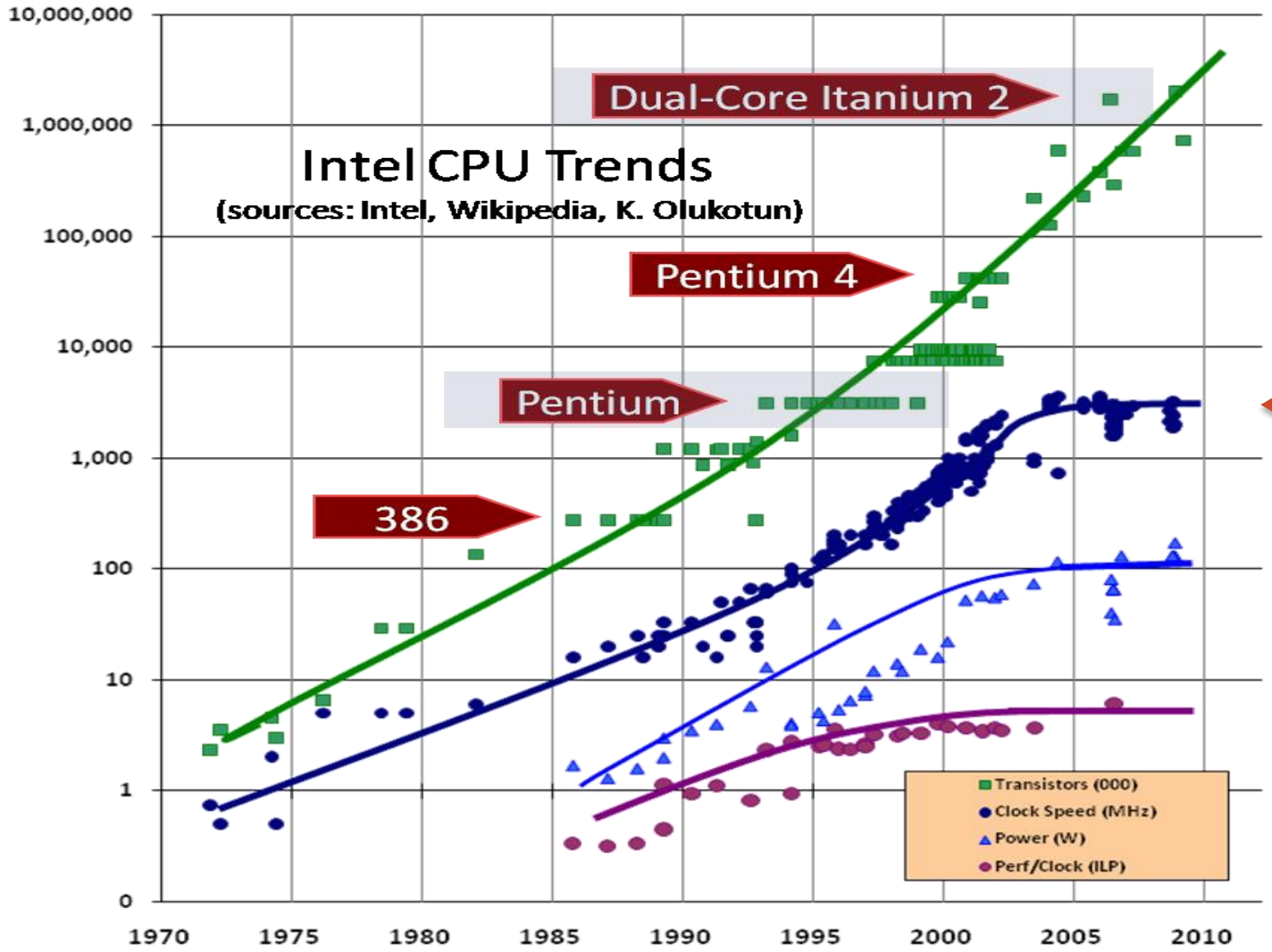
Cloud Computing

Kapitel 2: Programmiermodelle für die Cloud (Reactive Programming)

Dr. Josef Adersberger, Dr. Alexander Krauss

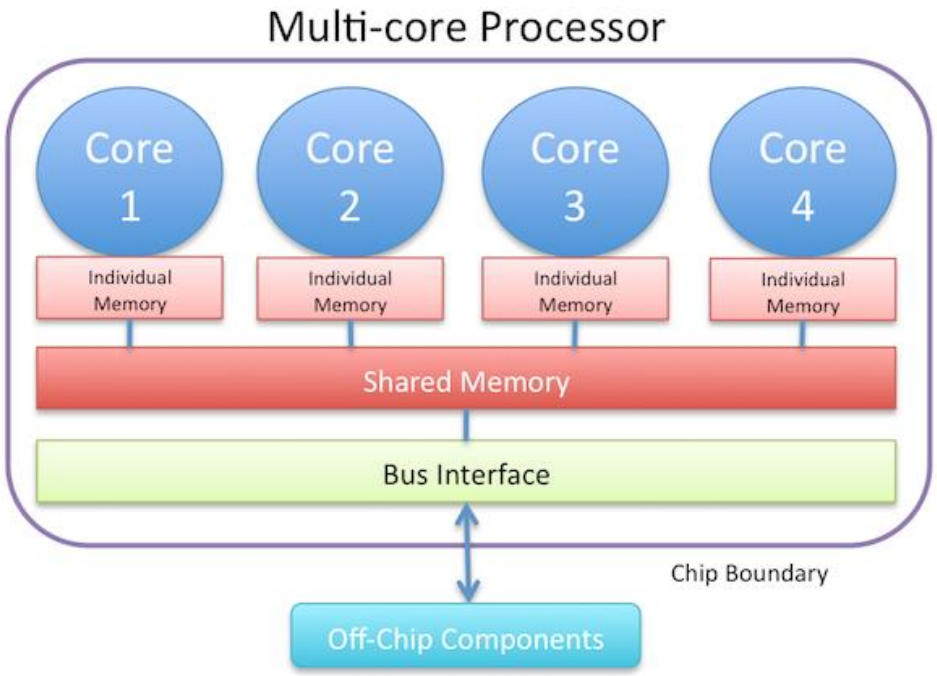
„The free lunch is over“: Es gibt keine kostenlose Performanzsteigerung mehr – Nebenläufigkeit zählt.



