

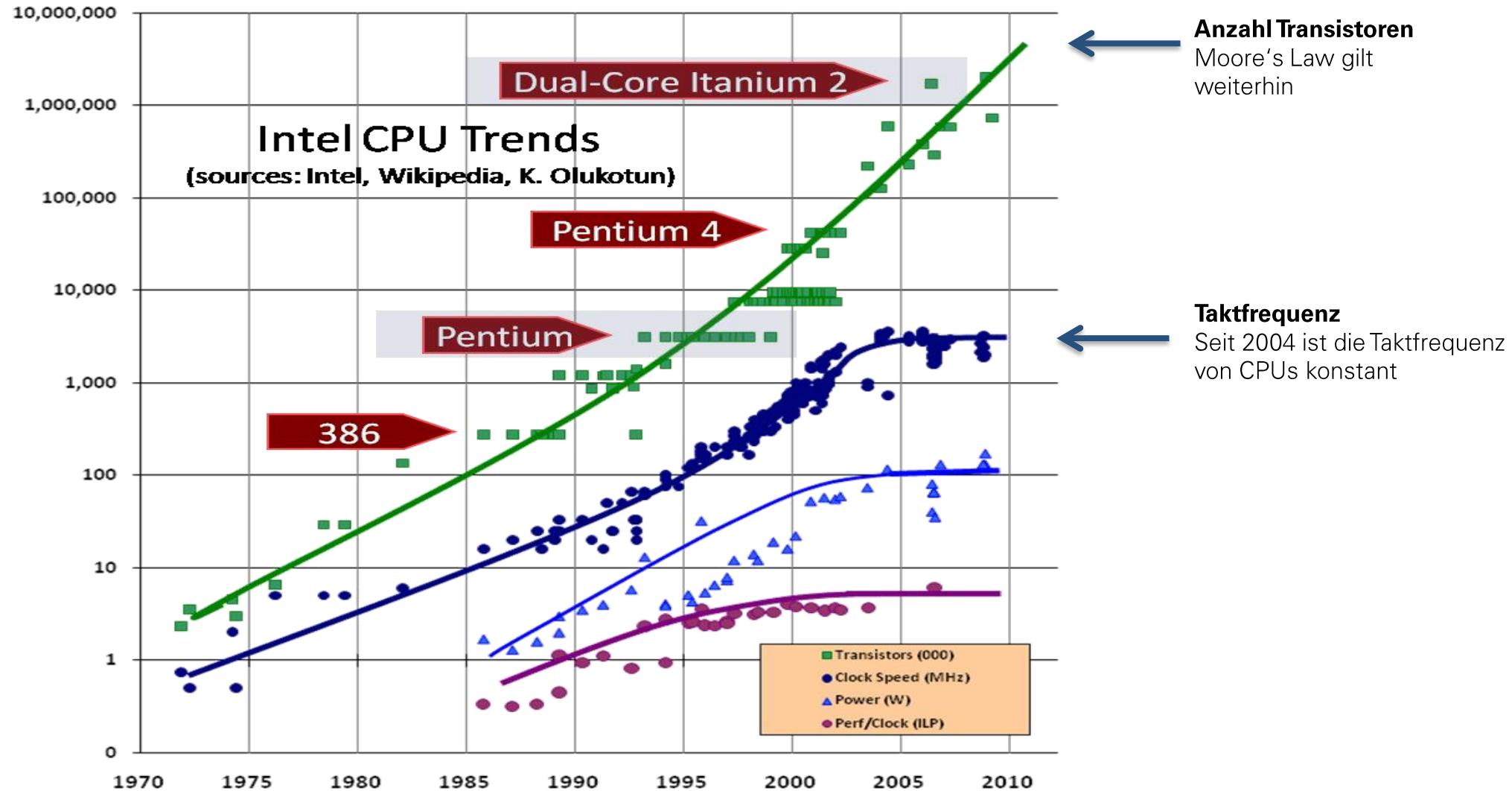
Kapitel 2: Programmiermodelle



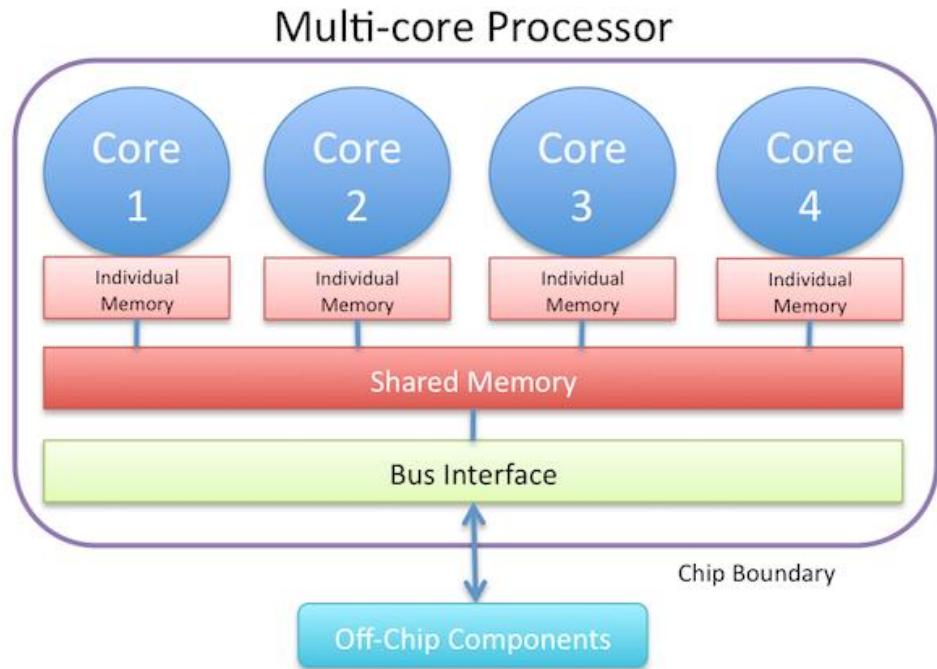
vorlesung
**CLOUD
COMPUTING**

Dieses Kapitel ist nicht Teil
der Vorlesung im WS
2019/20 in Rosenheim.

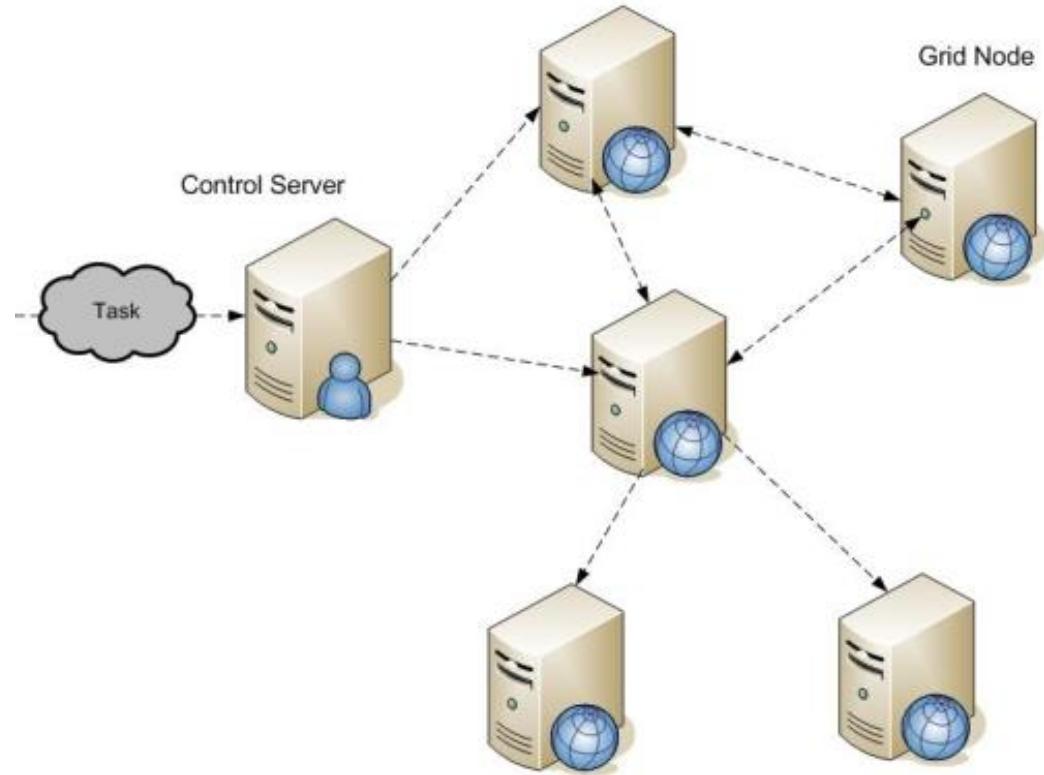
„The free lunch is over“: Es gibt keine kostenlose Performancesteigerung mehr – Nebenläufigkeit zählt



