QAWARE
SOFTWARE ENGINEERING

Hochschule **Rosenheim**
University of Applied Sciences

We Are Reactive

Cloud Computing

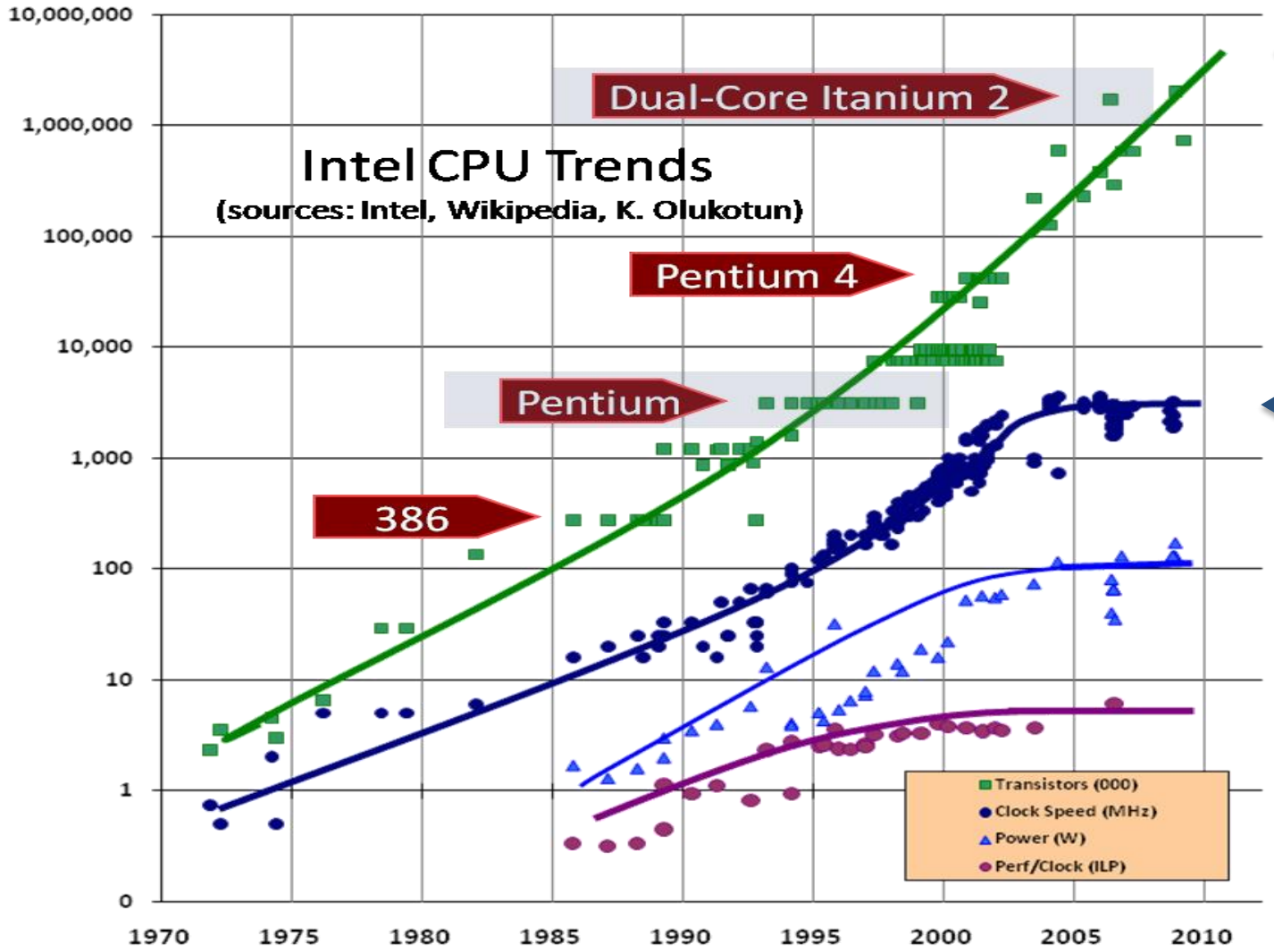
Kapitel 2: Programmiermodelle

Mario-Leander Reimer

mario-leander.reimer@qaware.de

Rosenheim, 16.10.2017

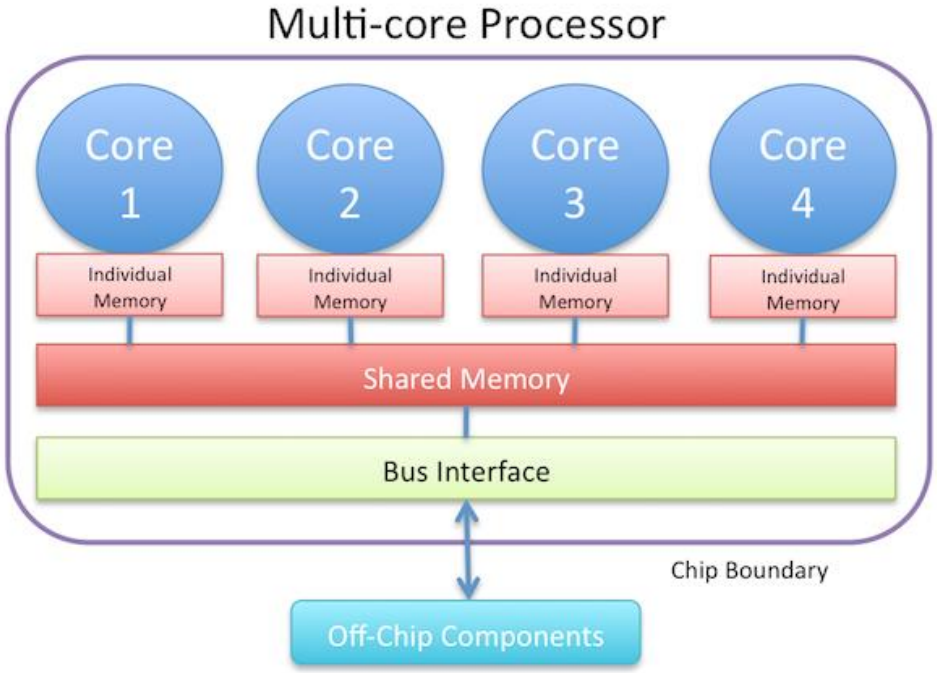
„The free lunch is over“: Es gibt keine kostenlose Performanzsteigerung mehr – Nebenläufigkeit zählt.