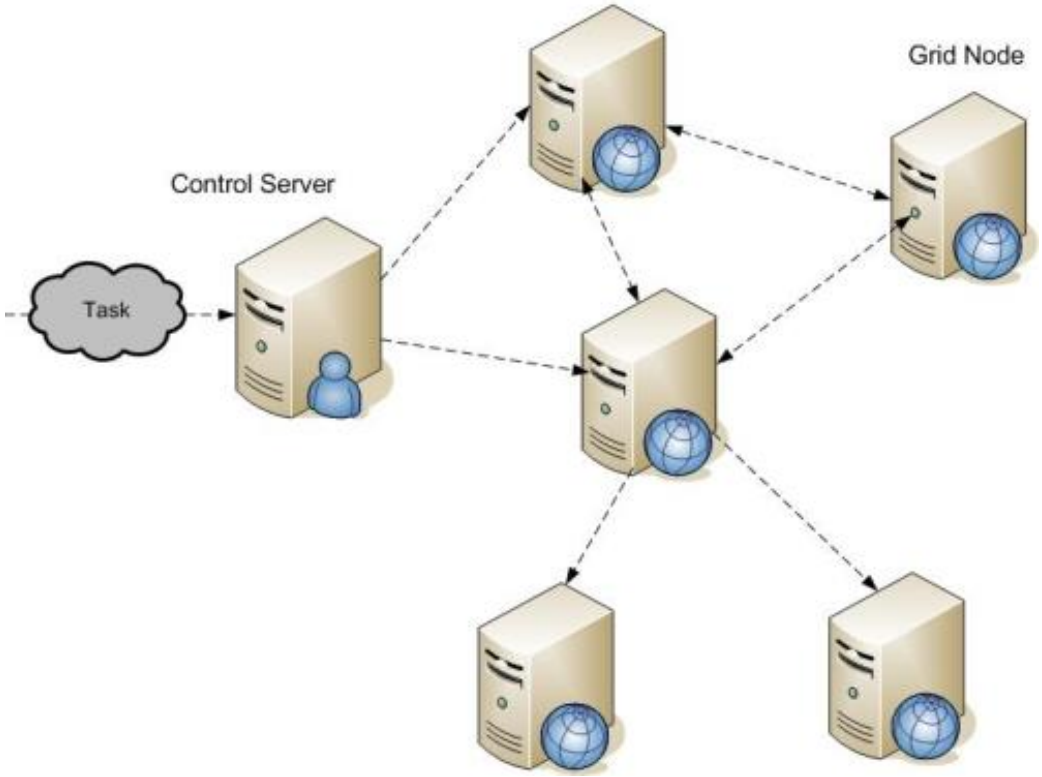
Anzahl Transistoren
Moore‘s Law gilt weiterhin

Taktfrequenz
Seit 2004 ist die Taktfrequenz von CPUs konstant

Nebenläufigkeit kann im Kleinen und im Großen betrieben werden.

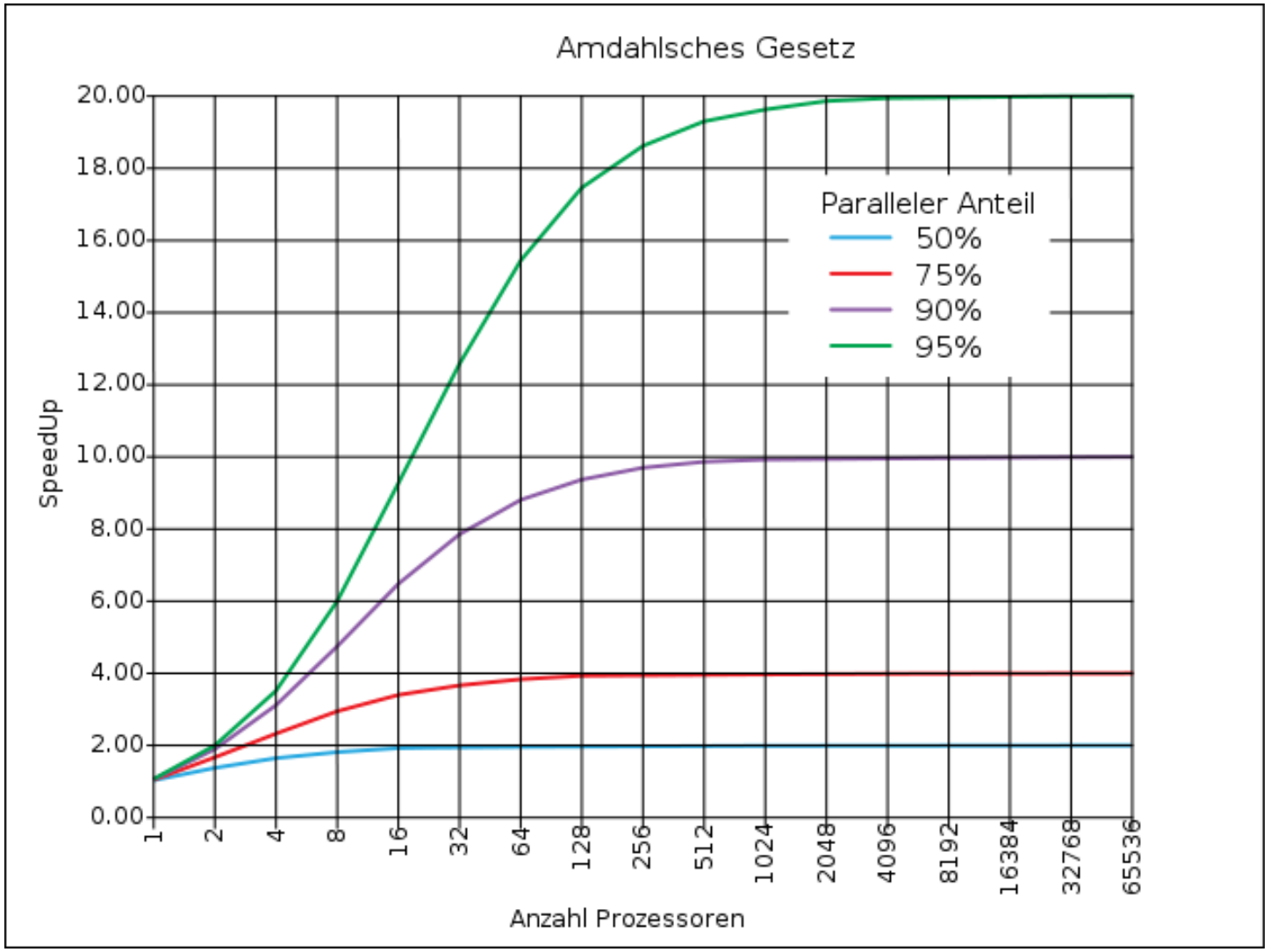


Multi Core



Multi Node
(Cluster, Grid, Cloud)

