e23c717eb0b0136522b5bf8a | UAGUAGAGUAUCUGUUCUGGACUAUUAUGUACUUGAGAAGUGUCAUGCUUUAGUCCCGC |
| 4 | Christoph | 90895995e9c899( 82b17488e6d3d8a621 | AGGGUGUUGAAUAGUGUCUG AGAGUUCGGUACCC |
| 5 | Daniel Ralf | 86a85f0e7eaaffe( 44044f877890522 | GAAACCCGGAUCUUAUUAACA GAGGACUAAGCUAC |
| 6 | Daniel Simon | 69b135098b945b1 74edf8249d608a1927 | AACUGCCUAUUAUGAGUGGAA JGCGAUCCGAUGUG |
| 7 | Elena | 83a04d67cdc5d7a 59fb4f11b9750a3d1 | UUCAGCCCCAGCACGGGAUUA GAGAGCUAAUUGCAC |
| 8 | Fabian | 485bf901f89b6376 af0a9dd7bce96044 | AGGUGAAGUAAAGCAGCCUCU GGUGUCCCAGACGC |
| 9 | Fabian Louis | 26747909f192deb bea9e0c2a48cb6997 | UUGUGGUCAAAUGUAAACAUU GGCUGGAAGAUGGA |
| 10 | Falco Dominic | 5f65ff719d37fb742 566318b9460f8 | CCCGACGAGGAAUGGUCCUG GGUUUGUAGCGUCL |
| 11 | Felix | c3d459c825154b2 0753be15e8450876c8 | UUGAACGGCUUUGGCAAAGC CGUUGCUAGUUAAC |
| 12 | Frederic | 326e4821636d86b 0c95a7376eaf36c85 | GAUAAUCUGCAAUCGACCUUA CGUGACCUAUAUUA |
| 13 | Frederike | 2e92e003005f508 80310627e7fb47ef | GGGAAAAGGAACCACCCGCGC JAGAUUCAUAUUAAG |
| 14 | Jan | e38de9abc1e162c 8eef2d9068ffef60e | AUUUACAGUGUUCUAAAACUA AACCCUGUACAAGU |
| 15 | Janos | 69c81a5e499fa79 859afe349debfa23 | AAACACCGAGUAGGAGGGGUU AUAUGCUACGUUUG |
| 16 | Janusch | 69e277fc58d19bc 5d504616810c8c059 | CAGUACCGGUUCAUUCUGAG UCUCUUCGCAGGGC |
| 17 | Jonas | bf684b40c26ee09 b6ae7c2405c557401 | AGGUACAGGGUAAUUCAACAA ICACCGAAAACGAUG |
| 18 | Jonas | e5358d9a9c6cbf0 52072f79d903ea0 | ACUAGCGUGUGAGCAACCAGC GACUCCGAGCGGAC |
| 19 | Juliane Katharina | 4d12ae8d8154ee2 84eb6004e459008c0 | CUGGAAGGAUCACGCAGGUG IACAACGAGAGAACC |
| 20 | Konstantin Max | ac7349c678a263a c1544bd68589e4deb | GUUAAAAGGCGGAAUCUCCUC CGUAUGAUCUUAAU |
| 21 | Lawrence Elliott | f81d59339451868 3b4583ebe3f1e1bfc | CCCUCAUUGCGUCAGUACCUC GAUGAGUGAAAUCUC |
| 22 | Lukas Jürgen | 9f1a659444d441d 5c3733317f41eea08 | GAUGAAGUCAAGCUGAAUGAU GACUGAGGGGUCUA |
| 23 | Maxi | 3d9014a297aad46 5e5bd5b18224487cb | CGACAACCGUGCGCACCCCAC UCUGAGGCGCCUUG |
| 24 | Nils Dimitri | 3aad27f05312ec0 35bb0526bcb2fd44 | UGACUUUUGGGACCGUGCAU CGCAUGCUCACUUG |
| 25 | Philipp | d08eab1b6f786b8 c5484774fcc3cd980 | AAGUGGCCAGUUACGGGCCG UCAUGUAUGAGAUG |
| 26 | Philipp Konstantin | dfc00113cc9d9ab fa4a7f72fe48f80f | CAUUCGAUCGAGGUAACCCCC CUGGGACUCUCAUA |
| 27 | Sebastian | 2e322243222f72c 2add7ec7593f9b19 | AUGCGCGUGCGUCUGUGUCG AUAUUUGGCAUCC |
| 28 | Sven Patrick | d41dc38e72c26ba 66979d0da6a7970df | GCUAUUAGGAACAAUCUUGGU GGUCUUGUCUCCA |
| 29 | Svetlana | a67f2af79bacef6d 4d2e8e5e754ec860 | AAACCCGGCAGGCCUGCAUAA IGAAUCUUACGAAUA |
| 30 | Tobias | 8feb487ece589c8 597bd702bc873 | UCUCUGCAUGUGGUUCACUA CAACUGGGACGUAC |
| 31 | Victor | 92d2ef94d290b8d578335f57dca35ec5aa276c770598d4c5be7de7f431df82f1 | CCGCGAUACCGACGUUGCCAGAACGUUUGGGGAGACGUCGCACACAAAGCACGUUAAAC |
| 32 | Willi | 7d12d0bb4331caed2ca2900b2c29bd6bf485e08338eabbc45adca13ab0561d51 | CGCUGGUGUCCAGGCCGUGGGGGCGGAAUCCCCUAUGAACUCUCGUUUGCAUACAACU |
| 33 | Zenko | 80a98a5f8890151b389734f08d9d9e69a7d5a516d578525cfc0b2ef0e57cbc7d | CAUCAGACUGAUCAAAUCAGUUGGCAAAGGGGCCUGAUGUGCAAUUACCCGUCGAAGAA |