Nebenläufigkeit kann im Kleinen und im Großen betrieben werden

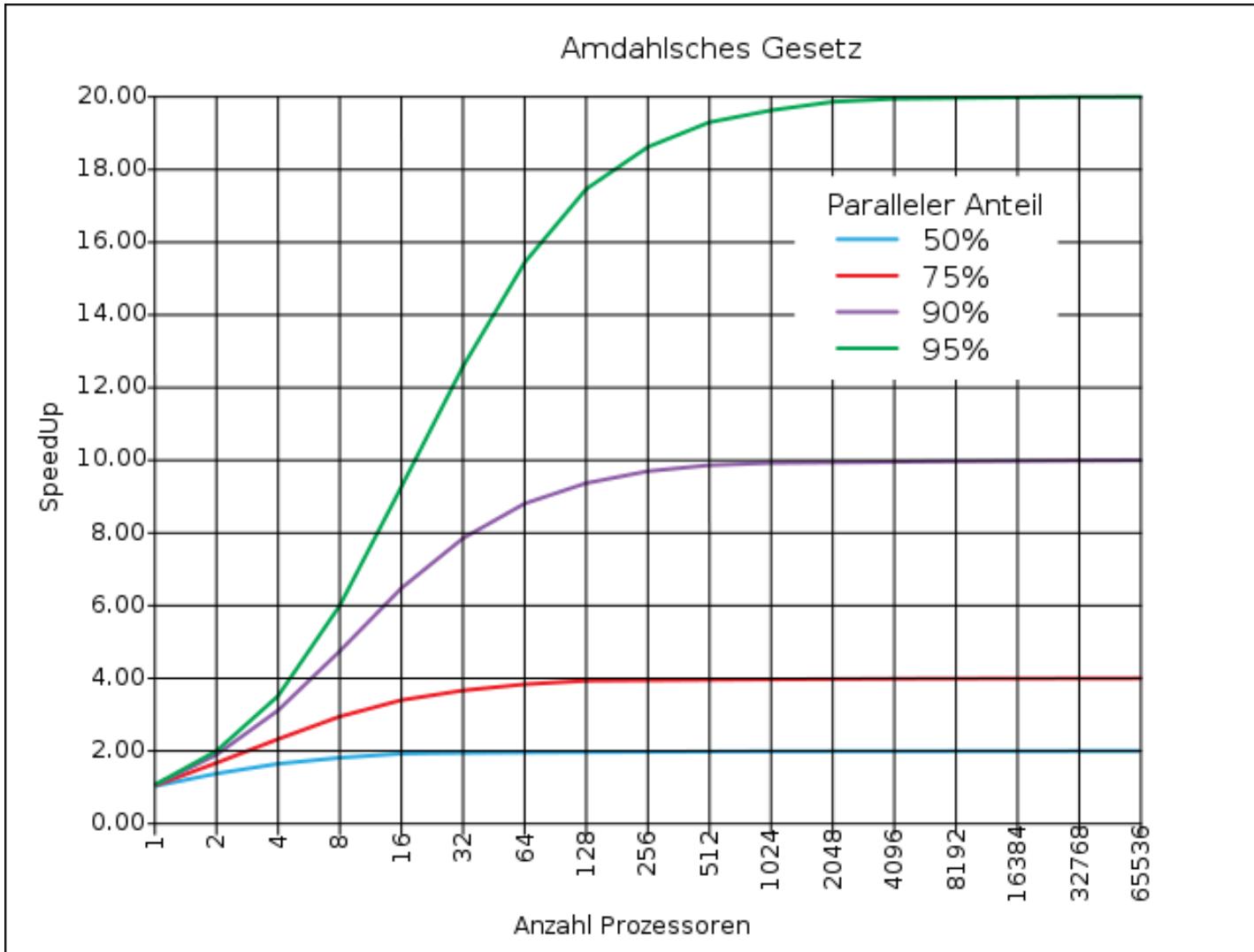


Multi Core



Multi Node
(Cluster, Grid, Cloud)

Das Amdahlsche Gesetz: Die Grenzen der Performance-Steigerung über Nebenläufigkeit



P = Paralleler Anteil

S = Sequenzieller Anteil

N = Anzahl der Prozessoren

Speedup = Maximale Beschleunigung

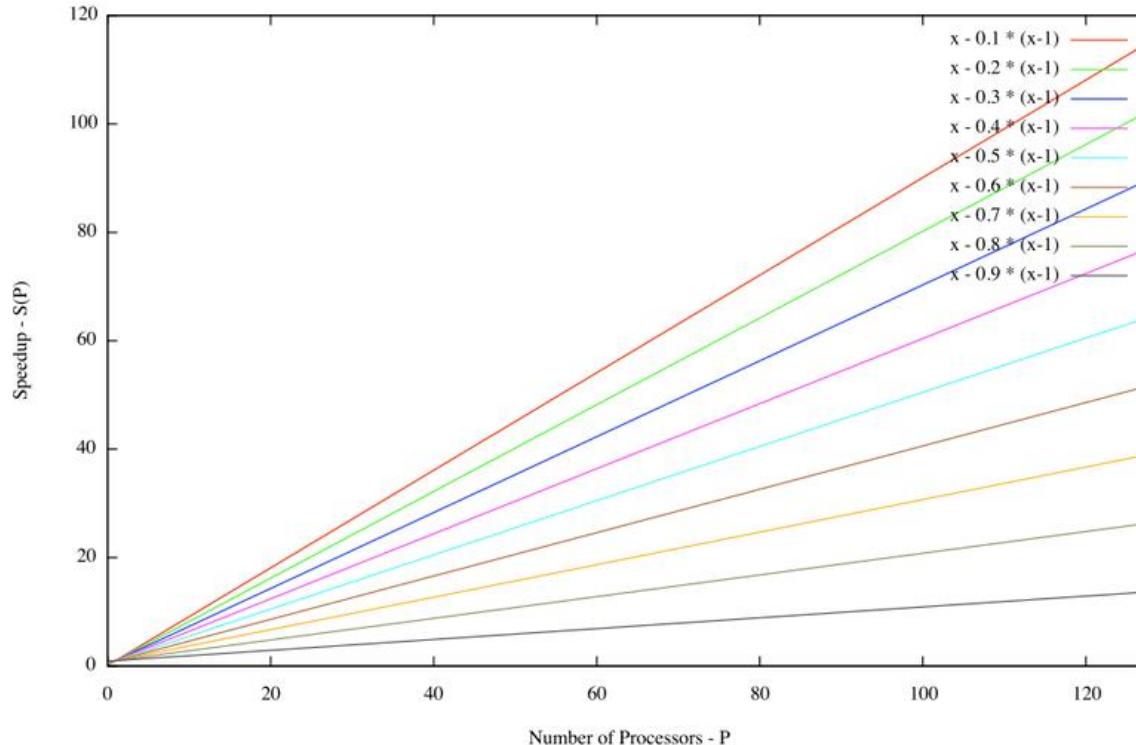
$$\text{Speedup} = \frac{1}{1-P} \quad \text{für } N = \infty$$

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

Das Amdahlsche Gesetz: Die Grenzen der Performance-Steigerung über Nebenläufigkeit

Angenommen, ein Programm benötigt 20 Stunden auf einem Rechner mit einer CPU, und eine Stunde davon wird sequentiell ausgeführt (beispielsweise Initialisierungs-Routinen oder Speicher-Allokation). Die verbleibenden 19 Stunden machen 95 % des Gesamtaufwandes aus und können auf beliebig viele Prozessoren verteilt werden. Die Gesamtrechenzeit kann aber selbst mit unendlich vielen Prozessoren nicht unter 1 Stunde fallen, die maximale Beschleunigung (Speedup) ist also Faktor 20.

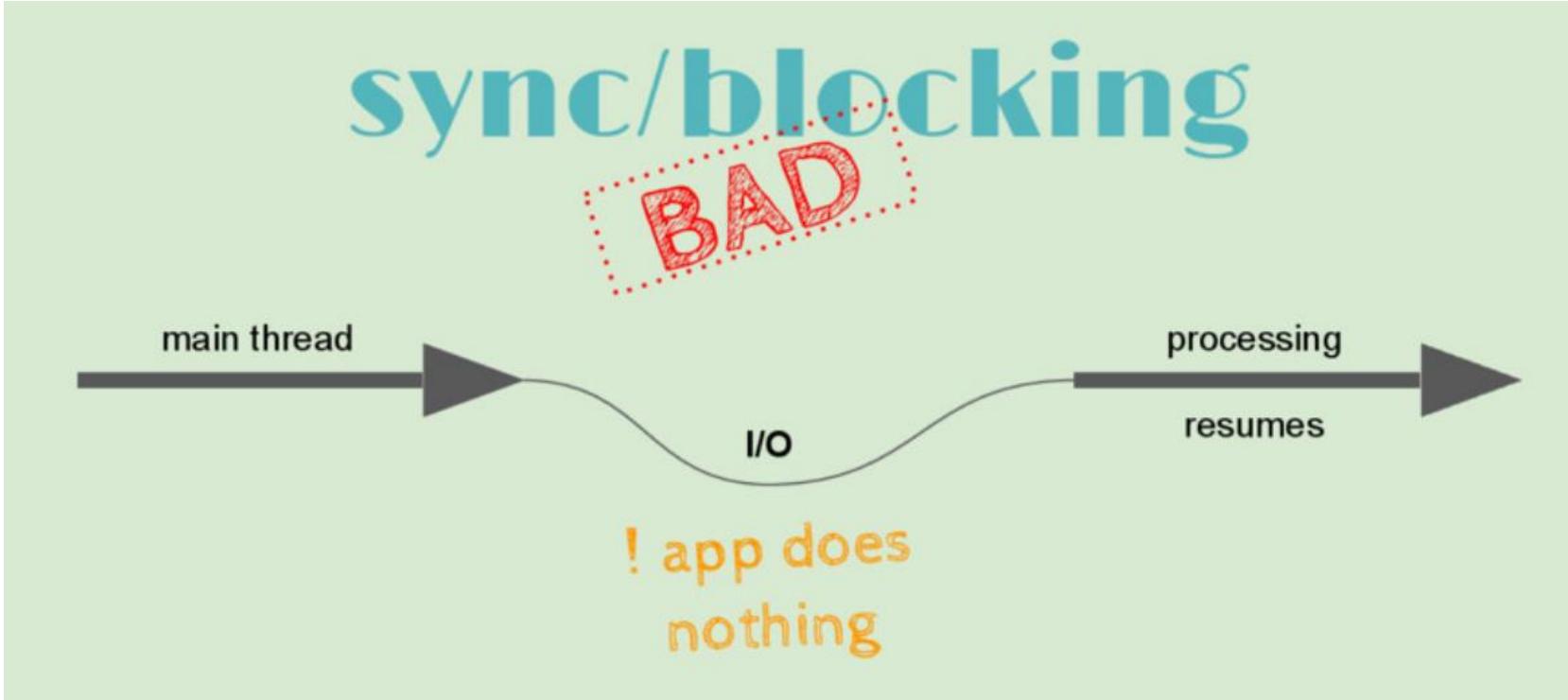
Gustafsons Gesetz: Ist bei großen Datenmengen jedoch oft passender