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



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

$$Speedup = \frac{1}{1-P}$$ für N = ∞

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

Planspiel: Paralleles Rechnen

Wir wollen gemeinsam folgenden Algorithmus ausführen

- Eingabe
  - Ein kontinuierlicher Strom von Werten x
- Ausgabe

$$\sum_{x} f(x) * g(x)$$

Für jeden eingehenden Request x
- Berechne f(x) durch externen REST-Service A
- Berechne g(x) durch externen REST-Service B
- Multipliziere die Ergebnisse
- Addiere das Produkt auf eine Variable a

Dazu machen wir zwei Teams

Vs.

- Konventionell
  - Jeder Teilnehmer ist ein Thread, nimmt einen Request entgegen, und arbeitet ihn bis zum Ende ab:
    - Services aufrufen
    - Berechnung durchführen
    - Zustand aktualisieren
  - Die Zustandsvariable a ist durch ein Lock gesichert:

    Bitte immer wieder ordentlich zusperren!

- Reactive
  - Es gibt folgende Aktoren
    - Request Handler
      - Request verwalten
      - Multiplizieren
      - Repräsentiert als passive Nummerierte Mailbox.
    - Service-Clients
      - Services abfragen
    - Aggregator
      - Zustandsvariable verwalten
      - Eingehende Nachrichten darauf addieren

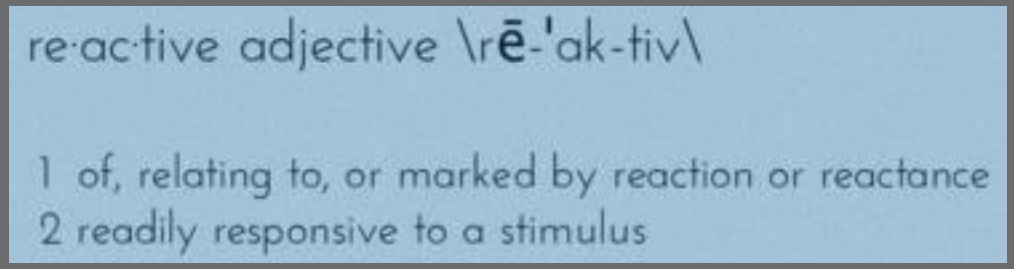
Ablauf

- Die übrigen Teilnehmen sind Beobachter. Bitte auf folgende Fragen achten:
  1. Wo sind die Engpässe im jeweiligen Modell?
  2. Seht ihr offensichtliche Optimierungsmöglichkeiten?
  3. Welche Aspekte des Planspiels weichen von der Realität ab?

- Wir proben beide Varianten hintereinander, je ca. 3-4 Minuten.
  - Konventionell
  - Reactive

Grundbegriffe

Das Programmiermodell der Cloud: Functional Reactive Programming



Das Reactive Manifesto

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

http://www.reactivemanifesto.org

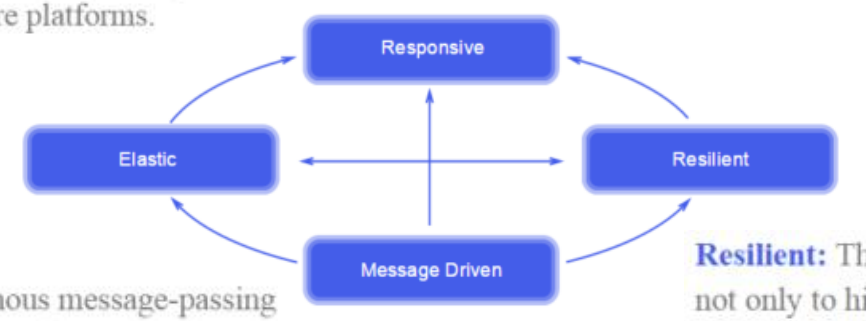
Das Reactive Manifesto

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

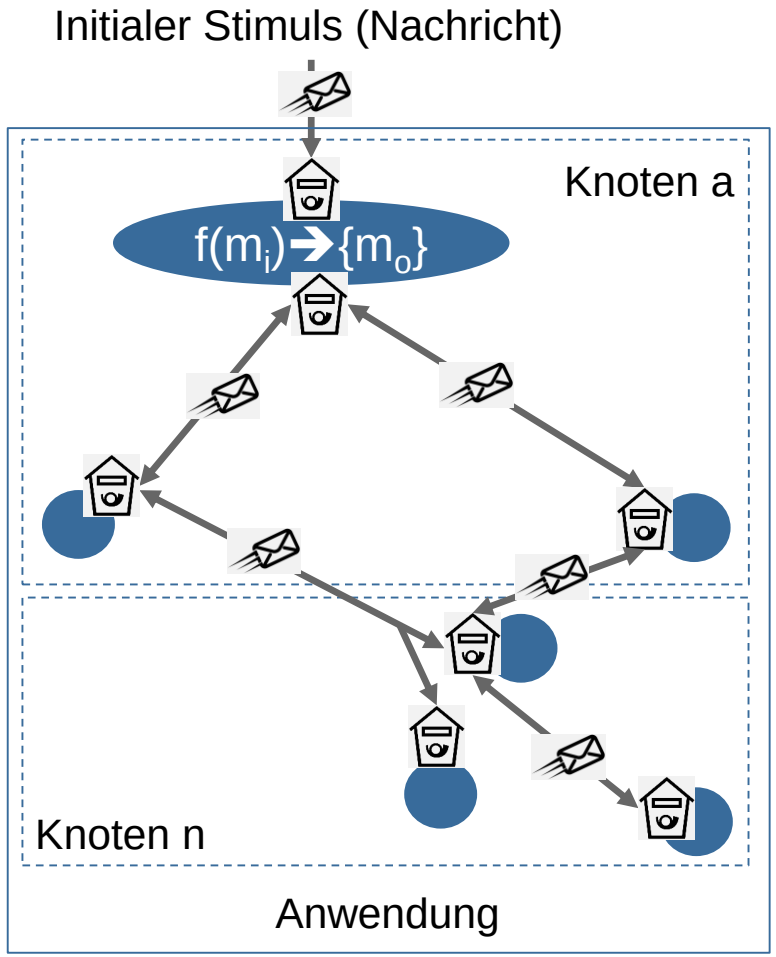
Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, 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.