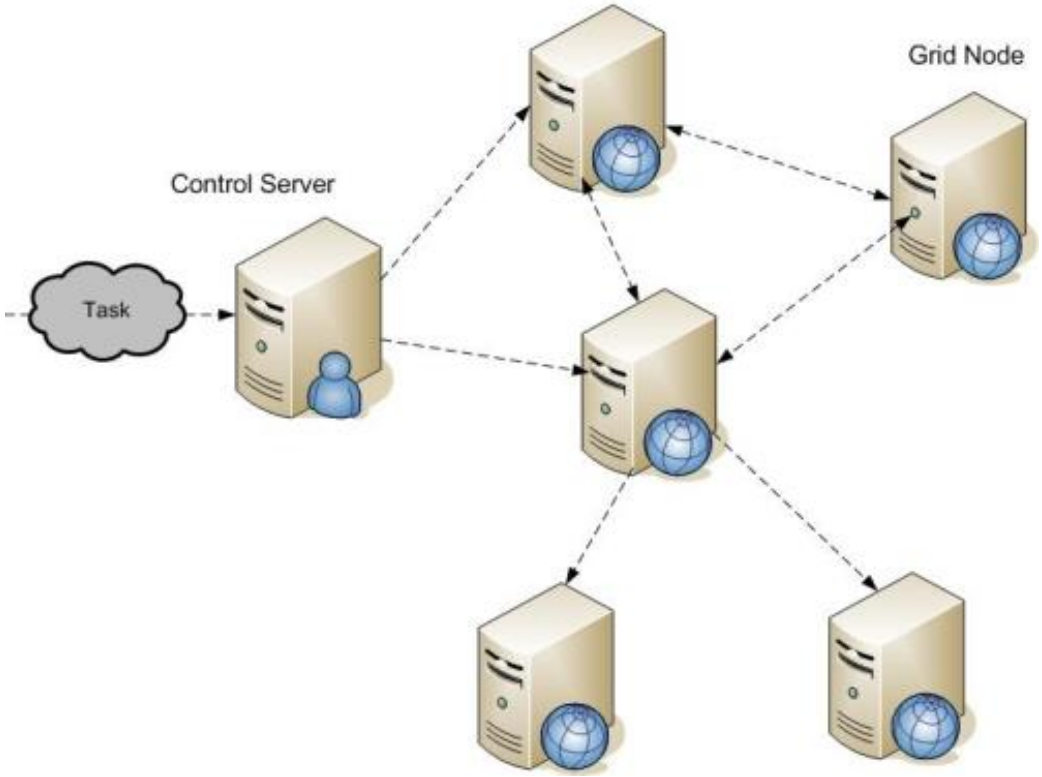
Anzahl Transistoren
Moore‘s Law gilt weiterhin

Taktfrequenz
Seit 2004 ist die Taktfrequenz von CPUs konstant

Nebenläufigkeit kann im Kleinen und im Großen betrieben werden.

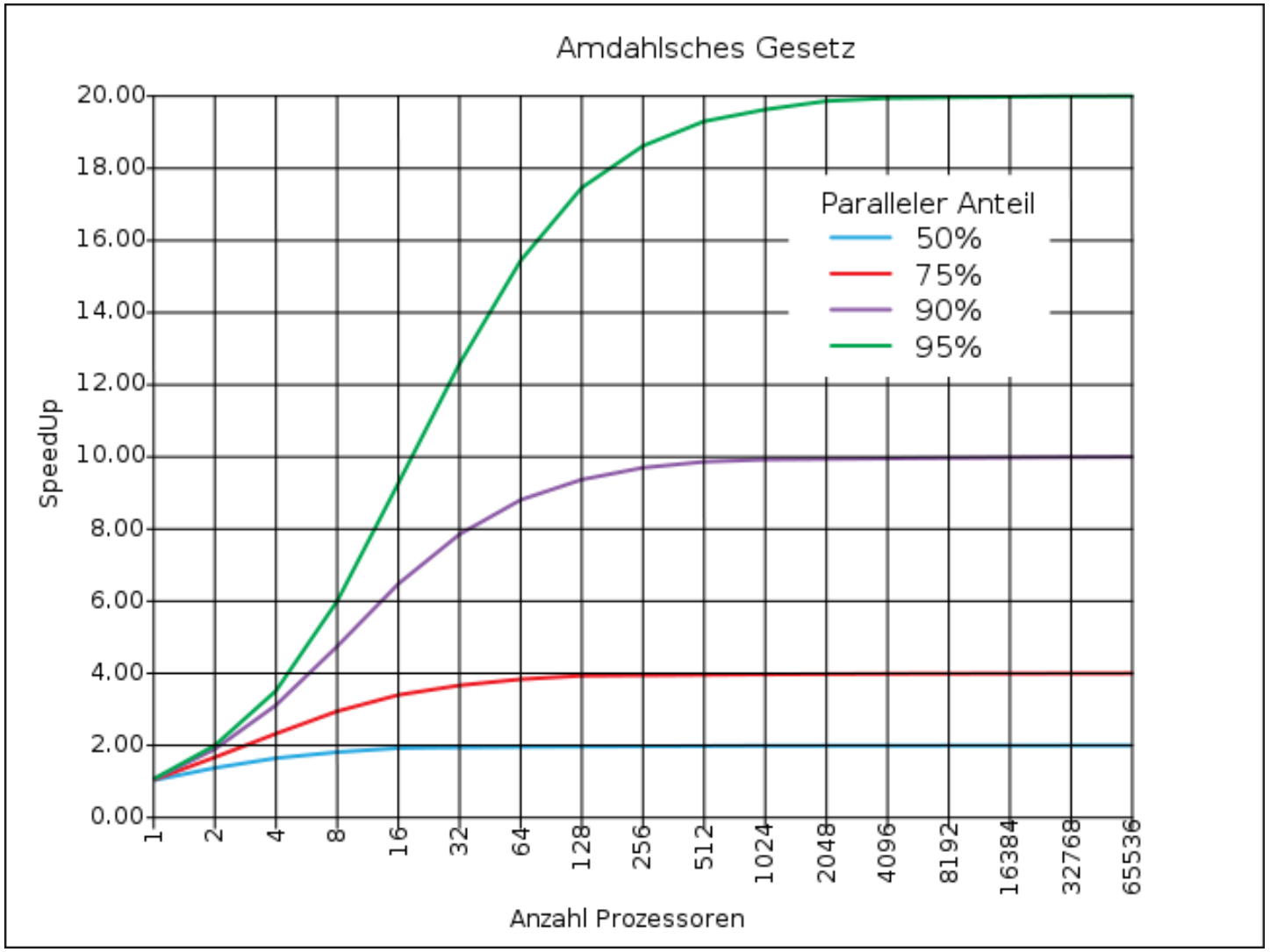


Multi Core



Multi Node
(Cluster, Grid, Cloud)

Das Amdahlsche Gesetz: Die Grenzen der Performanz-Steigerung über Nebenläufigkeit.

