

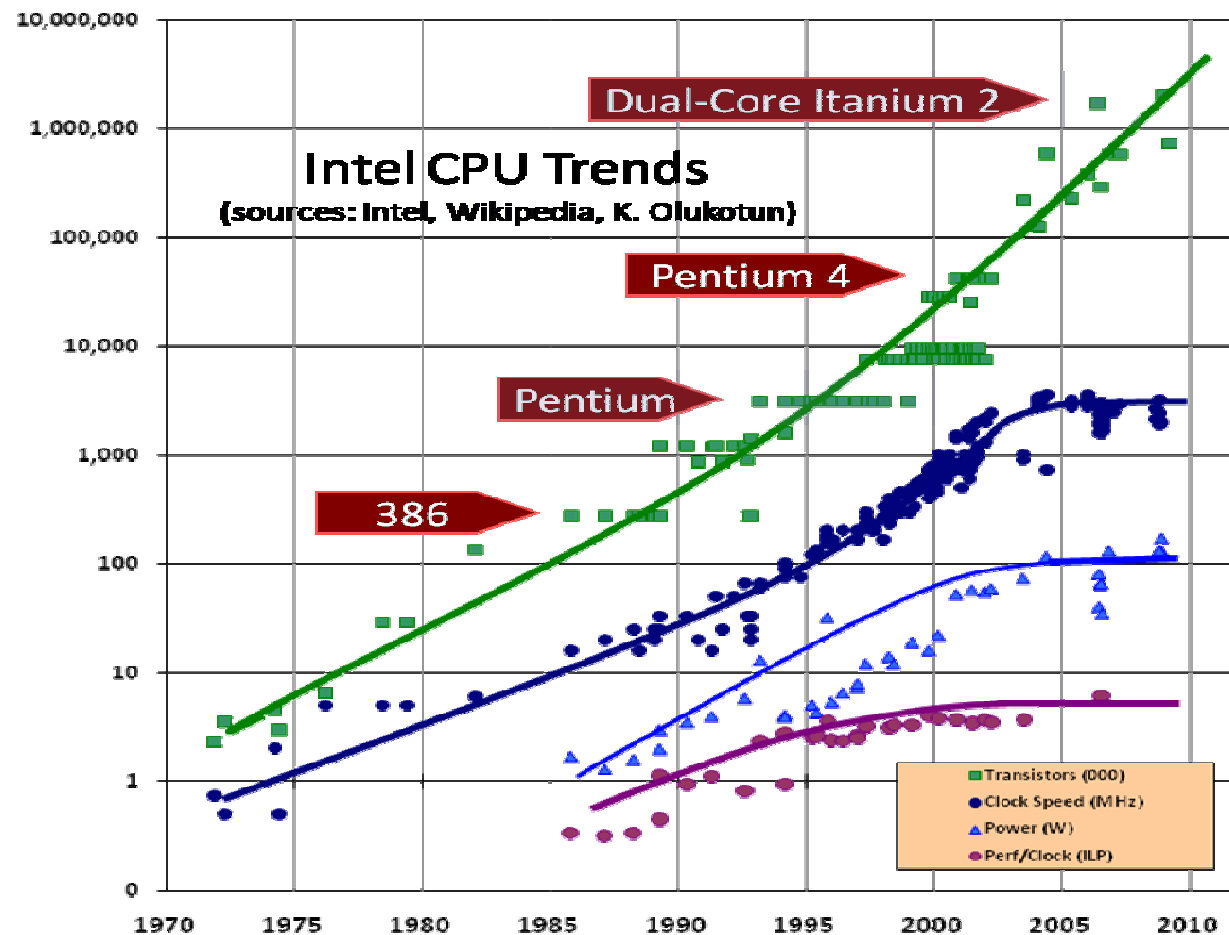
We Are Reactive

Cloud Computing

Kapitel 2: Programmiermodelle für die Cloud (Reactive Programming)