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.

React to failures

Functional Reactive Programming (FRP): 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 im Regelfall 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 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** möglich und damit höherer möglicher Speedup.
  Für den Fall, dass die Logik und die Datenmenge sich entsprechend partitionieren lässt.

Functional Reactive Programming am Beispiel *akka*

Grenzen des klassischen Thread-Modells

- Im klassischen JEE-Modell bearbeitet ein Thread eine Anfrage von Anfang bis Ende.
  - Viel warten (auf IO).
  - Viel Synchronisation (bei gemeinsamen Zustand).
  - Parallelisierung und Bulk-Verarbeitung schwierig.

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

akka: Die Open-Source-Eigenschaften

https://www.openhub.net/p/akka

In a Nutshell, akka...

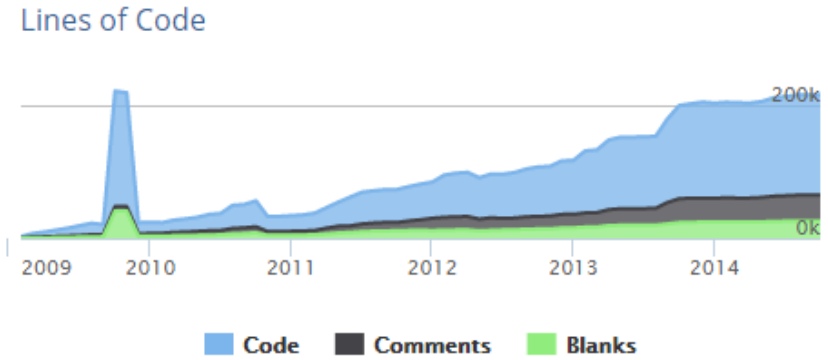
... has had 14,247 commits made by 196 contributors representing 152,147 lines of code

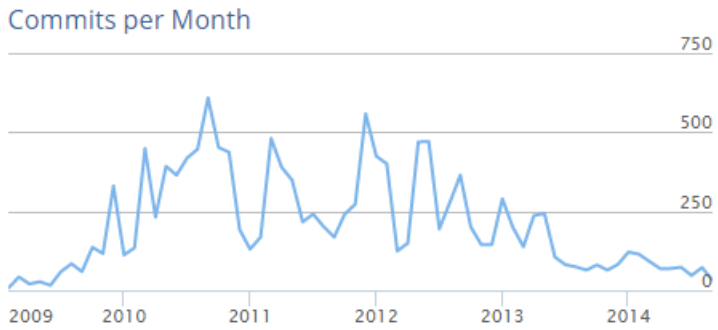
... is mostly written in Scala with an average number of source code comments

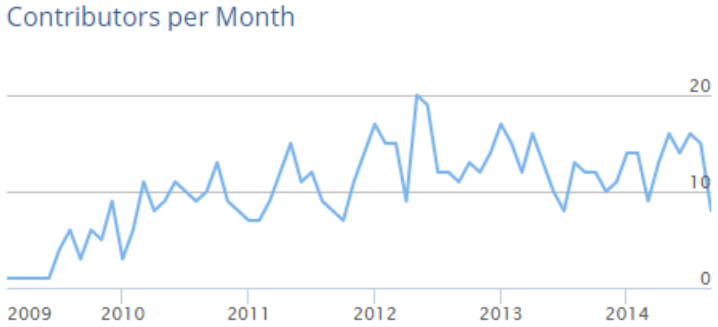
... has a well established, mature codebase maintained by a very large development team with decreasing Y-O-Y commits

... took an estimated 39 years of effort (COCOMO model) starting with its first commit in February, 2009 ending with its most recent commit 6 days ago