# Homework

## Assignment

- Write an Akka application that answers the following two questions:
  1. Password Cracking: What are the students' clear-text passwords?
  2. Gene Analysis: Which student pairs share the longest gene sub-strings?


# Homework

## Assignment

- Write an Akka application that answers the following two questions:
  1. **Password Cracking: What are the students' clear-text passwords?**
  2. Gene Analysis: Which student pairs share the longest gene sub-strings?
- Hints:
  - All passwords are numeric and have length seven
  - The password encryption algorithm is SHA-256

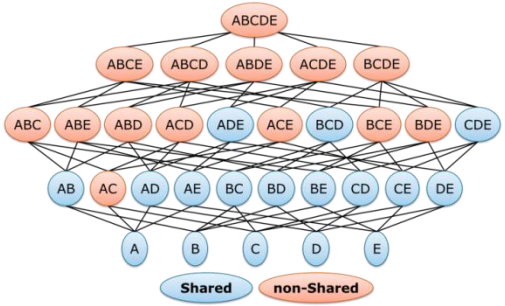
```
private String hash(String line) {
  MessageDigest digest = MessageDigest.getInstance("SHA-256");
  byte[] hashedBytes = digest.digest(line.getBytes("UTF-8"));
  StringBuffer stringBuffer = new StringBuffer();
  for (int i = 0; i < hashedBytes.length; i++)
    stringBuffer.append(Integer.toString((hashedBytes[i]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

# Homework

## Assignment

- Write an Akka application that answers the following two questions:
  1. Password Cracking: What are the students' clear-text passwords?
  2. **Gene Analysis: Which student pairs share the longest gene sub-strings?**
- Hints:
  - We need to check all pairs of records for their longest common sub-strings
  - Possible algorithms:

(a) Apriori-gen

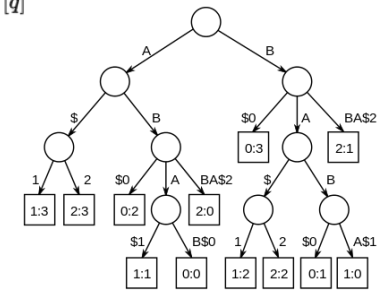



(b) Dynamic programming

$$LCSuff(S_{1..p}, T_{1..q}) = \begin{cases} LCSuff(S_{1..p-1}, T_{1..q-1}) + 1 & \text{if } S[p] = T[q] \\ 0 & \text{otherwise.} \end{cases}$$

| | | A | B | A | B |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 0 | 1 |
| A | 0 | 1 | 0 | 2 | 0 |
| B | 0 | 0 | 2 | 0 | 3 |
| A | 0 | 1 | 0 | 3 | 0 |

(c) Suffix tree




(d) …

# Homework – Solution Excerpt

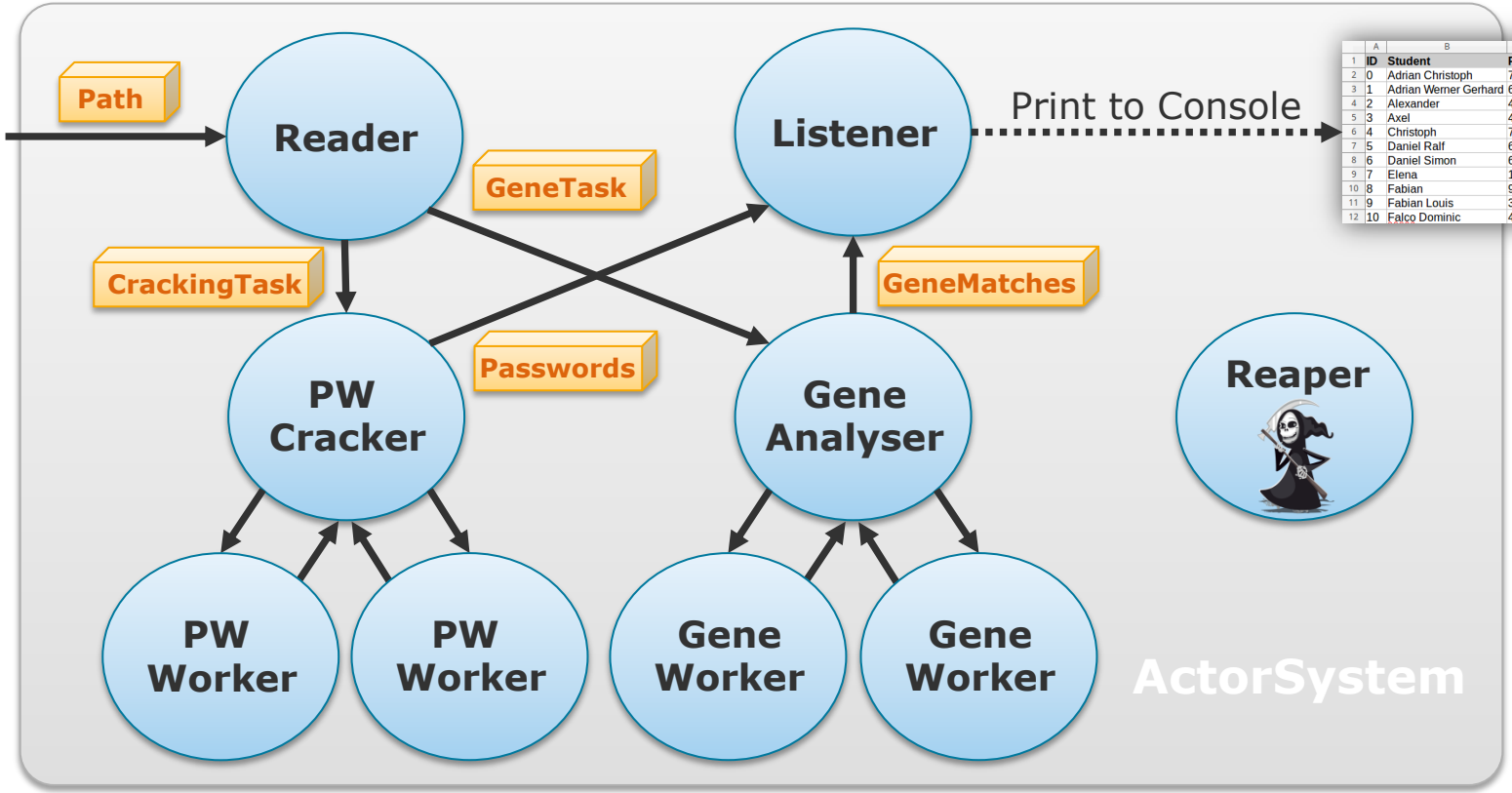
| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | **ID** | **Student** | **Password** | **Gene_Partner** | **Longest_Gene_Match** |
| 2 | 0 | Adrian Christoph | 7240492 | 17 | CUCCUGCUUGUUUAAA |
| 3 | 1 | Adrian Werner Gerhard | 6221100 | 4 | UACAUUACACGGAG |
| 4 | 2 | Alexander | 4800375 | 12 | CUGAUGGCUUAACCC |
| 5 | 3 | Axel | 4183998 | 21 | AUCGUCAAAGUCAGC |
| 6 | 4 | Christoph | 7355710 | 11 | GGCGUAUGCCCUCAC |
| 7 | 5 | Daniel Ralf | 6354693 | 20 | CGCAGAUGGCGUAUGA |
| 8 | 6 | Daniel Simon | 6163213 | 24 | CGAGCGCUGUCGAAA |
| 9 | 7 | Elena | 1076856 | 9 | UGGGGGCCAGACGCC |
| 10 | 8 | Fabian | 9412114 | 0 | GAAGUCUCUCCAAUG |
| 11 | 9 | Fabian Louis | 3737060 | 7 | UGGGGGCCAGACGCC |
| 12 | 10 | Falco Dominic | 4955316 | 31 | GAGGGAGAUGAAUGAUU |

# Homework – Rules

## Assignment

- Submission deadline
  - 22.12.2017 23:59:59
- Submission channel