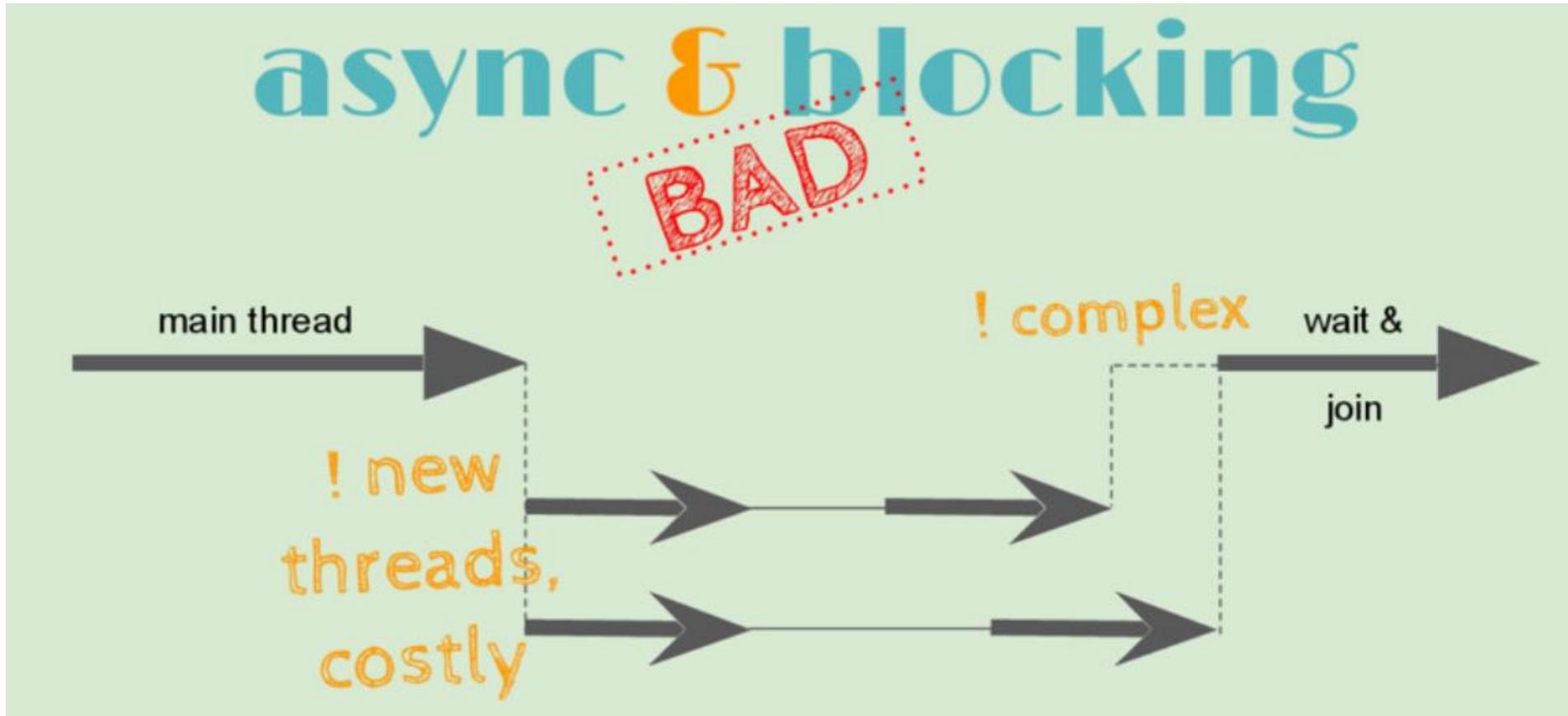
$$\text{Speedup} = \frac{1}{\frac{P}{N} + \cancel{S}}$$

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

Nicht-Parallele Programmierung



Parallele Programmierung mit Threads als Parallelisierungseinheit



re·ac·tive adjective \rē-'ak-tiv\

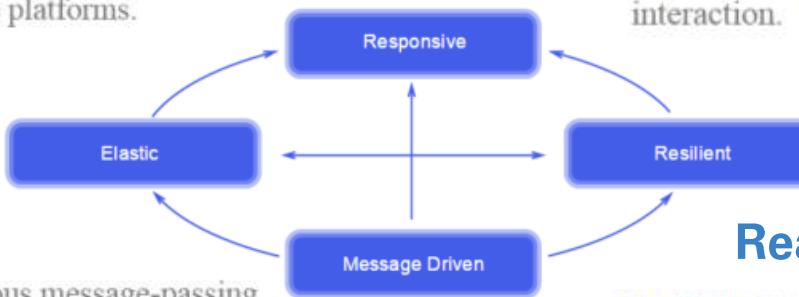
- 1 of, relating to, or marked by reaction or reactance
- 2 readily responsive to a stimulus

Das Programmiermodell der Cloud: 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 events / messages

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 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 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.

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

Dekomposition in Funktionen (auch **Aktoren**)

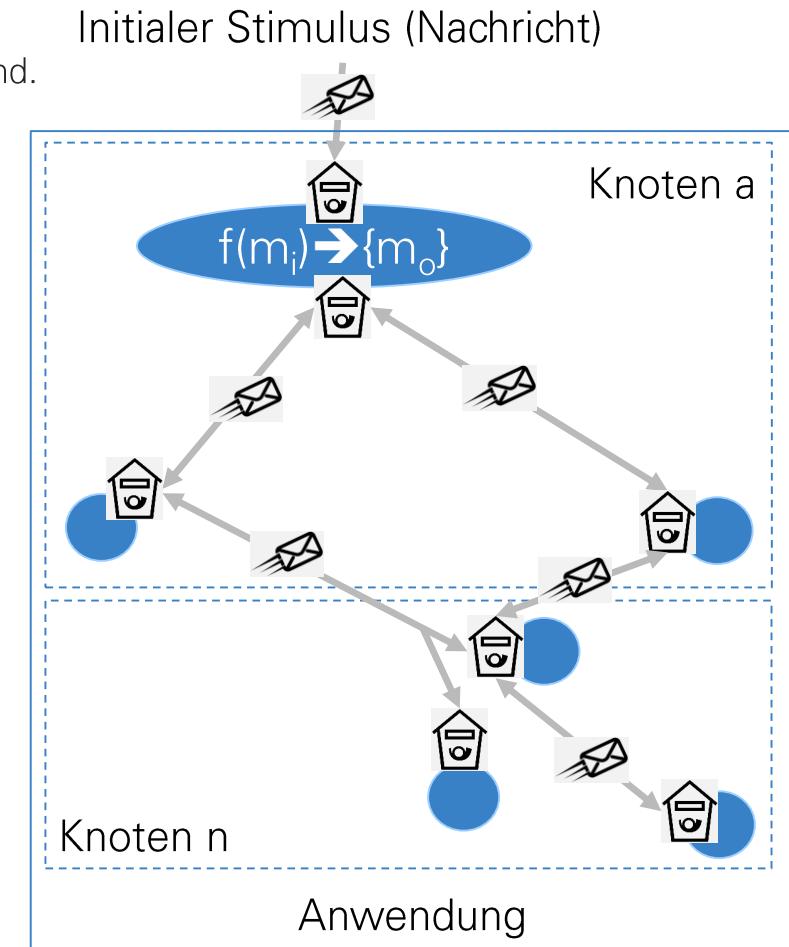
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wiederholbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)

Kommunikation zwischen den Funktionen über Nachrichten

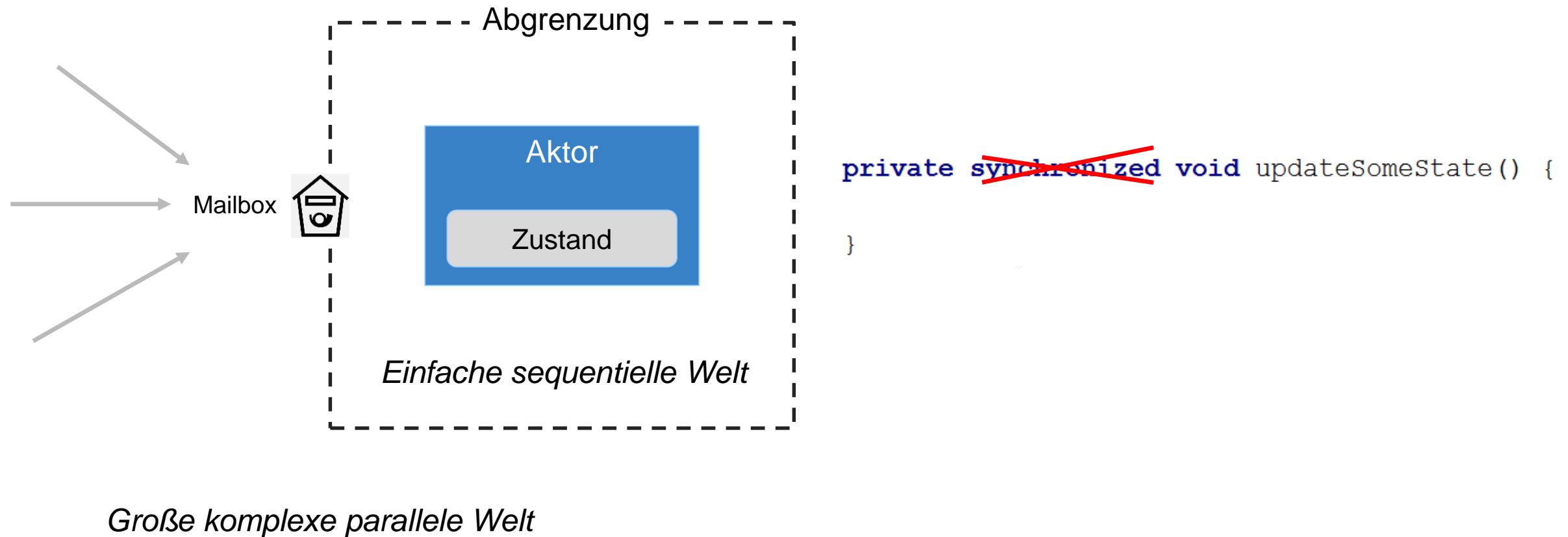
- asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber nicht auf sie.
- Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Produzenten und 1 Consumer)
- Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und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

Elastischer Kommunikationskanal

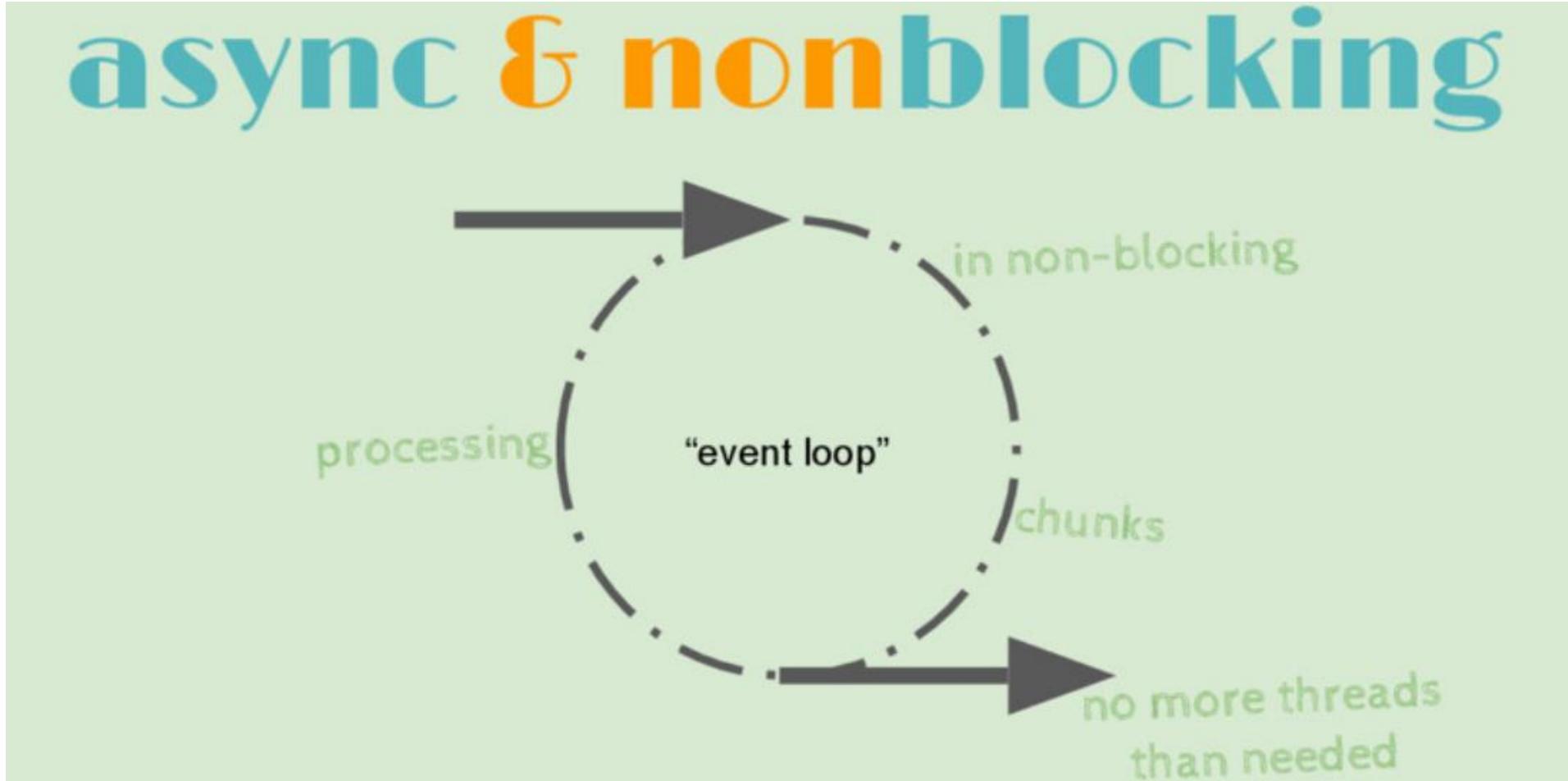
- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden (mehr oder minder) zuverlässig zugestellt
- Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)



Ein einzelner Aktor ist ein einfaches single-threaded Objekt, das über die Mailbox synchronisiert wird.