Dr. Josef Adersberger

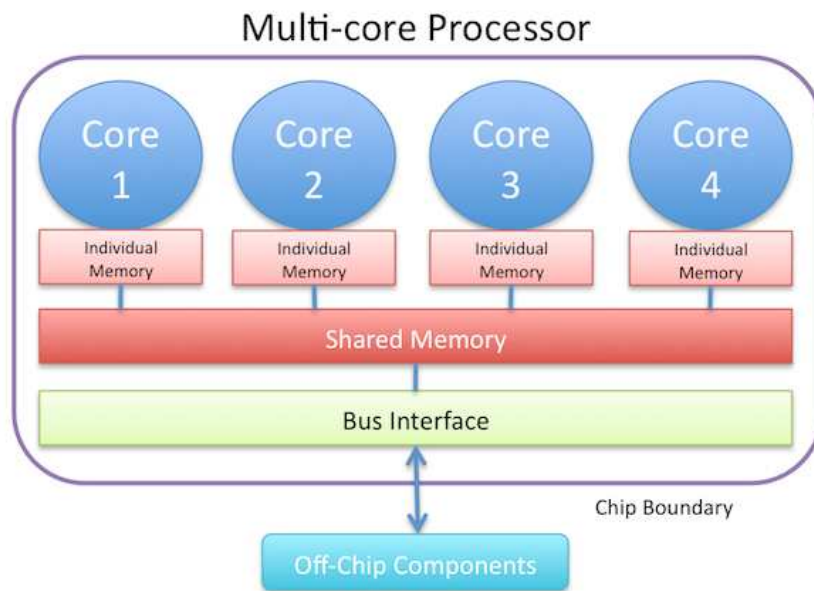
„The free lunch is over“: Es gibt keine kostenlose Performanzsteigerung mehr – Nebenläufigkeit zählt.



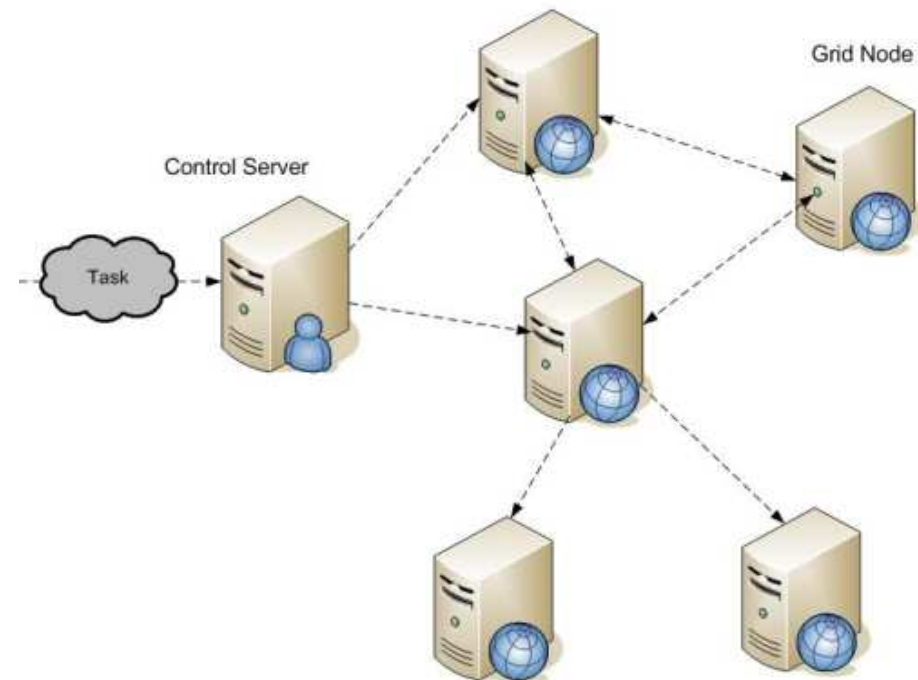
← **Anzahl Transistoren**
Moore's Law gilt weiterhin

← **Taktfrequenz**
Seit 2004 ist die Taktfrequenz von CPUs konstant

Nebenläufigkeit kann im Kleinen und im Großen betrieben werden.