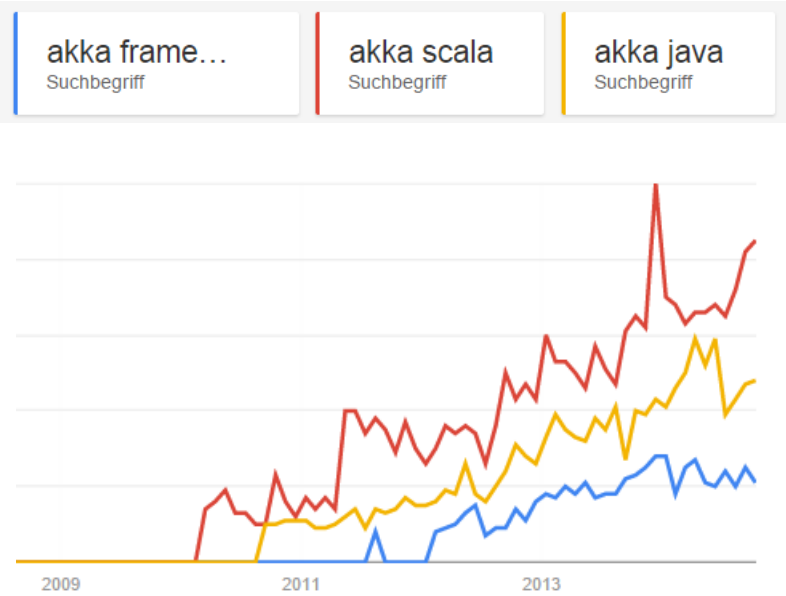
Licenses: Apache-2.0





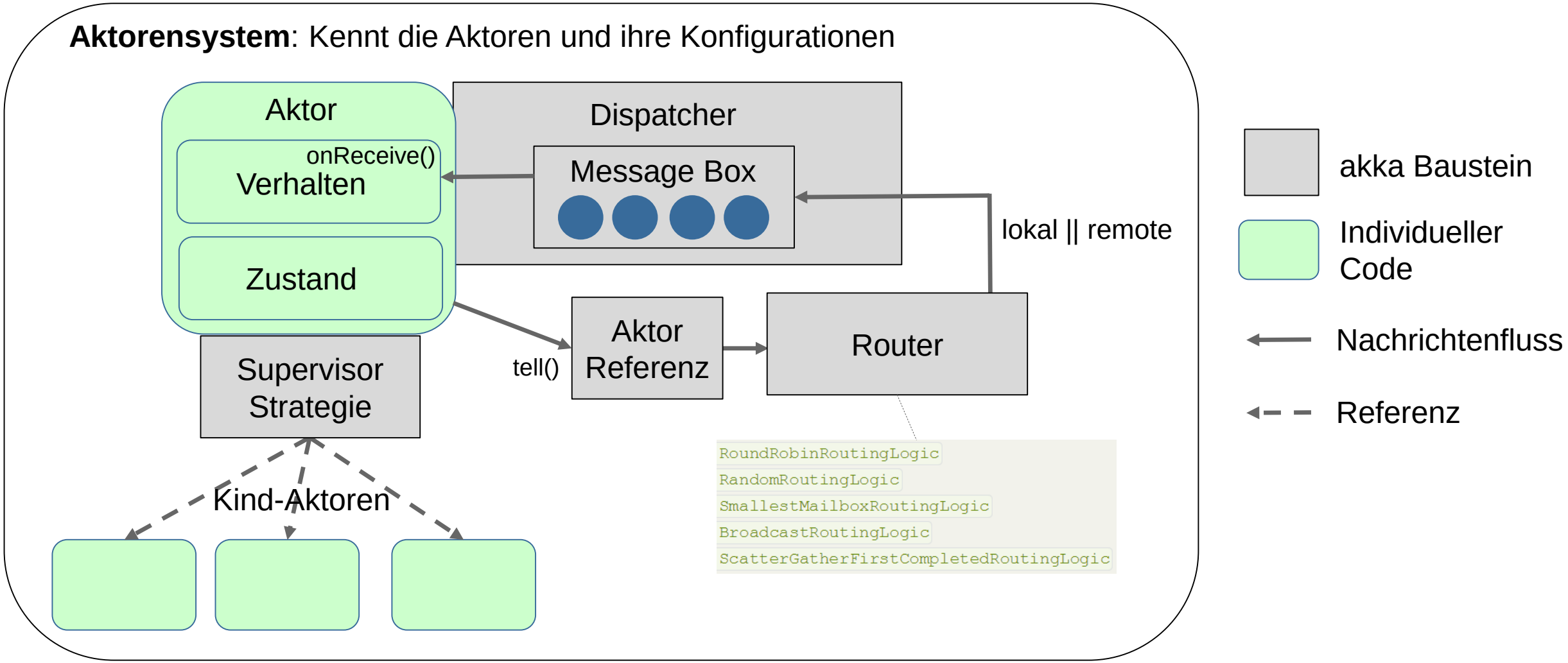


https://www.google.de/trends

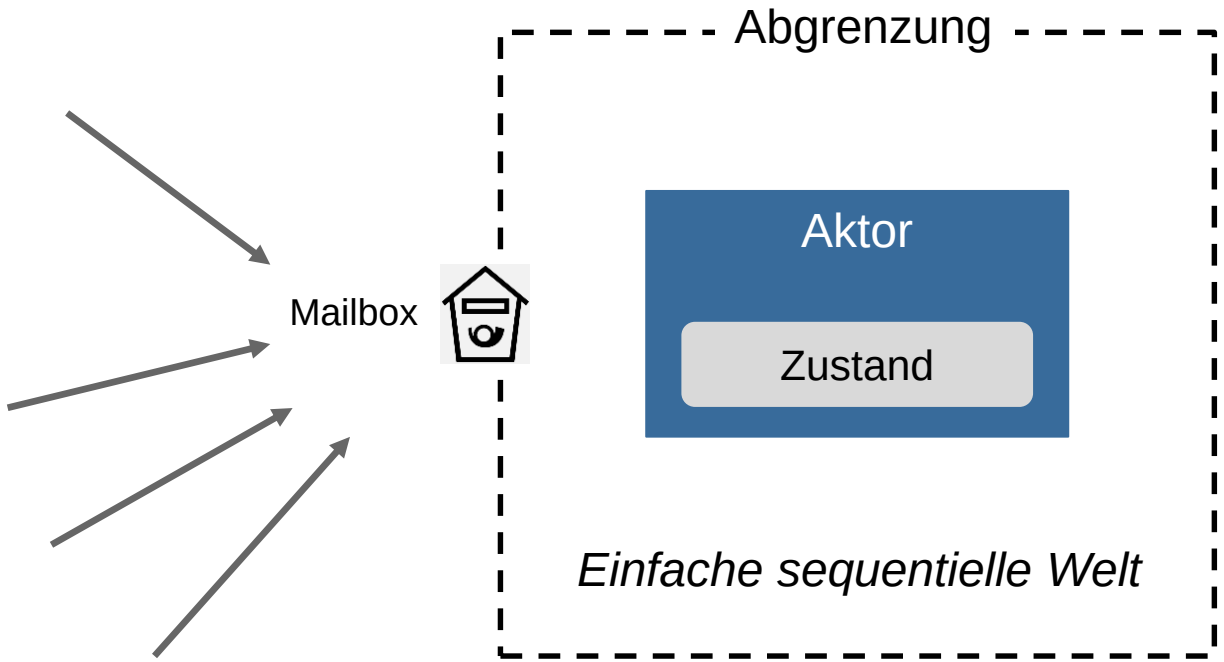


Die Grundkonzepte von akka

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

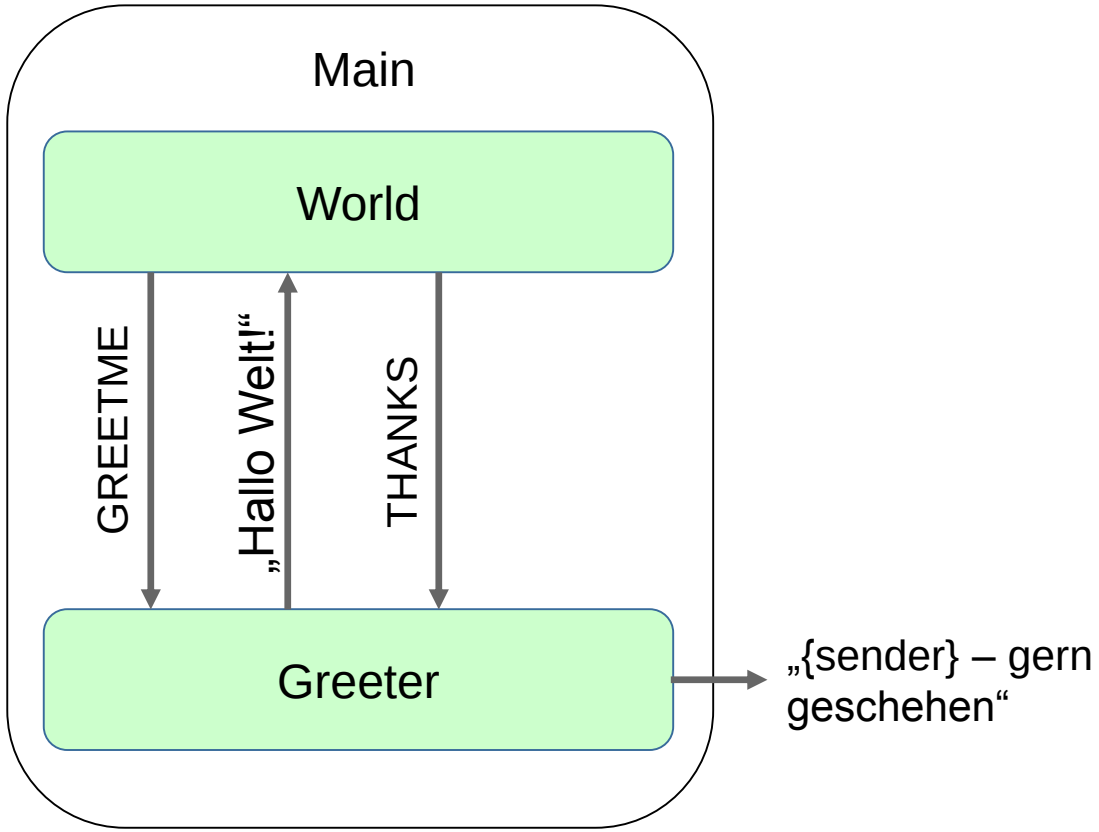
Im Gegensatz zu Threads sind Aktoren leichtgewichtig.

- Eine Million Aktoren? Kein Problem!
- Eine Million Threads? Probiert‘s mal aus!
- Warum?
  - Threads mappen direkt auf Ressourcen im System, und die sind begrenzt.