Nachrichten werden über eine Event Loop (aka Scheduler) zugestellt.



Reactive Programming am Beispiel

The akka logo consists of the word "akka" in a bold, white, sans-serif font. Above the letter "a", there is a teal-colored graphic element resembling a stylized mountain range or a series of curved arrows pointing upwards and to the right.

Reactive Programming mit 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.



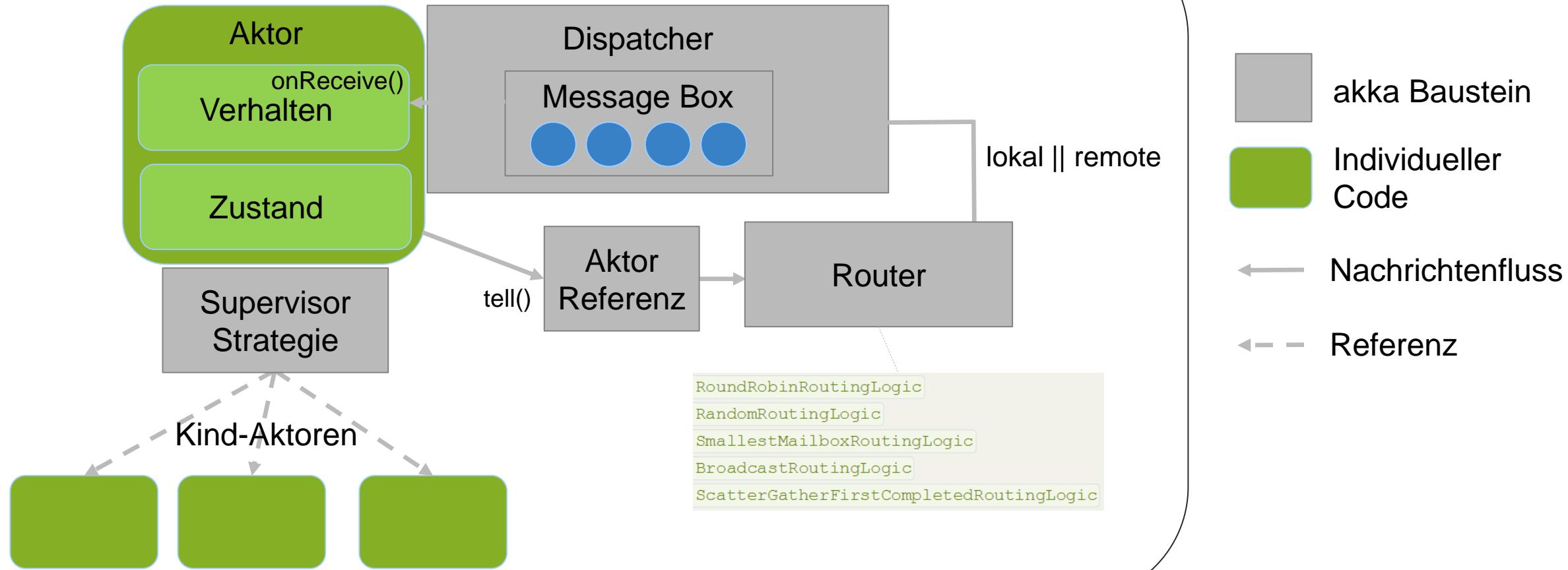
<http://akka.io>

Die Grundkonzepte von akka

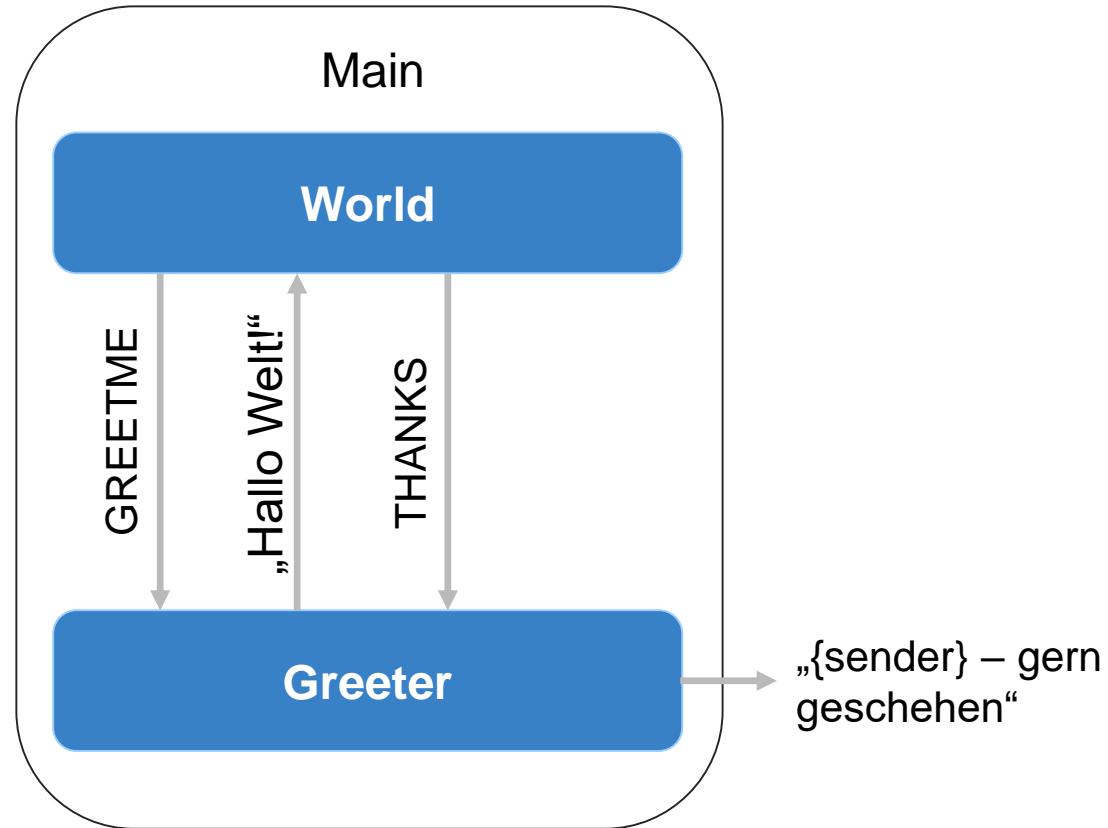
Konfiguration Aktor = Aktor-Klasse + Aktor-Name + Supervisor-Strategie + Dispatcher + Lokalität