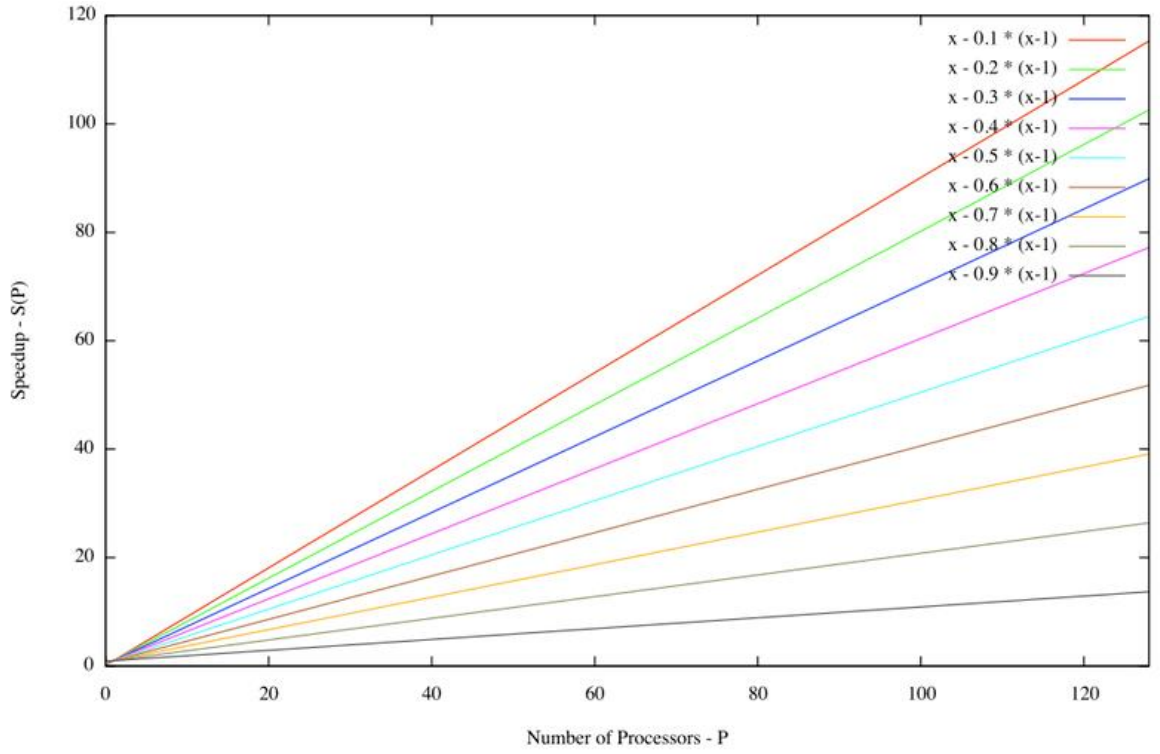


P = Paralleler Anteil
S = Sequenzieller Anteil
N = Anzahl der Prozessoren
Speedup = Maximale Beschleunigung

$$Speedup = \frac{1}{1-P}$$ für N = ∞

$$Speedup = \frac{1}{\frac{P}{N} + S}$$

Das Gustafsons Gesetz: ist bei großen Datenmengen jedoch oft passender.



$$Speedup = \frac{1}{\frac{P}{N} + \cancel{S}}$$

- **Annahme**: Der parallele Anteil P ist linear abhängig von der Problemgröße (i.W. der Datenmenge), der sequenzielle Anteil hingegen nicht.
- Beispiel: Mehr Bilder ➔ Mehr parallele Konvertierung
- Gesetz: Steigt der parallele Anteil P linear (oder mehr) mit der Problemgröße, so wächst auch der Speedup linear

re·ac·tive adjective \rē-'ak-tiv\

1 of, relating to, or marked by reaction or reactance
2 readily responsive to a stimulus

Das Programmiermodell der Cloud: Functional Reactive Programming

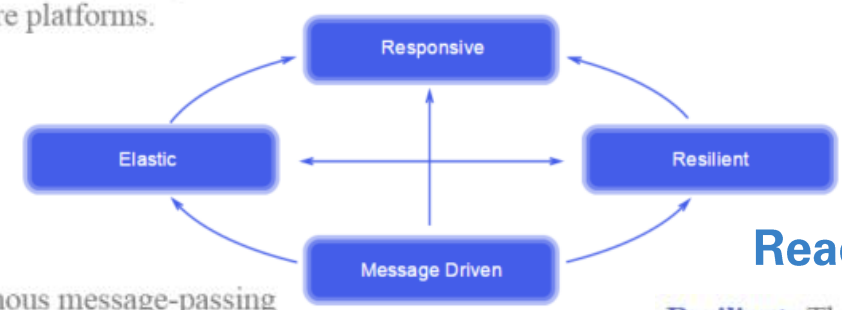
Das Reactive Manifesto

React to load

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

React to users

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.



React to events / messsages

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

React to failures

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

Functional Reactive Programming: Das Programmiermodell mit dem das Reactive Manifesto umgesetzt werden kann.

Dekomposition in Funktionen (auch Aktoren)

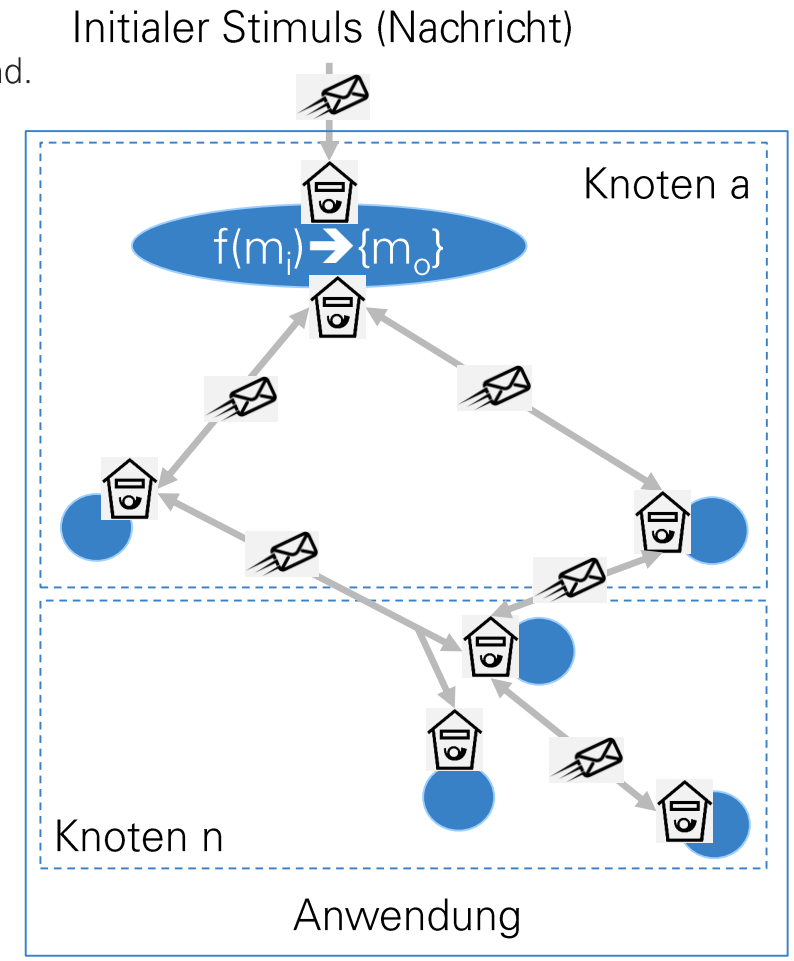
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wiederaufsetzbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)

Kommunikation zwischen den Funktionen über Nachrichten

- asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber nicht auf sie.
- Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Producern und 1 Consumer)
- Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und sind unveränderbar

Elastischer Kommunikationskanal

- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden (mehr oder minder) zuverlässig zugestellt
- Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)



Gegenüber synchronen Systemen bietet FRP einige kontextsensitive Vorteile.

- **Vorteil: Höhere Prozessorauslastung**
  - Der Prozessor befindet sich deutlich weniger Zeit in Wartezuständen (IO-Wait, Lock-Wait, …).
  - Diese Zeit wird für Berechnungen frei.
  - **Für den Fall**, dass die Anwendung auch genügend Berechnungen durchführen kann.

- **Vorteil: Höherer Parallelisierungsgrad**
  - Damit wird auch ein höherer Speedup möglich.
  - **Für den Fall**, dass die Logik und die Datenmenge sich entsprechend partitionieren lassen.

Functional Reactive Programming am Beispiel

Darf ich vorstellen: akka.

Open-Source Java & Scala Framework für Aktor-basierte Entwicklung.