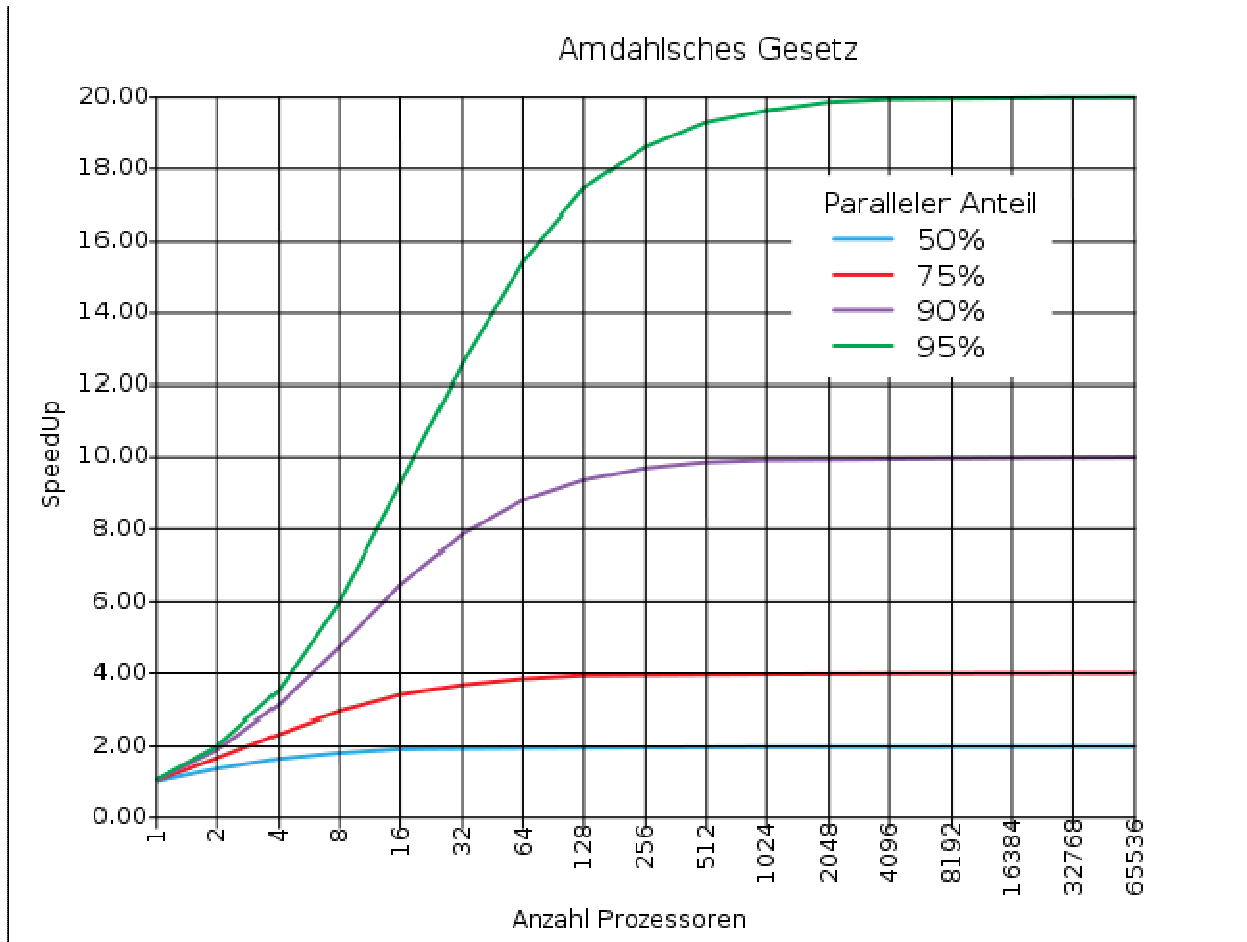


Multi Core



Multi Node
(Cluster, Grid, Cloud)