Konfiguration Aktor Referenz = Aktor-Name + Router

Aktorensystem: Kennt die Aktoren und ihre Konfigurationen



Beispiel: Hello World



akka Hello World: Die Klasse Greeter

```
public class Greeter extends UntypedAbstractActor {

    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 UntypedAbstractActor {

    @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

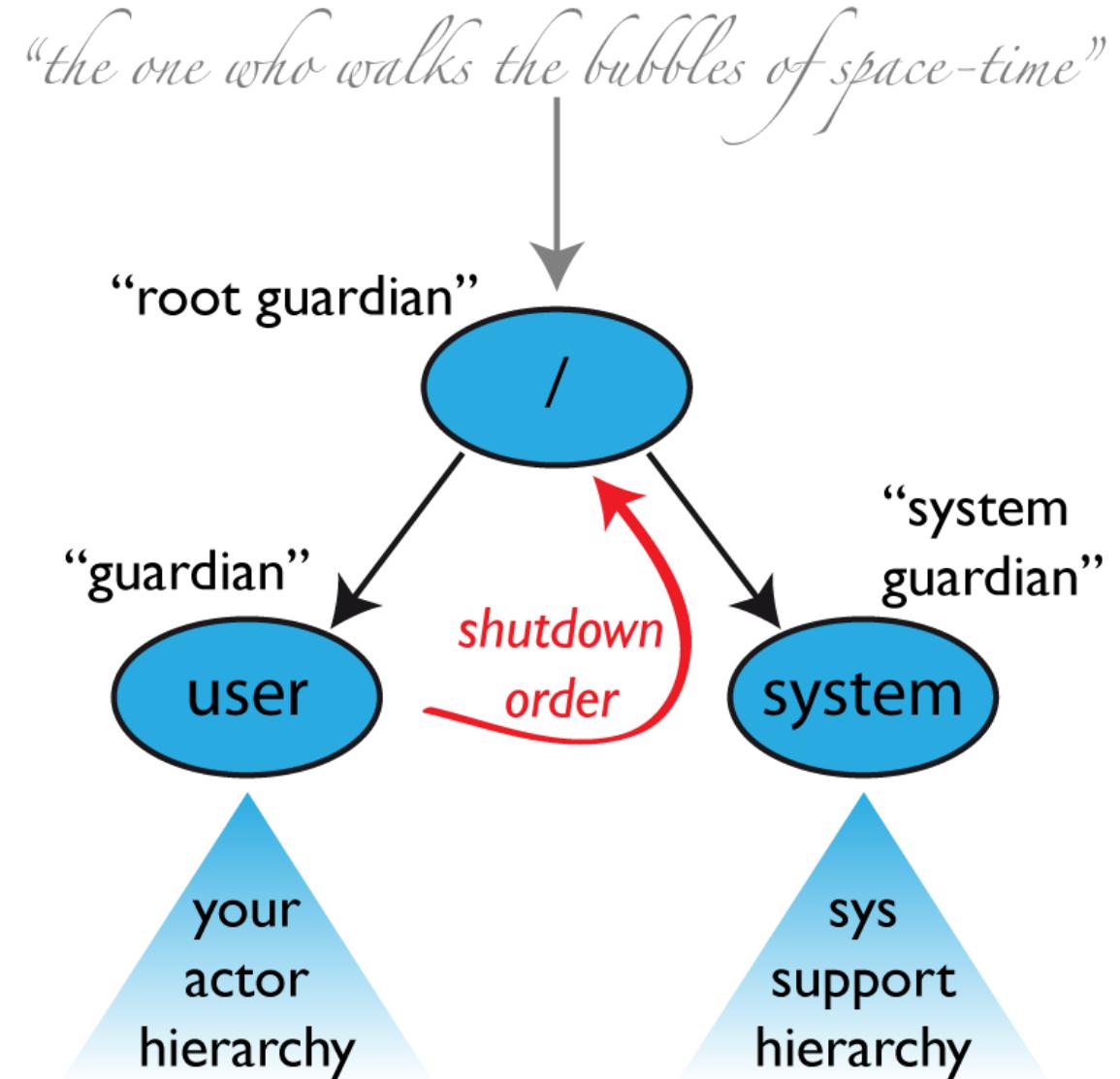
```
public class Main {  
    public static void main(String[] args) { akka.Main.main(new String[] {World.class.getName()}); }  
}
```

Parental Supervision in Akka

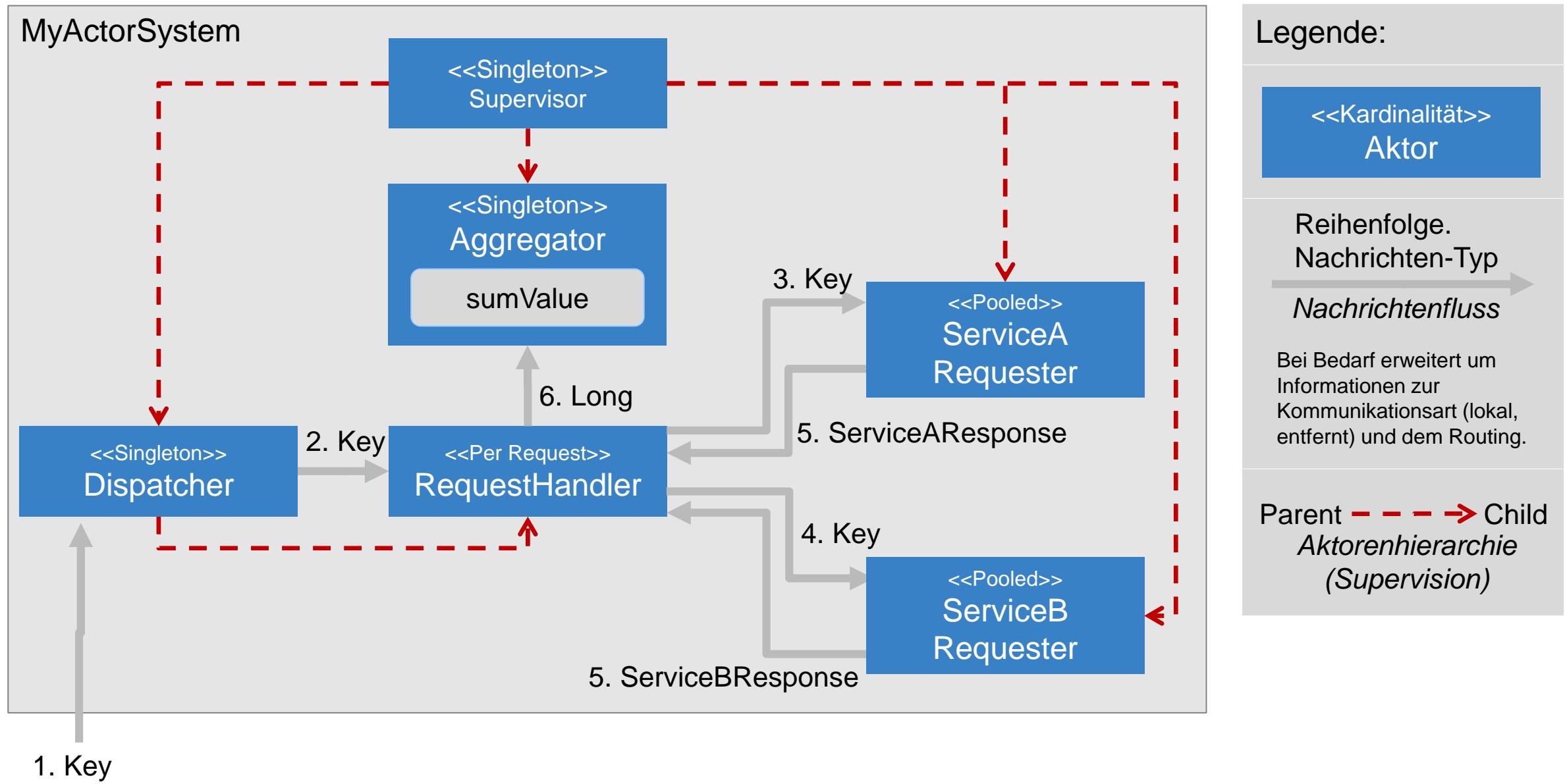
Zu jedem Aktor kann genau ein **Supervisor-Aktor** definiert werden. Standardmäßig ist es derjenige Aktor, der per `actorOf()` den Aktor erzeugt hat.

Der Supervisor-Aktor behandelt Exceptions, die in seinen untergeordneten Aktoren auftreten. Der Supervisor hat eine Strategie, die dann entscheidet ob:

- Einfach weiter gemacht wird
- Der Aktor oder alle Kind-Aktoren neu gestartet wird
- Der Aktor dauerhaft beendet wird
- Der Fehler eskaliert wird (Der Supervisor selbst schlägt dann fehl)