Ziel: Einfache Entwicklung von

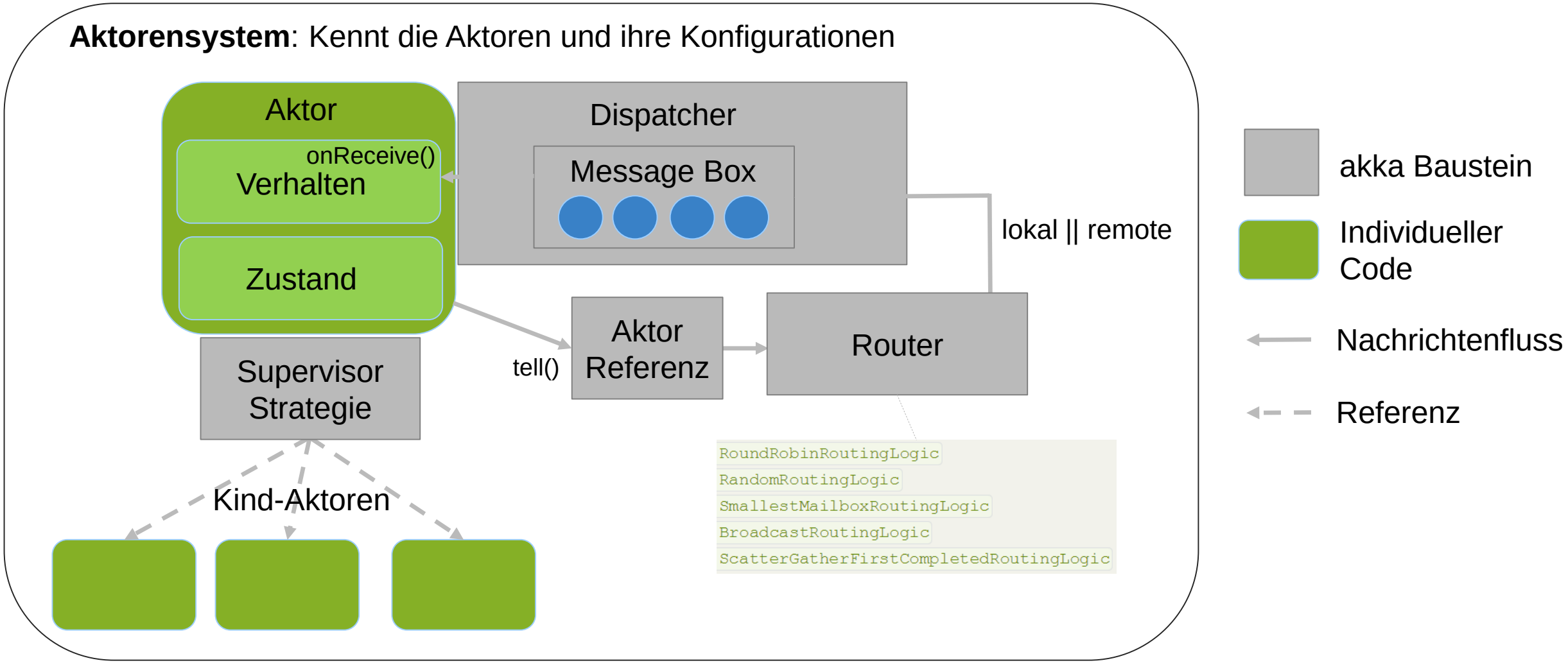
- funktionierender nebenläufiger,
- elastisch skalierbarer
- und selbst-heilender fehlertoleranter Software.

Start der Entwicklung 2009 durch Jonas Bonér im Umfeld Scala inspiriert durch das Aktor-Modell der Programmiersprache Erlang.

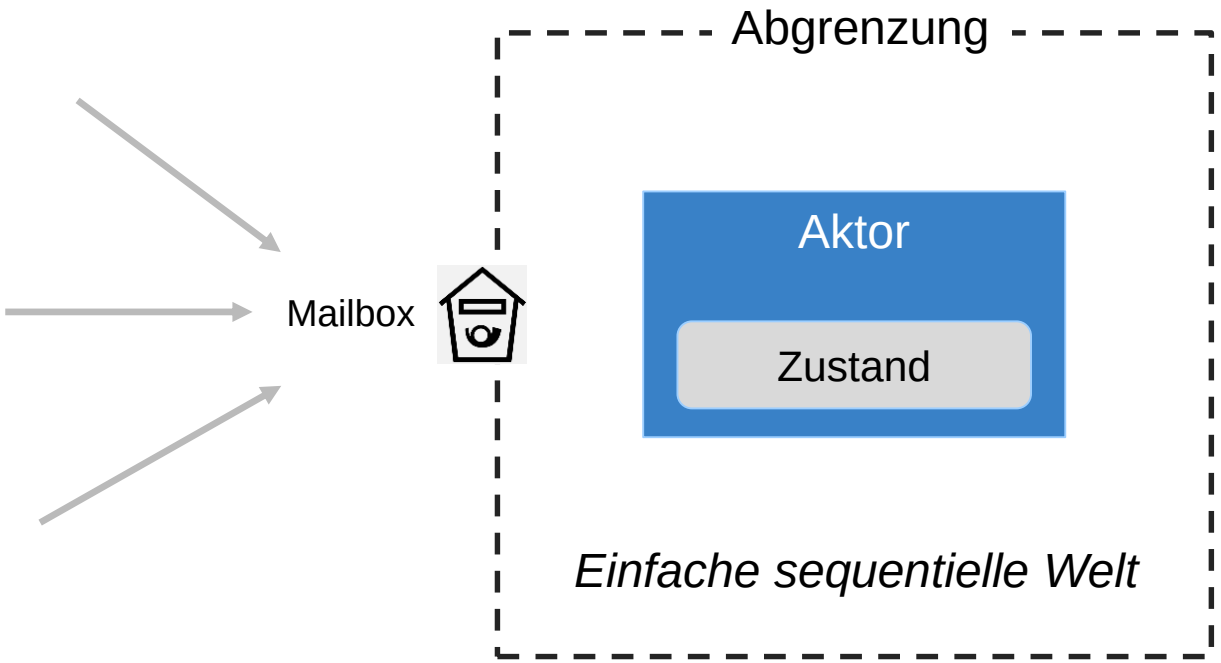
http://akka.io

Die Grundkonzepte von akka.

Konfiguration Aktor = Aktor-Klasse + Aktor-Name + Supervisor-Strategie + Dispatcher + Lokalität
Konfiguration Aktor Referenz = Aktor-Name + Router



Ein einzelner Aktor ist Single-Threaded und ohne weitere Synchronisation Herr über seinen Zustand.




```
private synchronized void updateSomeState() {

}
```

Große komplexe parallele Welt

Im Gegensatz zu Threads sind Aktoren leichtgewichtig.