  - Aktoren haben diese Einschränkung nicht und haben nur einen unwesentlichen Overhead

Parallelisierung findet durch geeignete Komposition von Aktoren statt.

- Worker Pool
  - Akka bietet Konstrukte zur Konstruktion von Pools, mit verschiedenen Dispatch-Strategien
- Parallelkomposition von verschiedenen Abläufen
  - Einfach verschiedene Nachrichten rausschicken
- …

Beispiel: Hello World



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 *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);
        }
    }

}
```

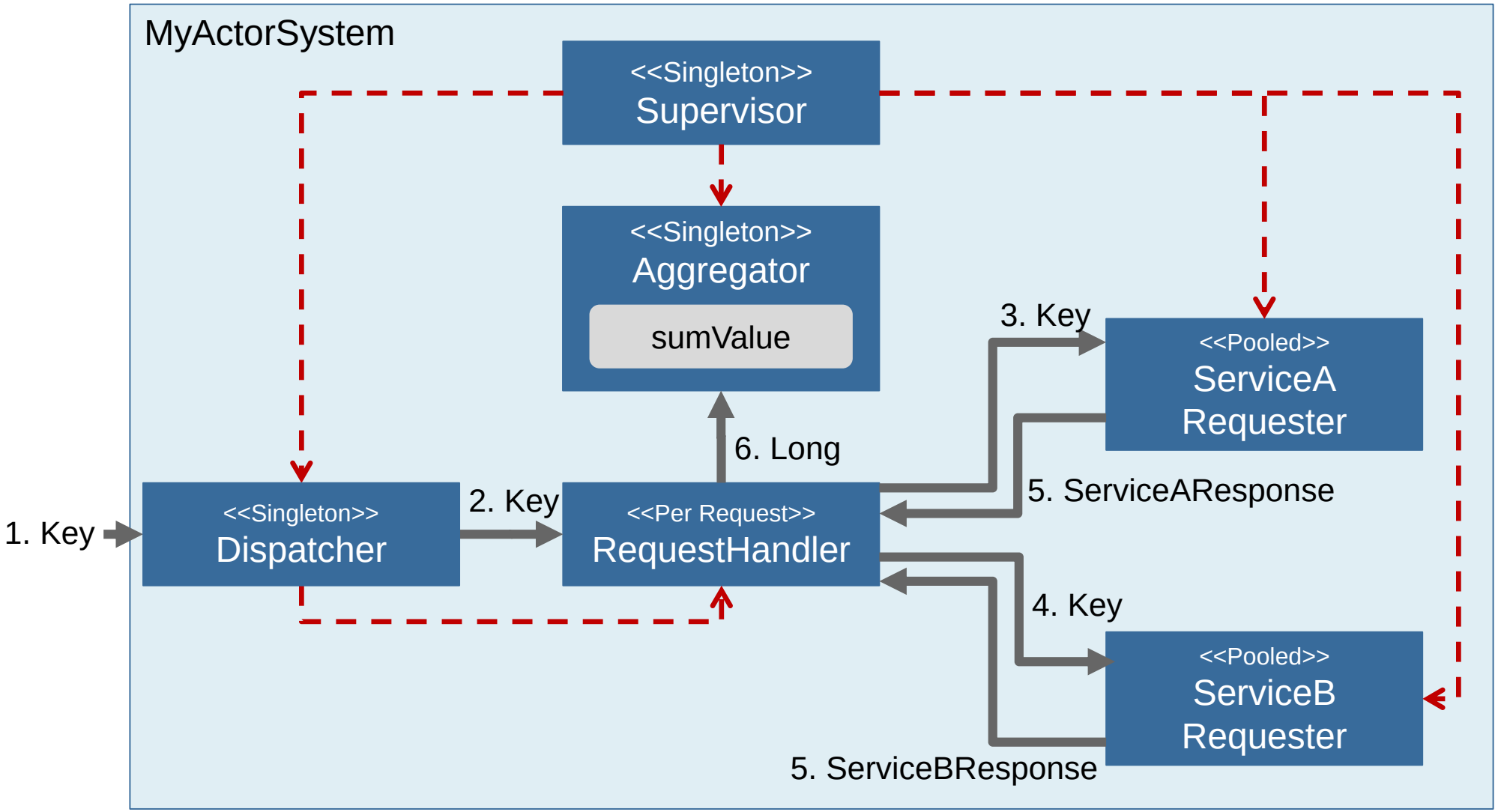
akka Hello World: Die Klasse *Main*