Ein Aktorensystem als Bild



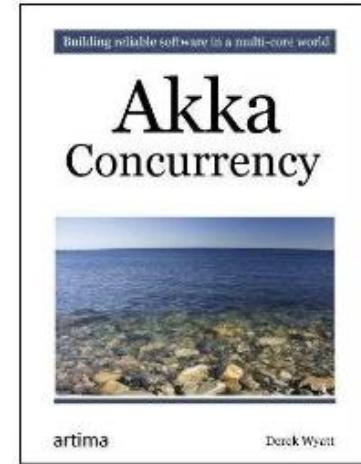
Links & Literatur

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>

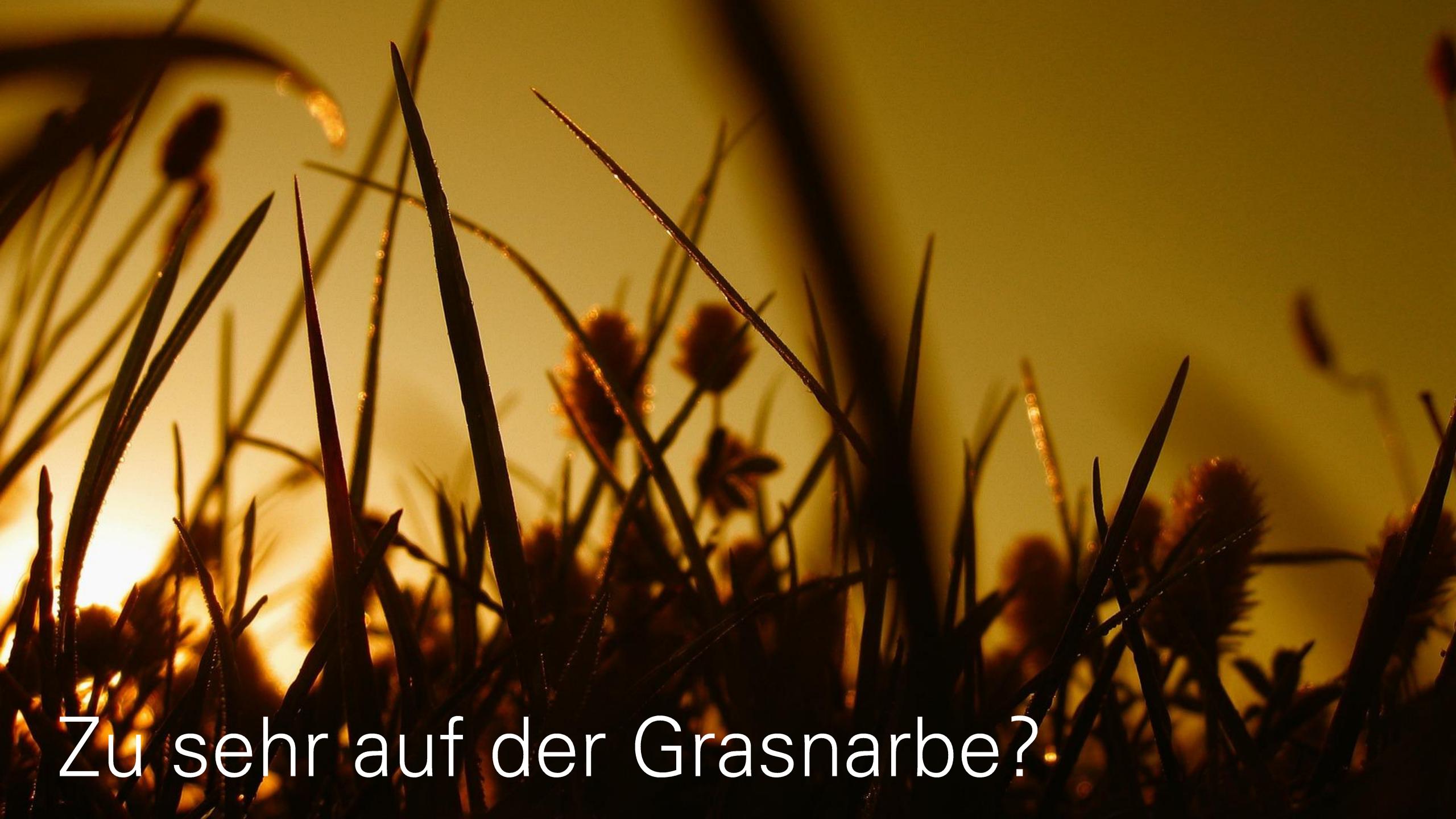
Akka

- <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)

Reactive Streams



Zu sehr auf der Grasnarbe?

Reactive Streams: 4 interfaces

- Publisher<T>
- Subscriber<T>
- Subscription
- Processor<T,R>

Eine Implementierung: Project Reactor



REACTIVE CORE

Reactor is a **fully non-blocking** foundation with efficient demand management. It directly interacts with Java 8 *functional API*, *CompletableFuture*, *Stream* and *Duration*.



TYPED [0|1|N] SEQUENCES

Reactor offers **2 reactive composable API** Flux [N] and Mono [0|1] extensively implementing Reactive Extensions.



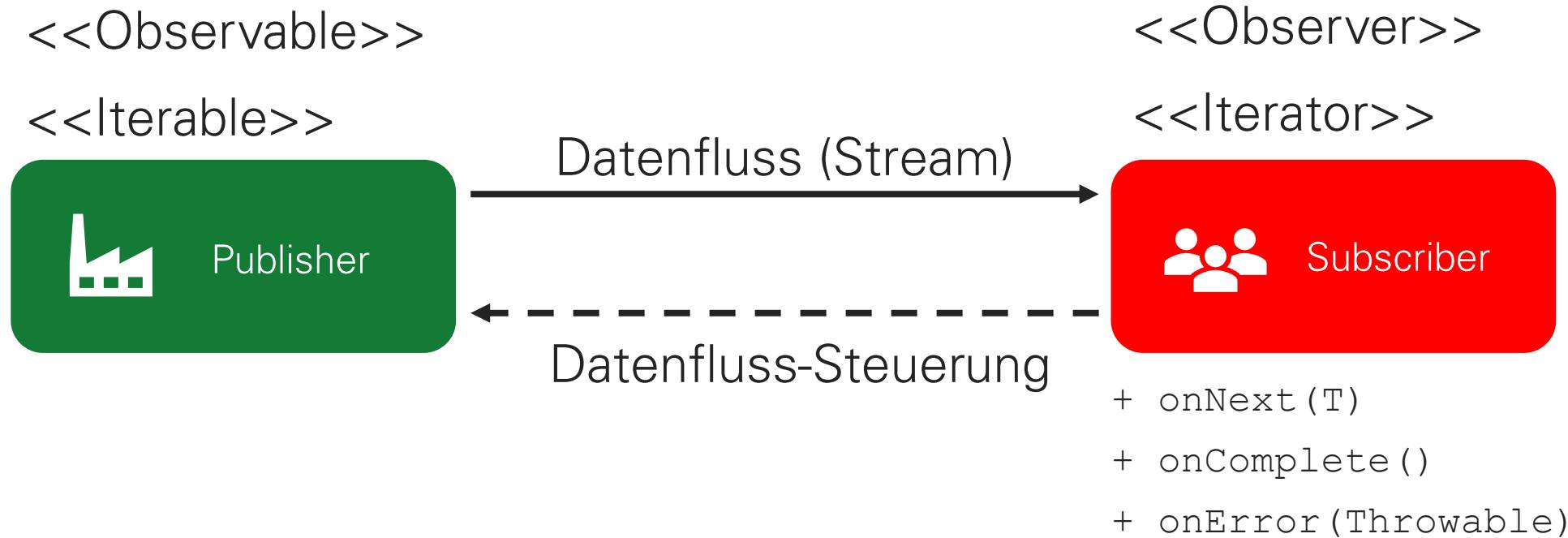
NON BLOCKING IPC

Suited for **Microservices Architecture**, Reactor IPC offers **backpressure-ready network engines** for HTTP (including Websockets), TCP and UDP. Reactive Encoding/Decoding is fully supported.

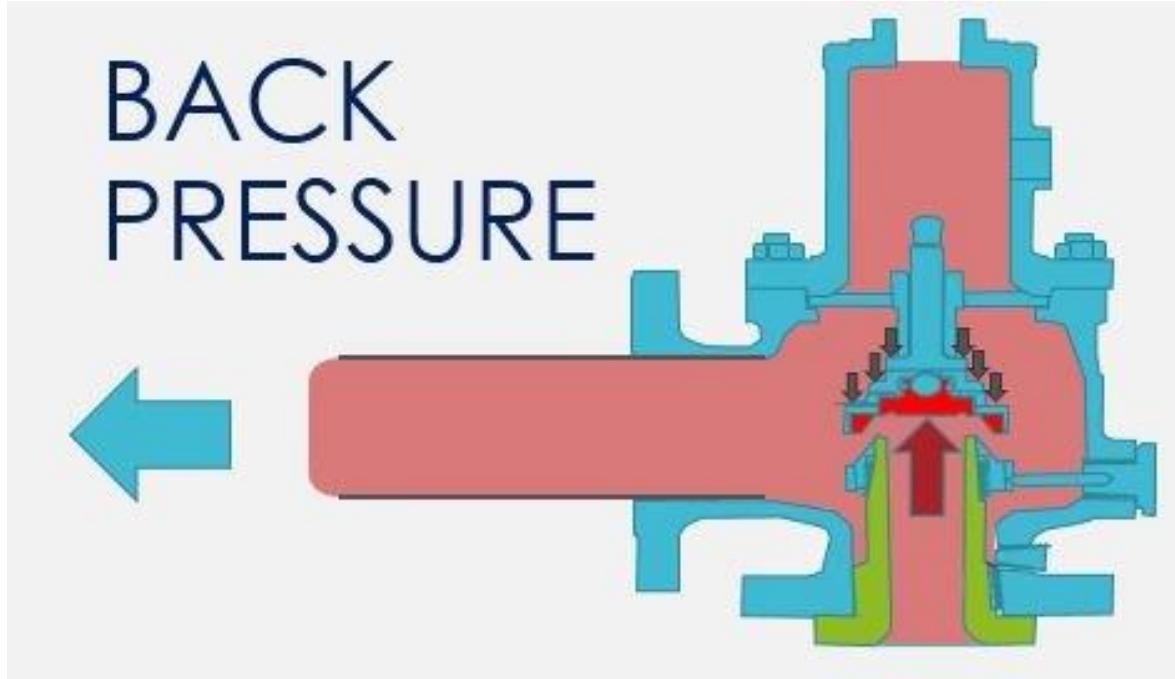
Noch eine Implementierung: ReactiveX / RxJava