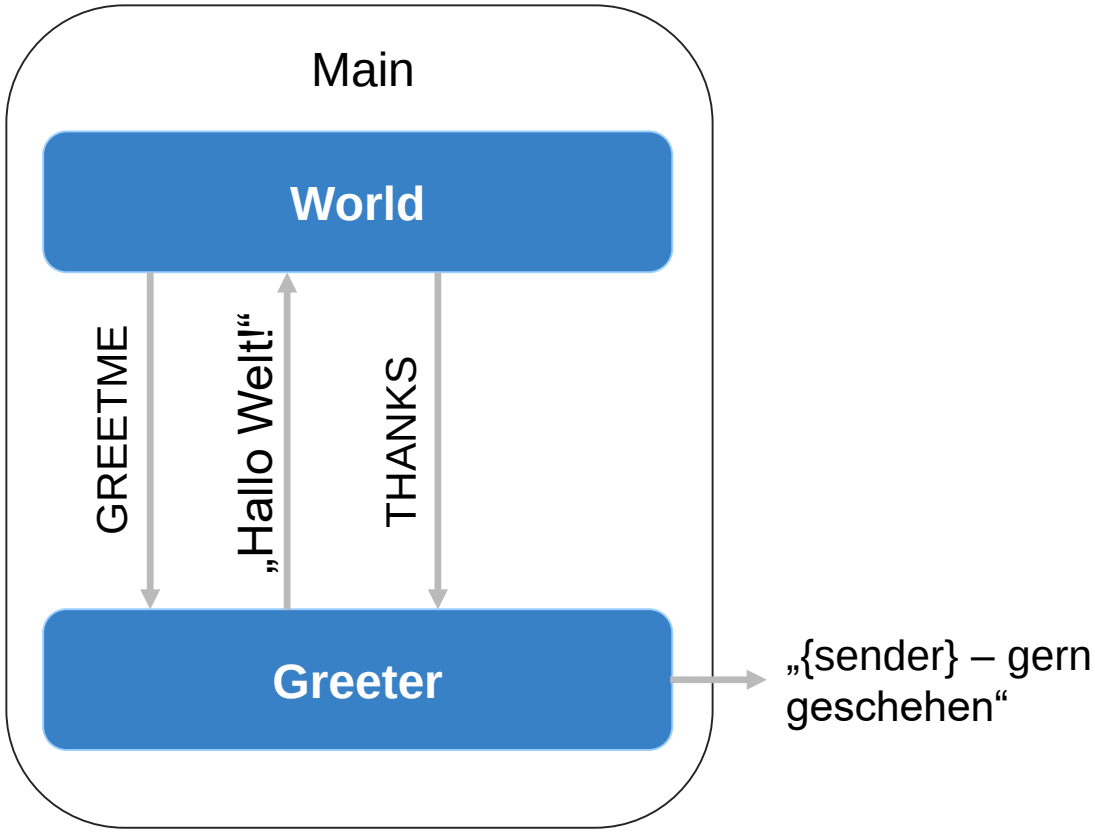
Eine Million Aktoren? Kein Problem!

Eine Million Threads? Probiert‘s mal aus!

Warum?

- Threads mappen direkt auf Ressourcen im System, und die sind begrenzt.
- Die Parallelität wird durch die Mailboxen gepuffert. Die Anzahl der für die Mailboxen notwendigen Threads ist viel kleiner als die Anzahl der Mailboxen.
- Aktoren sind reine Objekte und keine Threads. Sie haben aus diesem Grund nur ihren Speicher- und Laufzeit-Overhead.

Beispiel: Hello World



akka Hello World: Die Klasse *Greeter*

```
public class Greeter extends UntypedActor {

    public static enum Msg {
        GREETME,
        THANKS
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message == Msg.GREETME){
            getSender().tell("Hallo Welt!", getSelf());
        } else if (message == Msg.THANKS){
            System.out.println(getSender().toString() + " - gern geschehen");
        }
        else {
            unhandled(message);
        }
    }

}
```

akka Hello World: Die Klasse *World*

```
public class World extends UntypedActor{

    @Override
    public void preStart() {
        ActorRef greeter = getContext().actorOf(Props.create(Greeter.class), "greeter");
        greeter.tell(Greeter.Msg.GREETME, getSelf());
    }

    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof String){
            System.out.println(message);
            getSender().tell(Greeter.Msg.THANKS, getSelf());
        } else {
            unhandled(message);
        }
    }

}
```

akka Hello World: Die Klasse *Main*

```
public class Main {
    public static void main(String[] args) { akka.Main.main(new String[] {World.class.getName()}); }
}
```

Actor Best Practices

Do not block!

- Ein Aktor erledigt seinen Teil der Arbeit, ohne andere unnötig zu belästigen. Er gibt Arbeit für andere als Nachricht an diese weiter.

  Belege keinen Thread, um auf externe Ereignisse zu warten.

Pass immutable messages!

- Eine Nachricht darf syntaktisch ein beliebiges Objekt sein. Verknüpfe sie aber nicht mit dem Zustand des Aktors.

Let it crash!

- Wenn ein Aktor seine Aufgabe nicht ausführen kann, darf er den Fehlschlag melden. Sein Supervisor sollte eine geeignete Strategie implementieren, die mit dem Fehler umgeht.

Parental Supervision in Akka

Wenn ein Aktor nicht mehr weiterkommt, kann er eine Exception werfen.

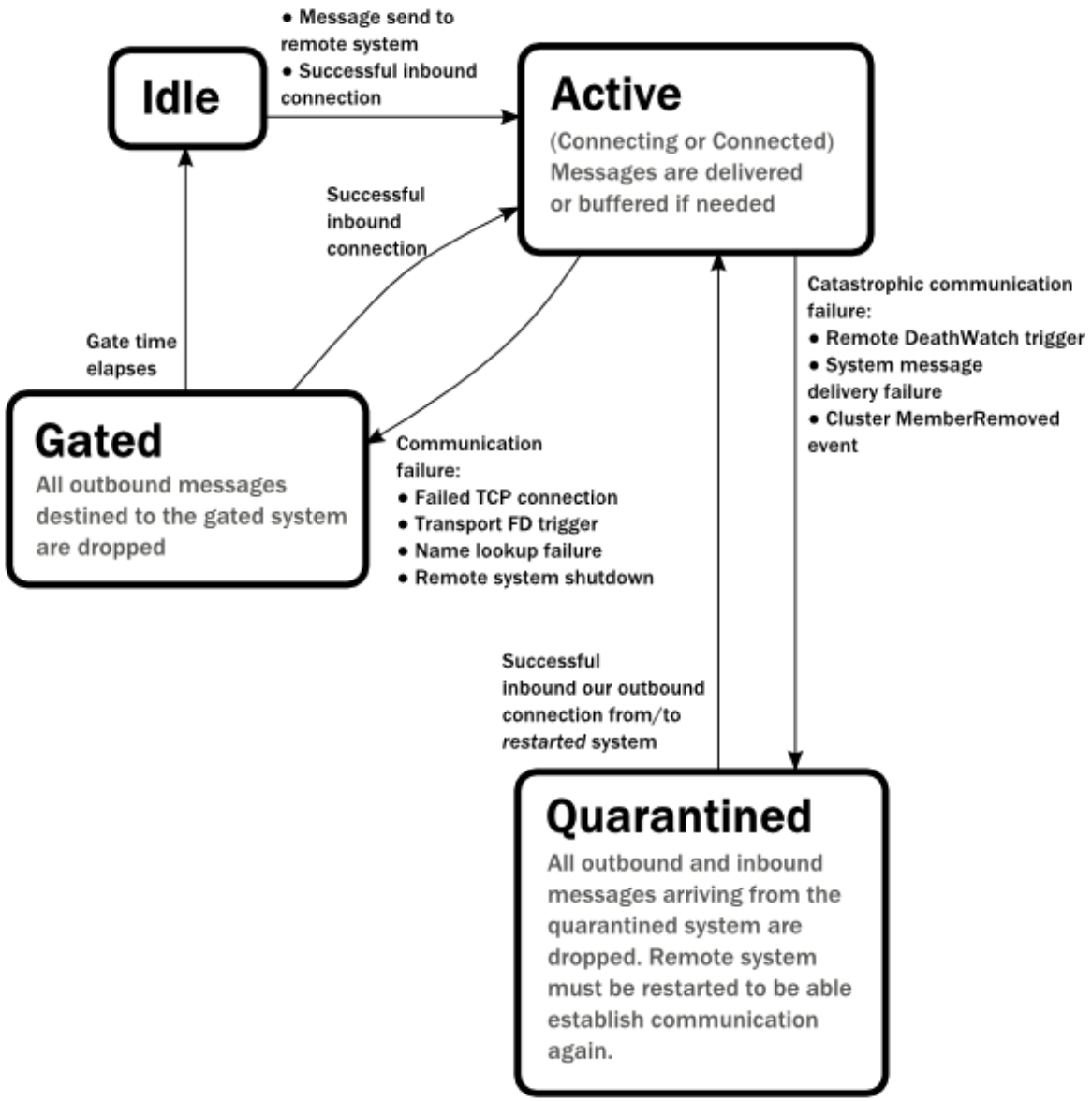
Der Supervisor hat eine Strategie, die dann entscheidet ob