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

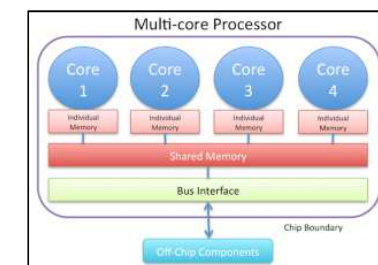


P = Paralleler Anteil
S = Sequenzieller Anteil
N = Anzahl der Prozessoren
Speedup = Maximale Beschleunigung

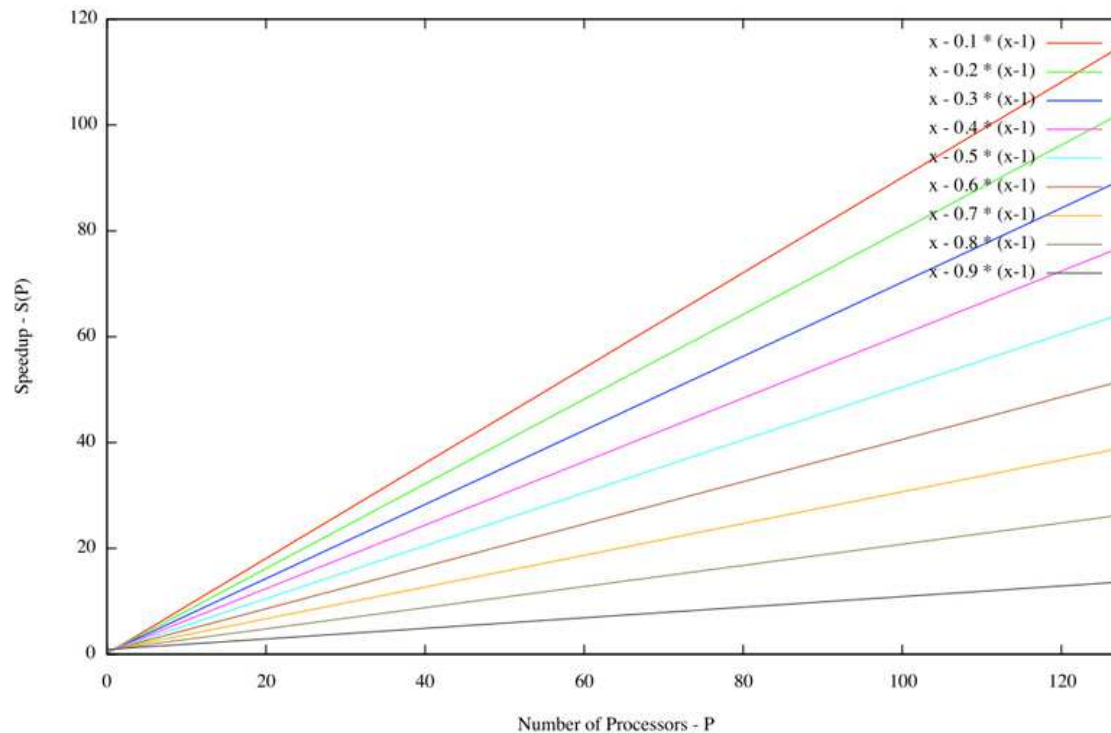
$$Speedup = \frac{1}{1 - P} \quad \text{für } N = \infty$$

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

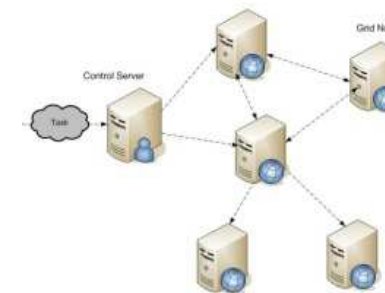
Gilt insbesondere bei:



Bei Multi-Node-Nebenläufigkeit ist Gustafsons Gesetz jedoch oft passender.