```
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. Do not close over actor state!
  - Eine Nachricht darf syntaktisch ein beliebiges Objekt sein. Verknüpfe sie aber nicht mit dem Zustand des Aktors. Sonst bist du zurück in der Hölle der unkontrollierten Parallelität (d.h., „lasset alle Hoffnung fahren“).
- 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.

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)

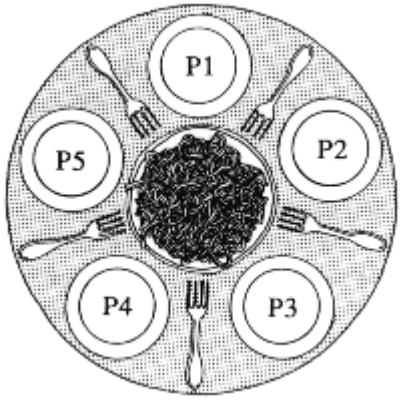
Ein ähnliches Beispiel: Content-Aggregation

- Der Benutzer gibt einen Suchbegriff ein.
- Mehrere Feeds parallel abfragen
- Ergebnisse vereinigen.

→ Tafel

Zweites Beispiel: Das Philosophenproblem

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

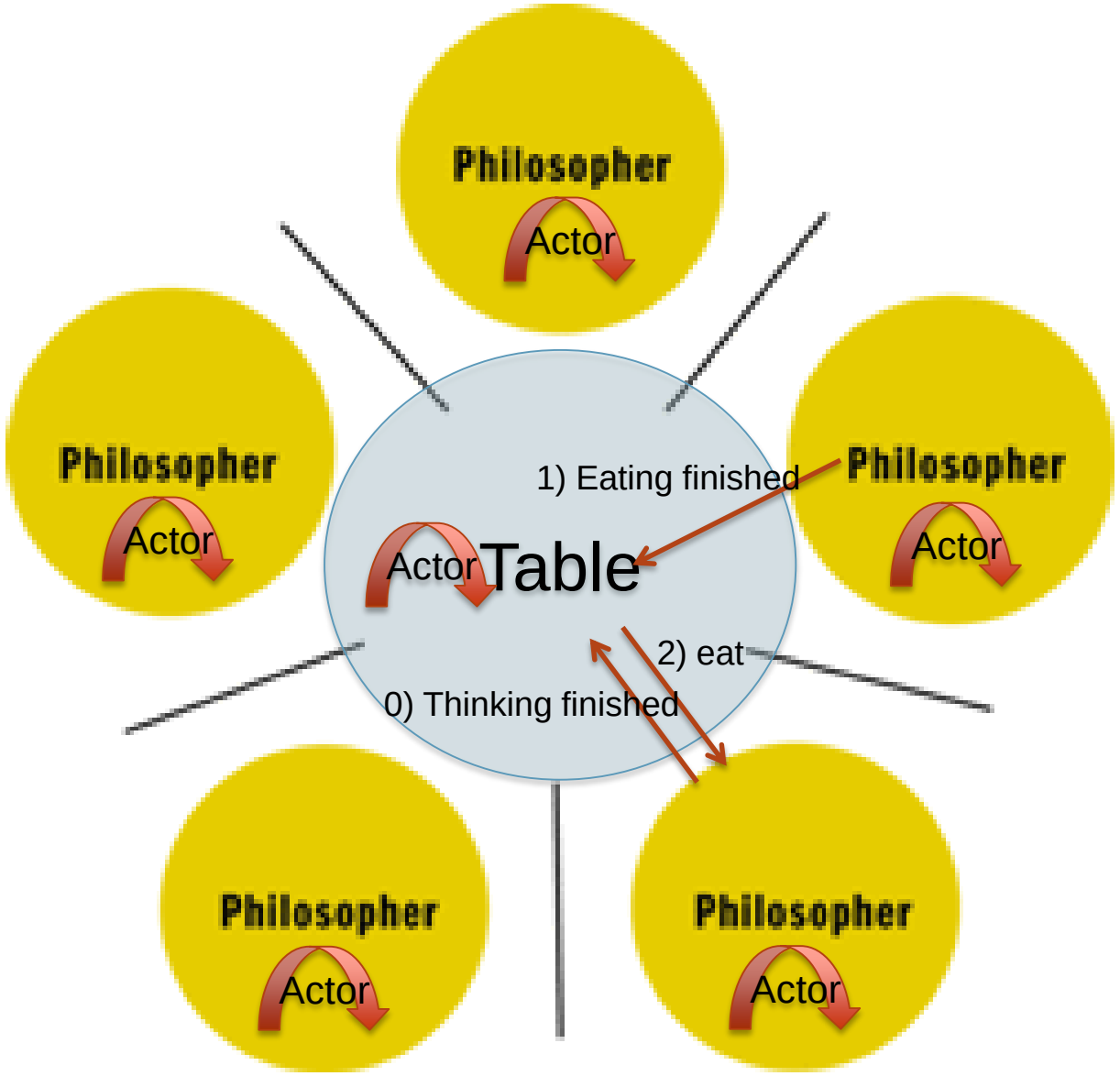


5 Philosophen denken entweder oder essen.
Zum Essen der Spaghetti benötigen jeder 2 Gabeln.
Ein hungriger Philosoph setzt sich, nimmt die rechte Gabel, dann die linke, isst, legt danach die linke Gabel auf den Tisch, dann die rechte.
Jeder benutzt nur die links bzw. rechts von seinem Platz liegende.