Reactive Streams: Das Fundament ist eine Kombination aus Observer- und Iterator-Pattern.



Back Pressure: Umgang mit Überlast

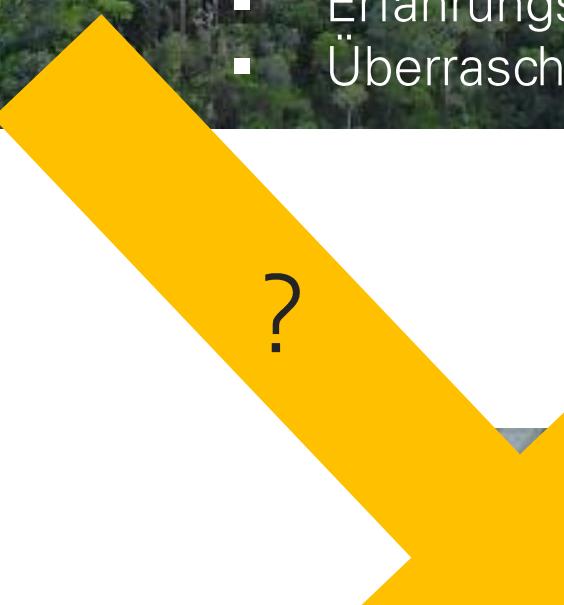




Zufluss an Daten und Anfragen

- Erfahrungsgemäße Abweichungen (Ebbe, Flut)
- Überraschende Abweichungen (Hochwasser, Dürre)

?



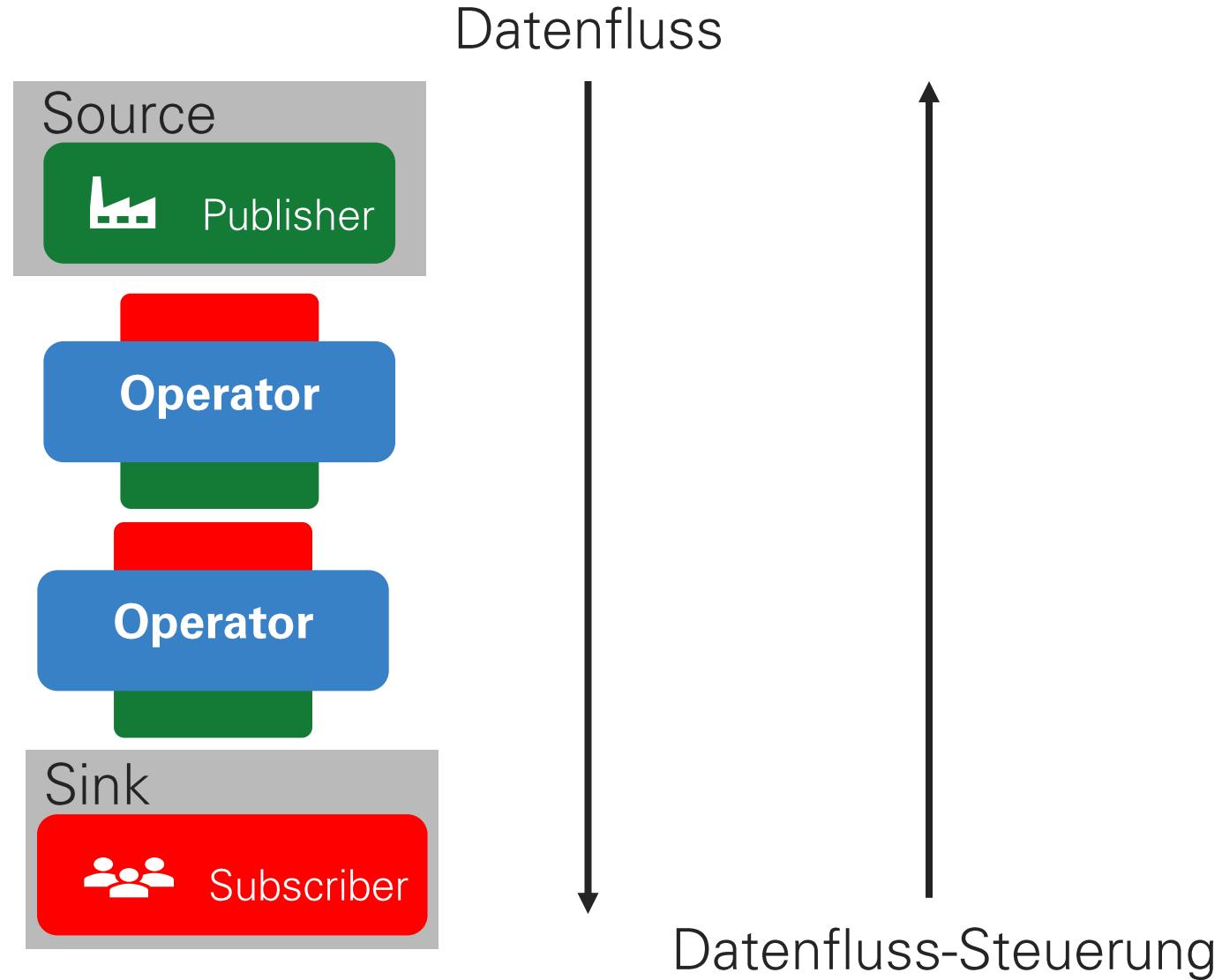


- 3 Staut in einem großen Becken auf
- 2 Reguliert die Schleuse entsprechend, dass der aktuelle max. Durchfluss nie überschritten wird
- 1 Meldet, wie viel max. Durchfluss aktuell möglich ist



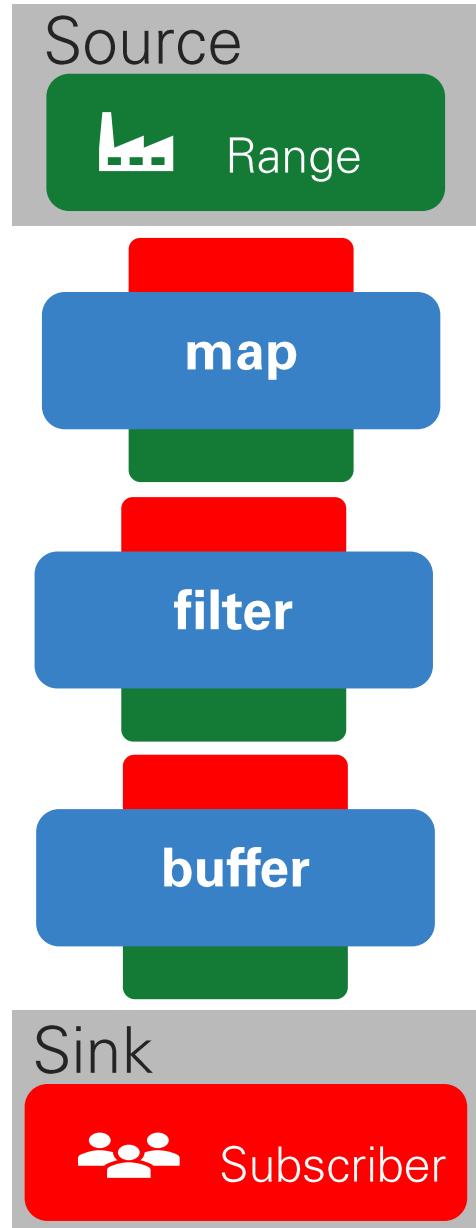
Reactive Streams: Das Programmiermodell

z.B. „lese zeilenweise von Datenbank“

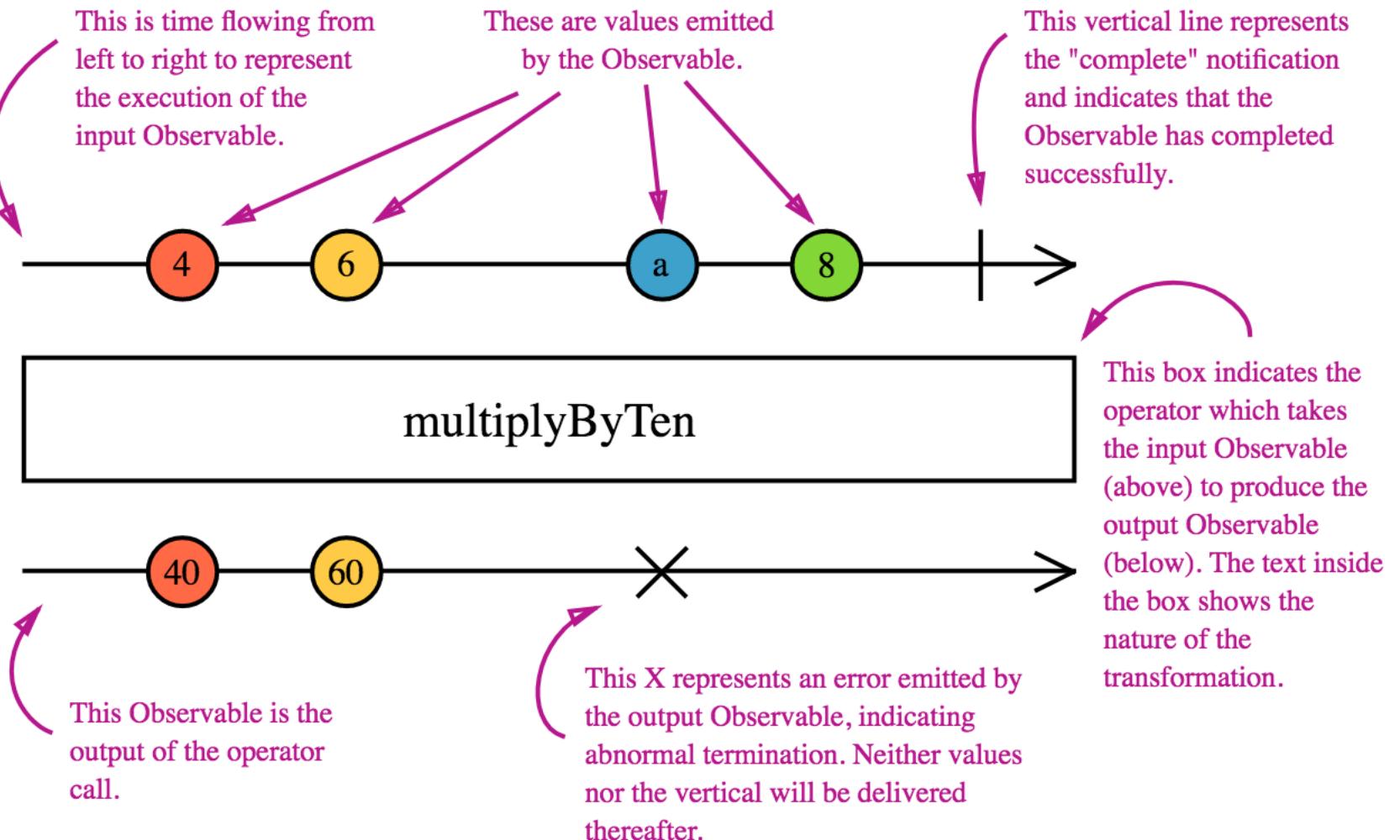


z.B. „sende AMQP-Nachrichten an Client“

Reactive Streams: Beispiel



Marble Diagrams



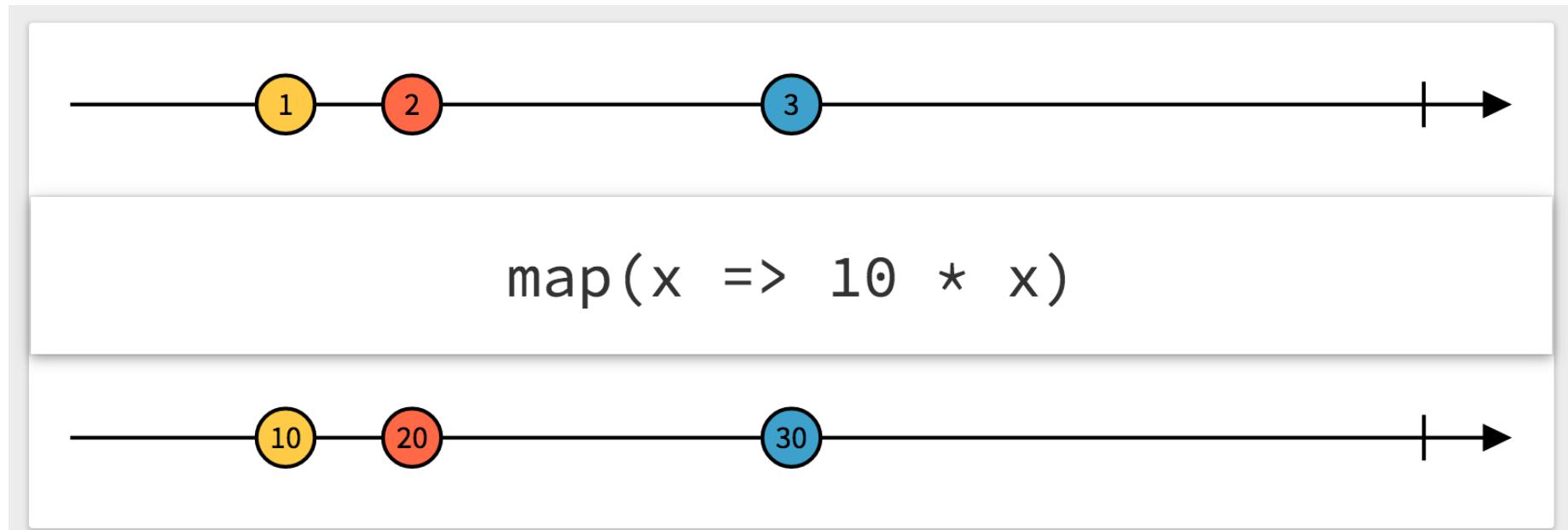
Source
Range

map

filter

buffer

Sink
Subscriber



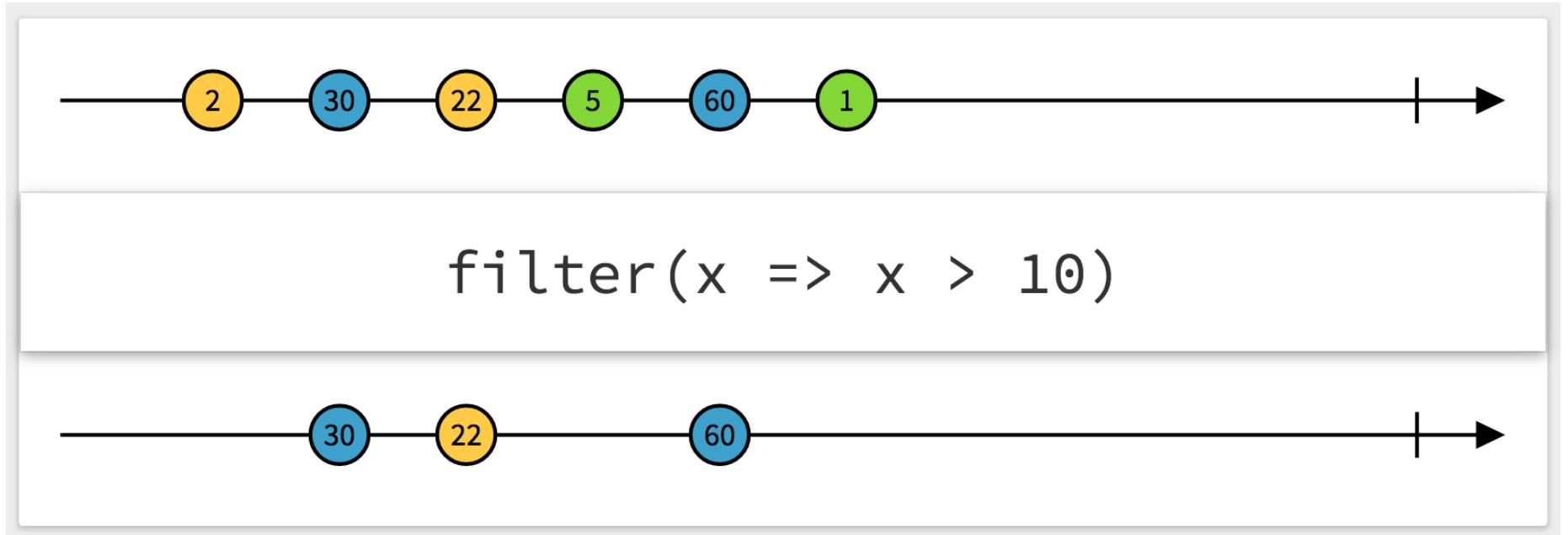
Source
 Range

 map

 filter

 buffer

Sink
 Subscriber



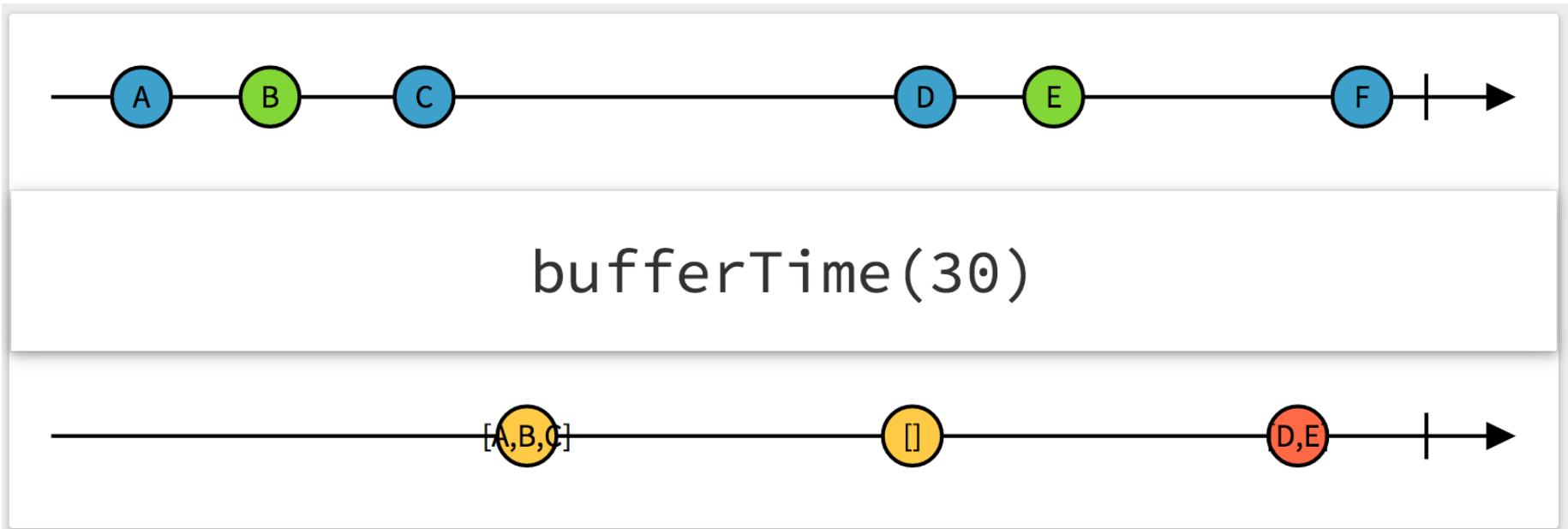
Source
Range

map

filter

buffer

Sink
Subscriber