Gilt insbesondere bei:



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

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

Das Programmiermodell der Cloud: Functional Reactive Programming

re·ac·tive adjective \rē-'ak-tiv\

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

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

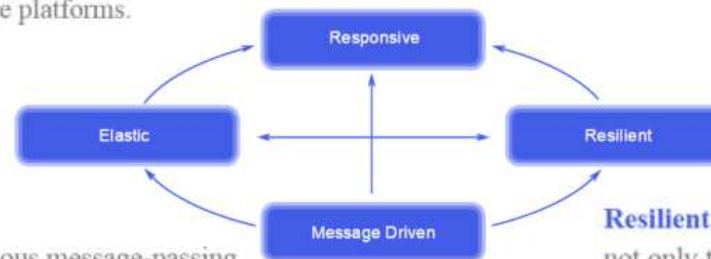
Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard 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 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.



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



What is Reactive Programming?

Principles of Reactive Programming

Martin Odersky



https://www.youtube.com/watch?v=7D9QfMj_KwI

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

■ Dekomposition in Funktionen (auch Aktoren)

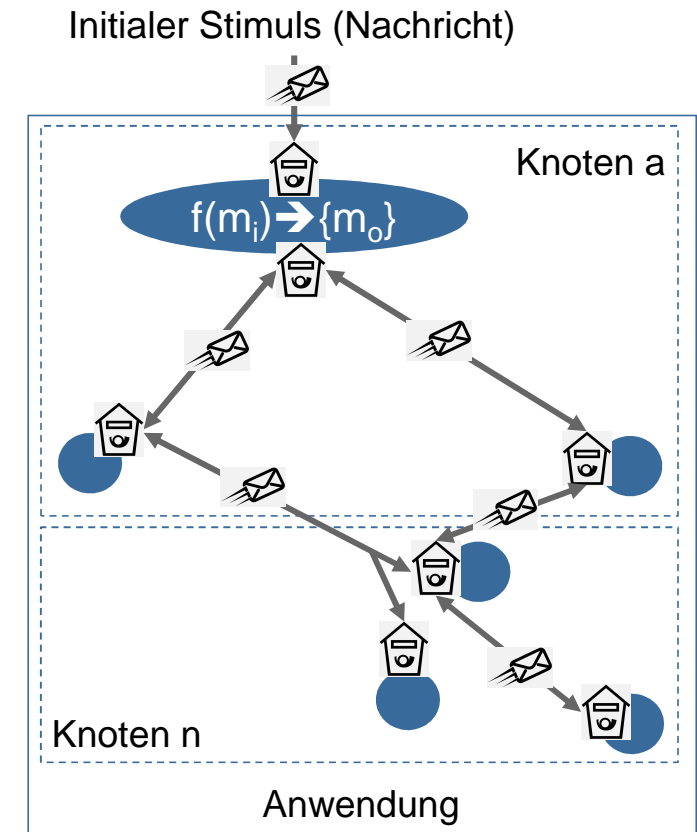
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wiederaufsetzbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)