- Einfach weiter gemacht wird
- Der Aktor neu gestartet wird (Zustand wird zurückgesetzt)
- Der Aktor dauerhaft beendet wird
- Der Fehler eskaliert wird (Der Supervisor selbst schlägt dann fehl)

Mit Akka Remoting können Aktoren miteinander übers Netzwerk kommunizieren.

- So lassen sich Aktorensysteme verteilen.
- Relativ enge Kopplung, im Gegensatz zu REST, WebSocket etc.
- Low-level-Protokoll, spezifisch für Akka.
- Alle Aktoren haben einen Pfad, der sich aus der Aktorenhierarche ergibt:
  - Lokal: */user/supervisor/serviceA/$xba*
  - Remote: *akka.tcp://app@127.0.0.1:2552/user/supervisor/serviceA/$xba*

Akka kümmert sich um den Lebenszyklus der Verbindung.



Literatur

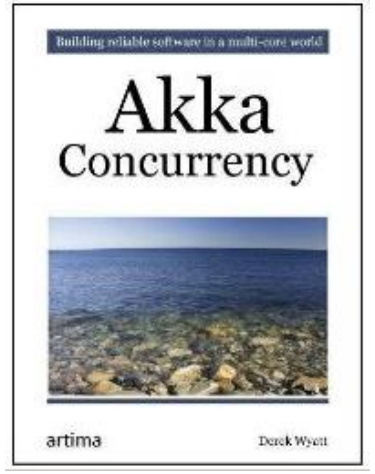
Links & Literatur

Functional Reactive Programming

- https://speakerdeck.com/cmeiklejohn/functional-reactive-programming
- https://speakerdeck.com/mnxfst/reactive-programming-on-example-of-the-basar-platform
- https://speakerdeck.com/peschlowp/reactive-programming

Akka

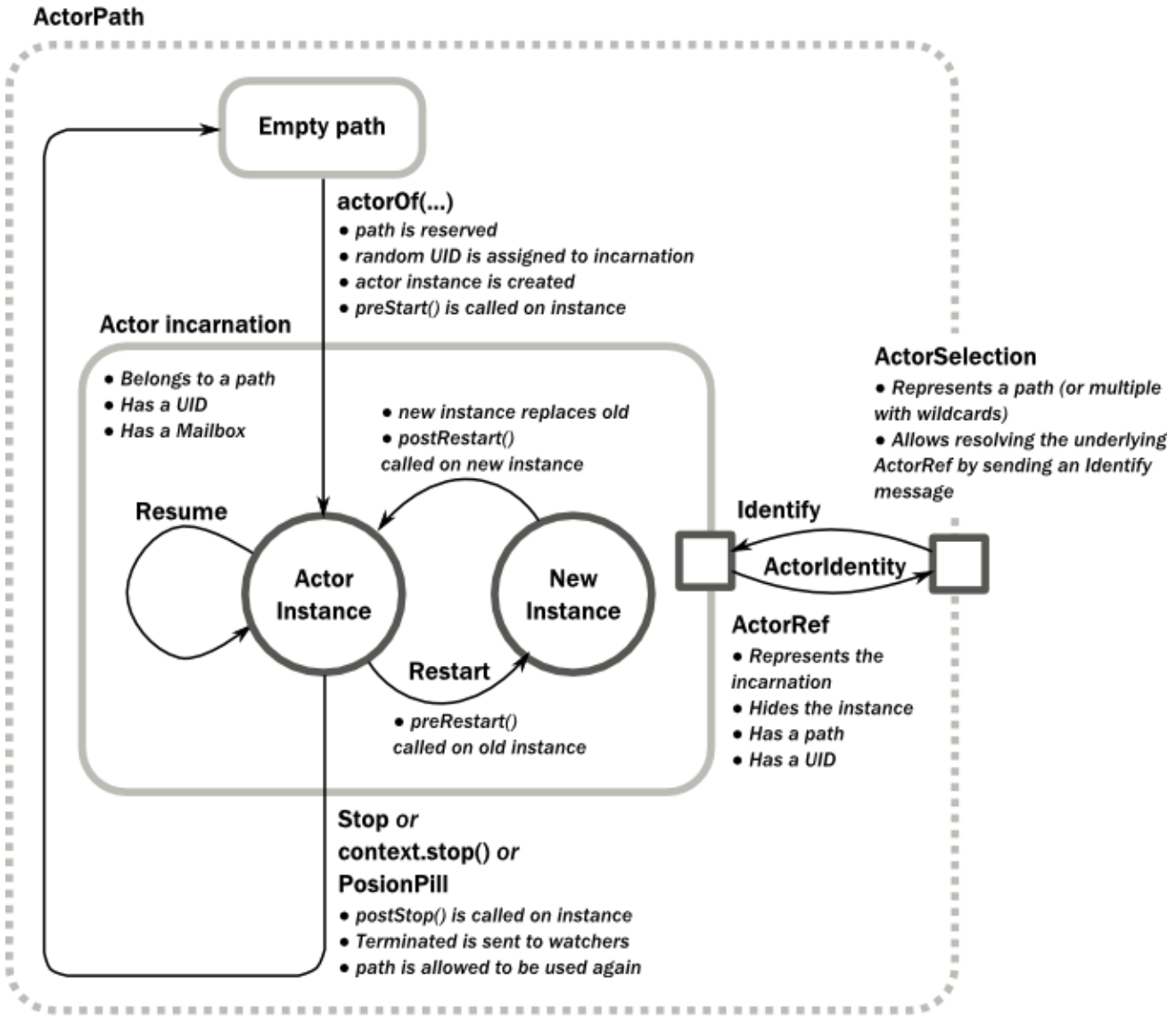
- http://doc.akka.io/docs/akka/2.3.6/intro/getting-started.html
- https://speakerdeck.com/rayroestenburg/akka-in-action
- https://speakerdeck.com/rayroestenburg/akka-in-practice



AKKA Concurrency
Derek Wyatt
Computer Bookshops
(24. Mai 2013)