```
Flux.range(1, 1000000000)
    .map(i -> i + 3)
    .filter(i -> i % 2 == 0)
    .bufferTime(1)
    .take(100000000)
```

Reactive Streams Implementierungen für Java



Project
Reactor

Quellen

- Reactive Streams Website: <http://www.reactive-streams.org>
- Reactive Streams Tutorial: <https://egghead.io/courses/asynchronous-programming-the-end-of-the-loop>
- Intro zu Reactive Programming: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>
- Beispiel zu Project Reactor: <https://github.com/mkheck/flux-flix-intro>
- Lernmaterialien zu Project Reactor: <https://projectreactor.io/learn>
- Reactive Streams in Java: <https://dzone.com/articles/what-are-reactive-streams-in-java>
- Visualisierung von Operatoren mit Marble Diagrammen: <http://rxmarbles.com>
- Einführung zu Marble Diagrammen: <https://medium.com/@jshvarts/read-marble-diagrams-like-a-pro-3d72934d3ef5>

Einsatzbereiche

Illustrative Benchmarks

Szenario	Non-reactive			Reactive		
findall_empty 4 threads and 100 connections	Avg Latency	5.92ms	Stdev 2.51ms	Avg Latency	6.13ms	Stdev 3.92ms
	Req/Sec	4.31k	433.04	Req/Sec	4.31k	744.88
	514556 requests in 30.04s			515231 requests in 30.05s		
findall_1000 4 threads and 100 connections	Avg Latency	233.74ms	Stdev 150.88ms	Avg Latency	430.03ms	Stdev 46.03ms
	Req/Sec	111.43	29.86	Req/Sec	57.83	18.43
	13317 requests in 30.04s			6930 requests in 30.03s		
findall_empty_8threads 8 threads and 1000 connections	Avg Latency	3.27s	Stdev 3.08s	Avg Latency	1.20s	Stdev 354.71ms
Mit künstlicher Latenz	Req/Sec	44.98	60.37	Req/Sec	205.58	247.15
	5500 requests in 30.07s			24469 requests in 30.06s		
	Socket errors: connect 0, read 0, write 0, timeout 800					

Bewertung

Vorteile

- Höherer Durchsatz bei IO-intensiven Anwendungen
- Der Prozessor befindet sich weniger Zeit in Wartezuständen (IO-Wait, Lock-Wait, ...)
- Der Hauptspeicherverbrauch ist niedriger durch häppchenweise Streams
- Die CPU und der Hauptspeicher sind für weiteren Durchsatz frei
- Toleranteres Verhalten im Hochlastbereich
 - Back Pressure
 - Leichtgewichtige Skalierungsressourcen (Aktoren, Scheduler)

Nachteile

- Performance-Einbußen bei CPU-intensiven Anwendungen
- Ungewohntes aber gut lesbares Programmiermodell
- Over-engineered für Anwendungen mit normaler Last und moderatem Durchsatz

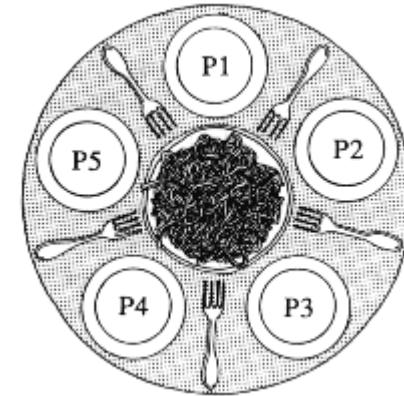
Bonusmaterial

Beispiel: Das Philosophenproblem

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

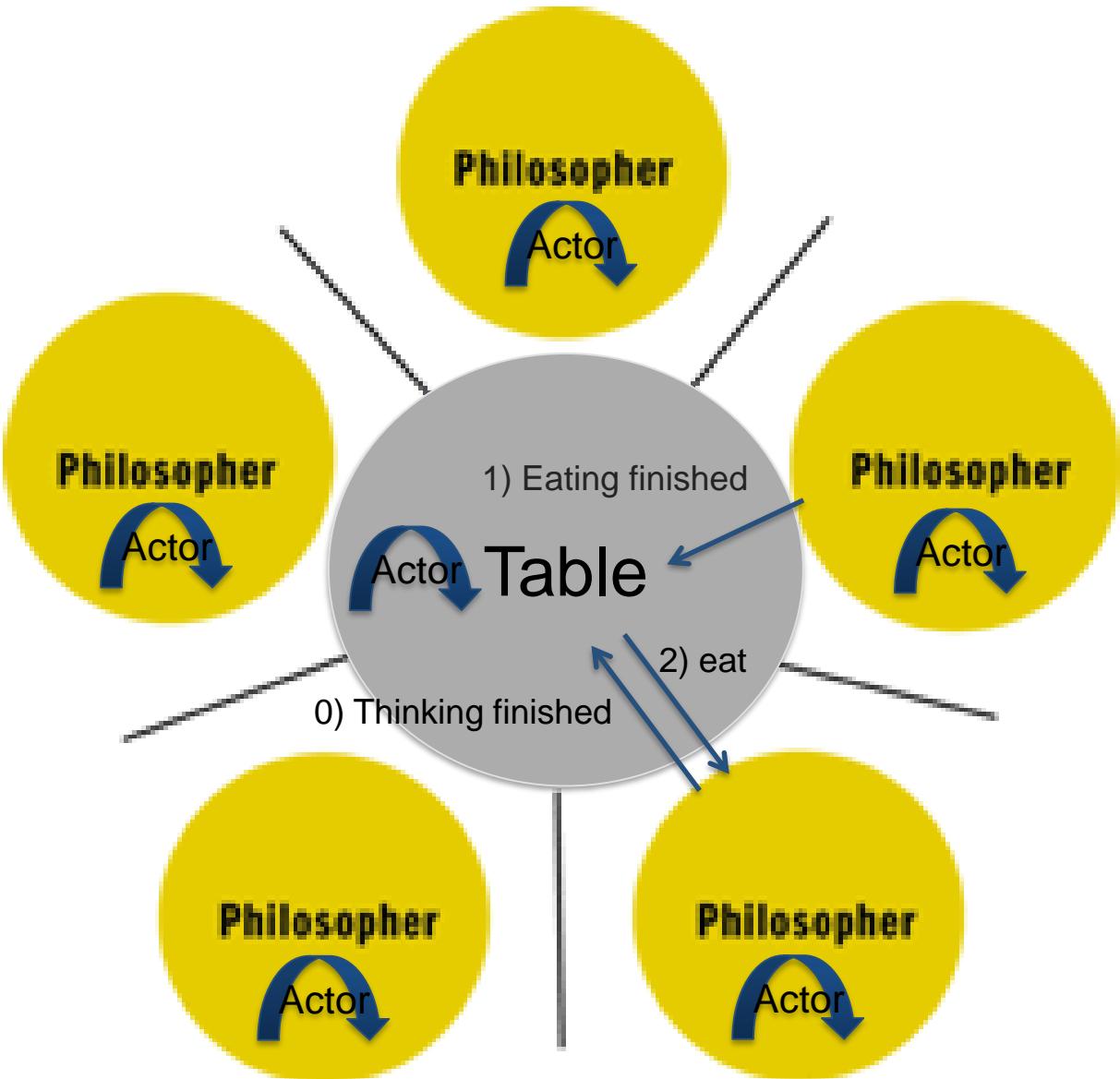
Umsetzung mit akka:

- 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



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.

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));  
    }  
}
```

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).

Actor Implementierungsregeln

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.

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 garantie Zustellung (zu sehr hohen Kosten), bieten aber auch keine echten Garantien.
- Es gibt Patterns, wie man mit diesen fehlenden Garantien umgehen kann.