■ 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

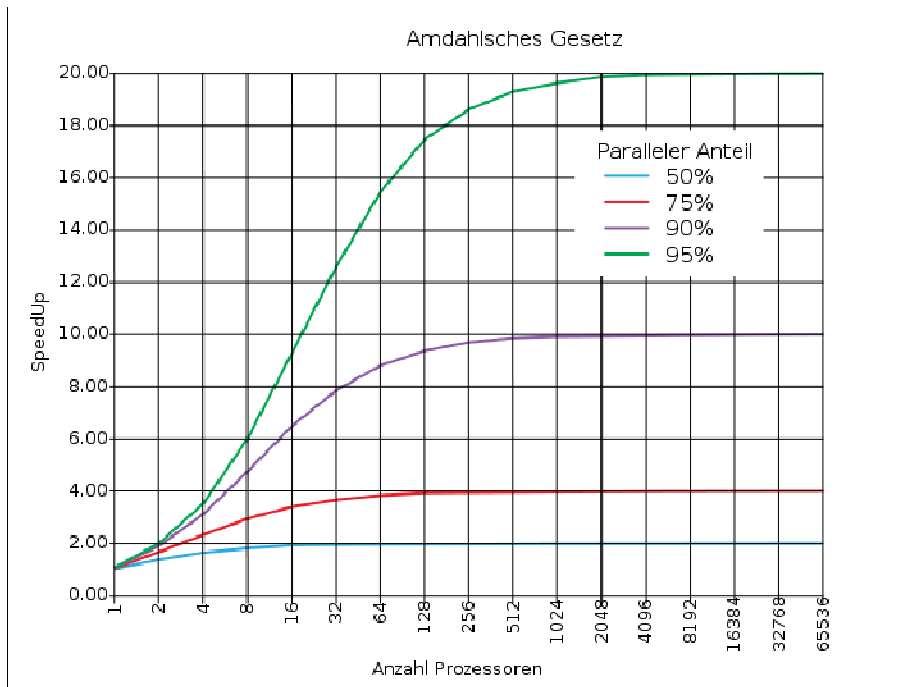
- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden zuverlässig zugestellt
- Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)



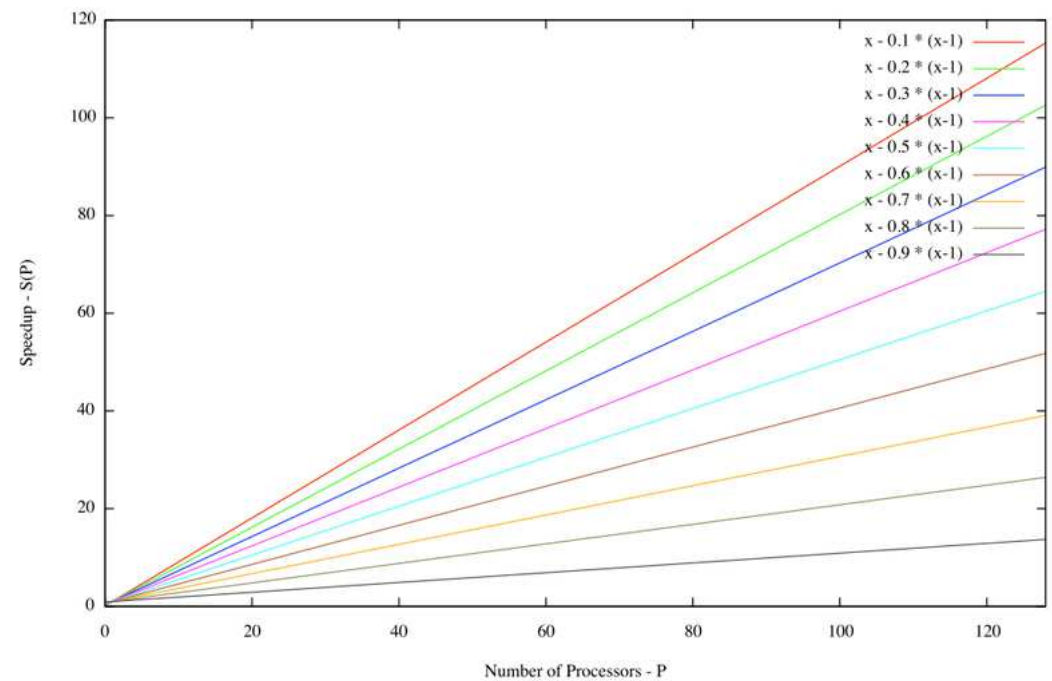
Gegenüber synchronen Systemen bietet FRP einige kontextsensitive Vorteile.

- **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.