Bonusmaterial

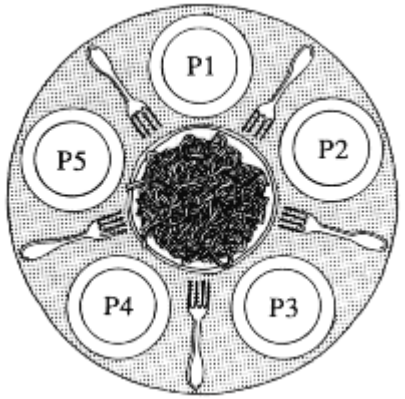
Der Lebenszyklus eines akka Aktors



http://doc.akka.io/docs/akka/2.3.6/java/untyped-actors.html

Beispiel: Das Philosophenproblem

http://www.inf.fu-berlin.de

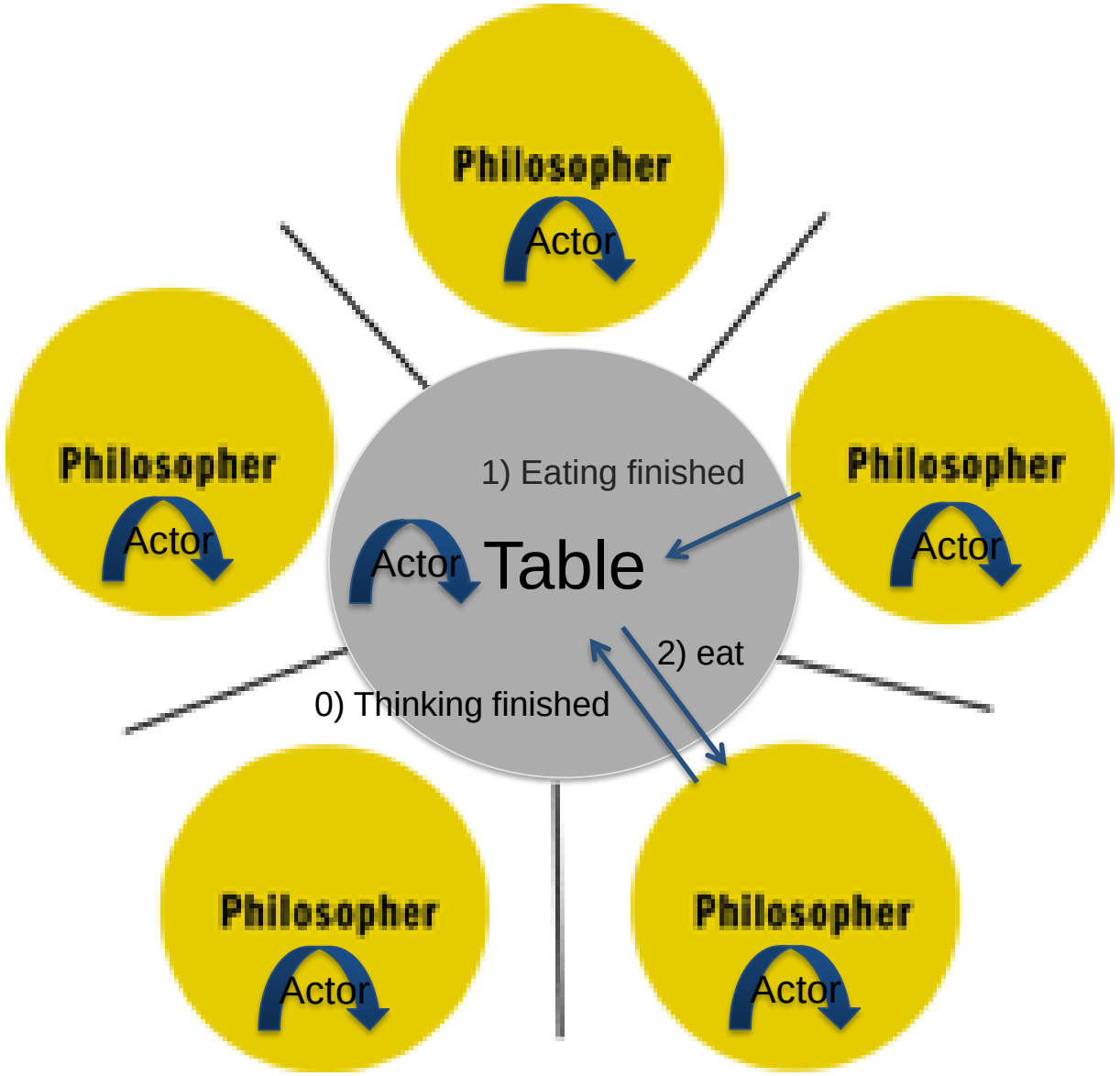


5 Philosophen denken entweder oder essen.
Zum Essen der Spaghetti benötigen jeder 2 Gabeln.
Ein hungriger Philosoph setzt sich, nimmt die rechte Gabel, dann die linke, isst, legt danach die linke Gabel auf den Tisch, dann die rechte.
Jeder benutzt nur die links bzw. rechts von seinem Platz liegende.

Umsetzung mit akka:

- Jeder Philosoph ist ein Actor
- Der Tisch ist ebenfalls ein Actor
- Der Tisch kümmert sich um die konsistente Verteilung der Gabeln wenn Philosophen essen möchten
- Der Philosoph wird benachrichtigt, dass er jetzt essen kann – er benachrichtigt den Tisch wenn er mit dem Essen fertig ist

Beispiel: Das Philosophenproblem



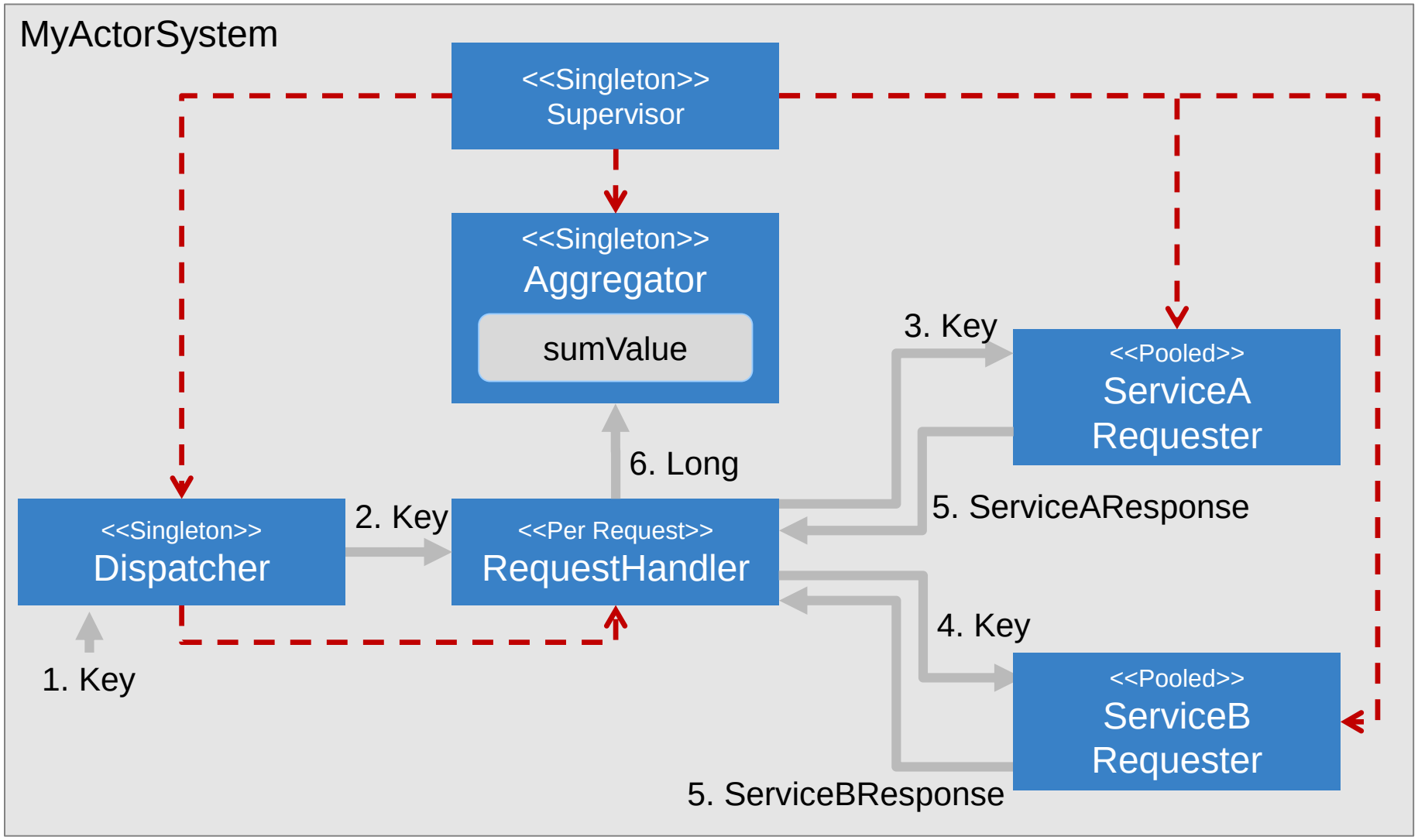
Der Philosoph.

```
public class Philosopher extends UntypedActor implements Serializable {
    @Override
     public void onReceive(Object o) throws Exception {
         Message m = (Message) o;
         if (m.getMessage().equals(DO_EAT)) {
               // eating … finish.
               Message eatFinished = new Message(m.sender, EAT_FINISHED);
               table.tell(eatFinished); // put forks

               // thinking … finish.
               Message thinkFinished = new Message(m.sender, THINK_FINISHED);
               table.tell(thinkFinished); // take forks
         }
     }
```