  - Email to thorsten.papenbrock@hpi.de
- Submission artifacts
  - Source code as zip (Maven project; Java or Scala)
  - Jar file as zip
- Parameter
  - "java -jar YourAlgorithmName.jar --path students.csv"
  - Default path should be "./students.csv"
- Teams
  - Please solve the homework in teams of two students
  - Provide the names of both students in your submission email

# Homework – Example Architecture




| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | ID | Student | Password | Gene_Partner | Longest_Gene_Match |
| 2 | 0 | Adrian Christoph | 7240492 | 17 | CUCCUGCUUGUUUAAA |
| 3 | 1 | Adrian Werner Gerhard | 6221100 | 4 | UACAUUACACGGAG |
| 4 | 2 | Alexander | 4800375 | 12 | CUGAUGGCUUAACCC |
| 5 | 3 | Axel | 4183998 | 21 | AUCGUCAAAGUCAGC |
| 6 | 4 | Christoph | 7355710 | 11 | GGCGUAUGCCCUCAC |
| 7 | 5 | Daniel Ralf | 6354693 | 20 | CGCAGAUGGCGUAUGA |
| 8 | 6 | Daniel Simon | 6163213 | 24 | CGAGCGCUGUCGAAA |
| 9 | 7 | Elena | 1076856 | 9 | UGGGGGCCAGACGCC |
| 10 | 8 | Fabian | 9412114 | 0 | GAAGUCUCUCCAAUG |
| 11 | 9 | Fabian Louis | 3737060 | 7 | UGGGGGCCAGACGCC |
| 12 | 10 | Falco Dominic | 4955316 | 31 | GAGGGAGAUGAAUGAUU |

**Distributed Data Analytics**

Akka Actor Programming


"I wait for green"
"Road ahead is free!"
"I wait for crossing traffic"
"I accelerate!"
"You are not in my path!"
"Attention, I break!"