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

akka 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
            if (!freeForks.contains(leftFork) || !freeForks.contains(rightFork)){
                super.getSelf().tell(m); // reschedule
                return;
            }

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

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

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)

Exception Handler können oft nicht adäquat auf Fehler reagieren.

```
try {
    callBackend();
} catch (BadRequestException e) {
    throw new IllegalArgumentException("...", e);
} catch (ConnectException e) {
    // Give up (easy) or retry (really hard)
} catch (TimeoutException e) {
    // Give up (easy) or retry (really hard)
} catch (Exception e) {
    // ??? give up...
    throw new IllegalStateException(e);
}
```

- Das Problem lässt sich abmildern durch
  - Klar definierte Fehlerfassade und Fehlermodell
  - Sauber dokumentierte Fehler an Schnittstellen
  - Kanal- und Türsteher-Modell (Siehe gestern)

- Exception Handling kann aber kaum mit korrupten Zuständen umgehen, z.B.
  - Request A setzt x auf einen ungültigen Wert.
  - Request B schlägt daraufhin fehl.
    - Fault: Auslösender Programmierfehler
    - Error: Ungültiger Zustand (x)
    - Failure: Ausfall bei Request B

Der normale Weg mit korrupten Zuständen umzugehen ist im Großen gesehen sehr erfolgreich.

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)

Best Practices zur Fehlertoleranz

- Baue Aktoren und Subsysteme möglichst 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

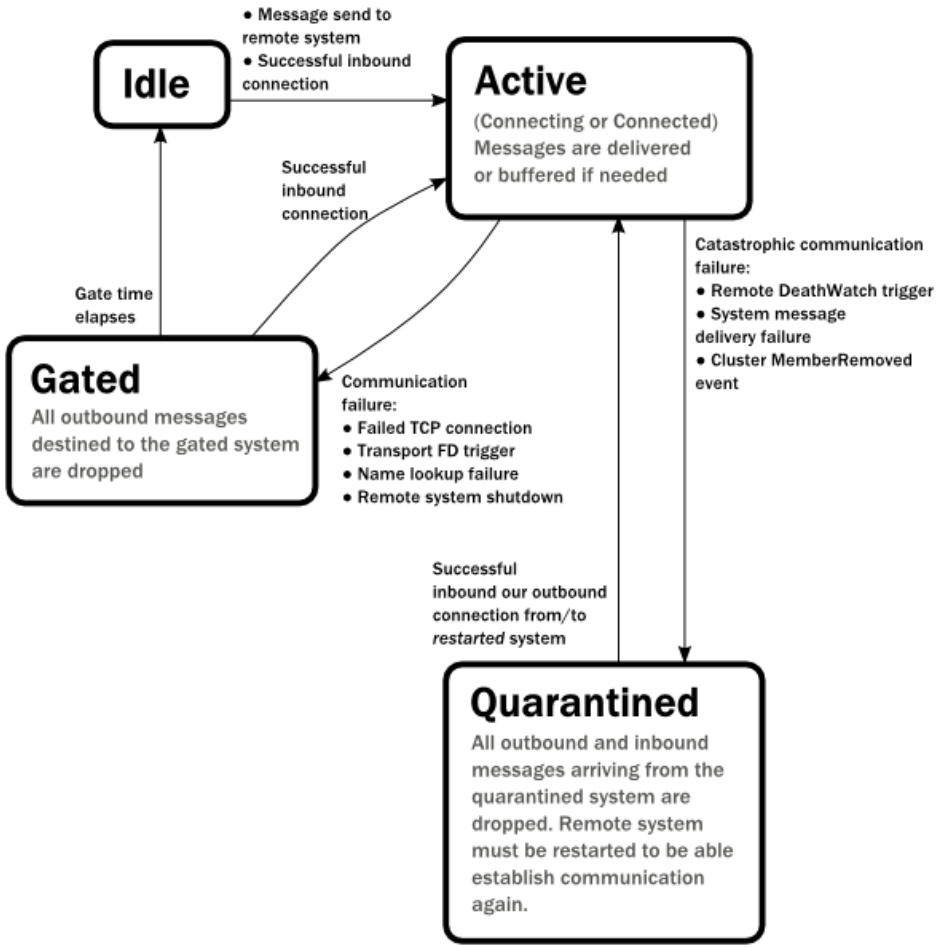
- At most once
  - D.h. vielleicht auch gar nicht.

- 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.

Mit Akka Remoting können Aktoren miteinander übers Netzwerk kommunizieren.

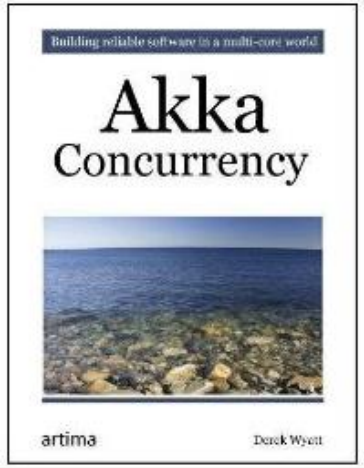
- So lassen sich Aktorensysteme verteilen.
- Relativ enge Kopplung, im Gegensatz zu REST, WebSocket etc.
- Low-level-Protokoll, spezifisch für Akka.
- Alle Aktoren haben einen Pfad, der sich aus der Aktorenhierarche ergibt:
  - Lokal: `/user/supervisor/serviceA/$xba`
  - Remote: `akka.tcp://app@127.0.0.1:2552/user/supervisor/serviceA/$xba`

Akka kümmert sich um den Lebenszyklus der Verbindung. Man bekommt mit, wenn die Verbindung nicht mehr besteht.



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)

API: *UntypedActor*

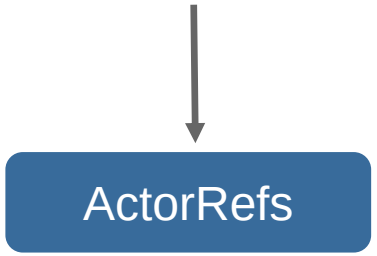
`getContext()` Returns this UntypedActor's UntypedActorContext The UntypedActorContext is not thread safe so do not expose it outside of the UntypedActor.
`getSelf()` Returns the ActorRef for this actor.
`getSender()` The reference sender Actor of the currently processed message.
`onReceive(java.lang.Object 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.