Der Tisch.

```
public class Table extends UntypedActor {

    private List<ActorRef> philosophers = new ArrayList();
    private List<Fork> allForks = new ArrayList();
    private Set<Fork> freeForks = new HashSet();

    public void onReceive(Object o) {
        Philosopher.Message m = (Philosopher.Message) o;
        Fork leftFork = allForks.get(m.getSender());
        Fork rightFork = allForks.get((m.getSender() + 1) % allForks.size());

        if (m.getMessage().equals(EAT_FINISHED) {
            // put forks
            freeForks.add(leftFork);
            freeForks.add(rightFork);

        } else if (m.getMessage().equals(THINK_FINISHED)) {
            int philosopher = m.getSender();

            // deadlock detection

            // take forks
            freeForks.remove(leftFork);
            freeForks.remove(rightFork);

            // start eat
            philosophers.get(philosopher).tell(new Philosopher.Message(philosopher, DO_EAT));
```

Unser Beispiel-Aktorensystem als Bild



Legende:

<<Kardinalität>>
Aktor

Reihenfolge.
Nachrichten-Typ

Nachrichtenfluss

Bei Bedarf erweitert um Informationen zur Kommunikationsart (lokal, entfernt) und dem Routing.

Parent - - - → Child
Aktorenhierarchie (Supervision)

Best Practices zur Fehlertoleranz.

- Baue Aktoren und Subsysteme so, dass sie leicht wiederaufsetzbar sind.
- Aktoren, die wichtigen Zustand halten, sollten „Gefährliche Aktivitäten“ an Kind-Aktoren delegieren, die dann wenn Notwendig neu gestartet werden können (Error Kernel Pattern).

Akka selbst bietet nur wenige Garantien.

- At most once, d.h. vielleicht auch gar nicht.
- Das ist eine Konsequenz der komplizierten Welt da draußen.
- Klassische Messaging-Frameworks versprechen garantierte Zustellung (zu sehr hohen Kosten), bieten aber auch keine echten Garantien.
- Es gibt Patterns, wie man mit diesen fehlenden Garantien umgehen kann.

API: *UntypedActor*

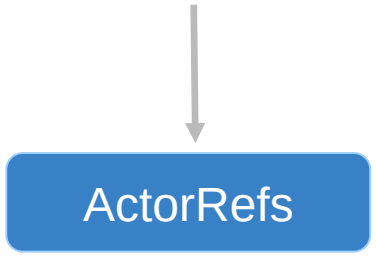
`getContext()` Returns this UntypedActor's UntypedActorContext The UntypedActorContext is not thread safe so do not expose it outside of the UntypedActor.
`getSelf()` Returns the ActorRef for this actor.
`getSender()` The reference sender Actor of the currently processed message.
`onReceive(java.lang.Object message)` To be implemented by concrete UntypedActor, this defines the behavior of the UntypedActor.
`postRestart(java.lang.Throwable reason)` User overridable callback: By default it calls `preStart()`.
`postStop()` User overridable callback.
`preRestart(java.lang.Throwable reason, scala.Option<java.lang.Object> message)` User overridable callback: '"By default it disposes of all children and then calls `postStop()`."' Is called on a crashed Actor right BEFORE it is restarted to allow clean up of resources before Actor is terminated.
`preStart()` User overridable callback.
`unhandled(java.lang.Object message)` Recommended convention is to call this method if the message isn't handled in `onReceive(java.lang.Object)` (e.g.

API: *ActorContext*

Actor suchen

Kind-Aktor erzeugen

`actorFor(ActorPath path)` Look-up an actor by path; if it does not exist, returns a reference to the dead-letter mailbox of the `ActorSystem`.
`actorFor(scala.collection.Iterable<java.lang.String> path)` Look-up an actor by applying the given path elements, starting from the current context, where ".." signifies the parent of an actor.
`actorFor(java.lang.Iterable<java.lang.String> path)` Java API: Look-up an actor by applying the given path elements, starting from the current context, where ".." signifies the parent of an actor.
`actorFor(java.lang.String path)` Look-up an actor by path represented as string.
`actorOf(Props props)` Create new actor as child of this context and give it an automatically generated name (currently similar to base64-encoded integer count, revers
`actorOf(Props props, java.lang.String name)` Create new actor as child of this context with the given name, which must not be null, empty or start with "$".



`child(java.lang.String name)` Get the child with the given name if it exists.
`children()` Returns all supervised children; this method returns a view (i.e.
`dispatcher()` Returns the dispatcher (MessageDispatcher) that is used for this Actor.
`parent()` Returns the supervising parent ActorRef.
`props()` Retrieve the Props which were used to create this actor.
`receiveTimeout()` Gets the current receive timeout.
`self()`
`sender()` Returns the sender 'ActorRef' of the current message.
`setReceiveTimeout(scala.concurrent.duration.Duration timeout)` Defines the inactivity timeout after which the sending of a `ReceiveTimeout` message is triggered.
`system()` The system that the actor belongs to.
`unbecome()` Reverts the Actor behavior to the previous one on the behavior stack.
`unwatch(ActorRef subject)` Unregisters this actor as Monitor for the provided ActorRef.
`watch(ActorRef subject)` Registers this actor as a Monitor for the provided ActorRef.
`writeObject(java.io.ObjectOutputStream o)` ActorContexts shouldn't be Serializable

API: *ActorRef*

`forward(java.lang.Object message, ActorContext context)` Forwards the message and passes the original sender actor as the sender.
`isTerminated()` Is the actor shut down? The contract is that if this method returns true, then it will never be false again.
`noSender()` Use this value as an argument to `tell(java.lang.Object, akka.actor.ActorRef)` if there is not actor to reply to
`path()` Returns the path for this actor (from this actor up to the root actor).
`tell(java.lang.Object msg, ActorRef sender)` Sends the specified message to the sender, i.e.