Amdahl und Gustafson revisited



→ Höherer Parallelisierungsgrad



→ Dynamische Skalierung je nach Last

Message Driven - Responsive - Reactive - Elastic
REACTIVE PROGRAMMING

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.



The diagram features five hexagonal icons arranged in a circle, each representing a key feature of the Akka framework. The icons are: a speedometer for High Performance, a network of nodes for Elastic & Decentralized, puzzle pieces for Extensible, a self-healing cell for Resilient by Design, and server racks for Simple Concurrency & Distribution.

- Simple Concurrency & Distribution**
Asynchronous and Distributed by design. High-level abstractions like Actors, Futures and STM.
- Resilient by Design**
Write systems that self-heal. Remote and/or local supervisor hierarchies.
- High Performance**
50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.
- Elastic & Decentralized**
Adaptive load balancing, routing, partitioning and configuration-driven remoting.
- Extensible**
Use Akka Extensions to adapt Akka to fit your needs.

<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

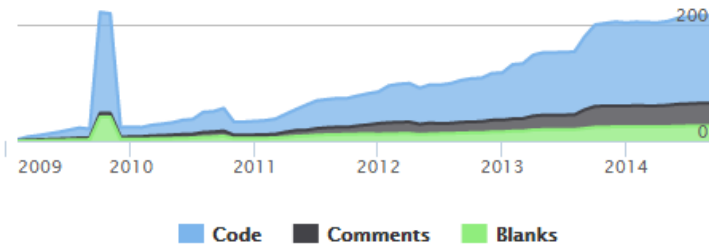
... is mostly written in Scala with an average number of source code comments

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

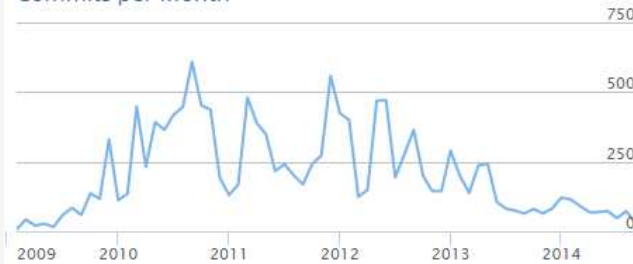
... took an estimated 39 years of effort (COCOMO model) starting with its first commit in February, 2009 ending with its most recent commit 6 days ago

Licenses: Apache-2.0

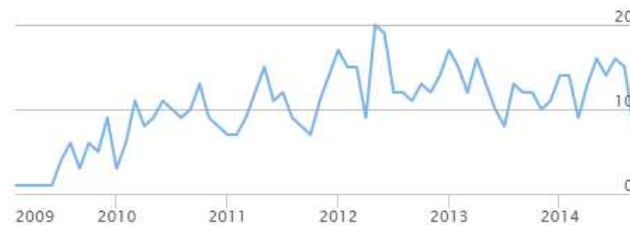
Lines of Code



Commits per Month



Contributors per Month

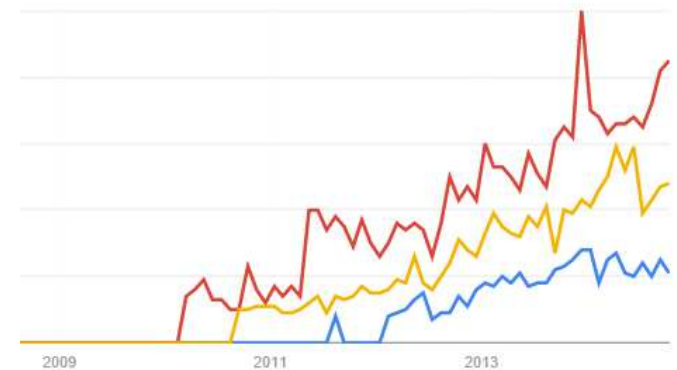


<https://www.google.de/trends>

akka frame...
Suchbegriff

akka scala
Suchbegriff

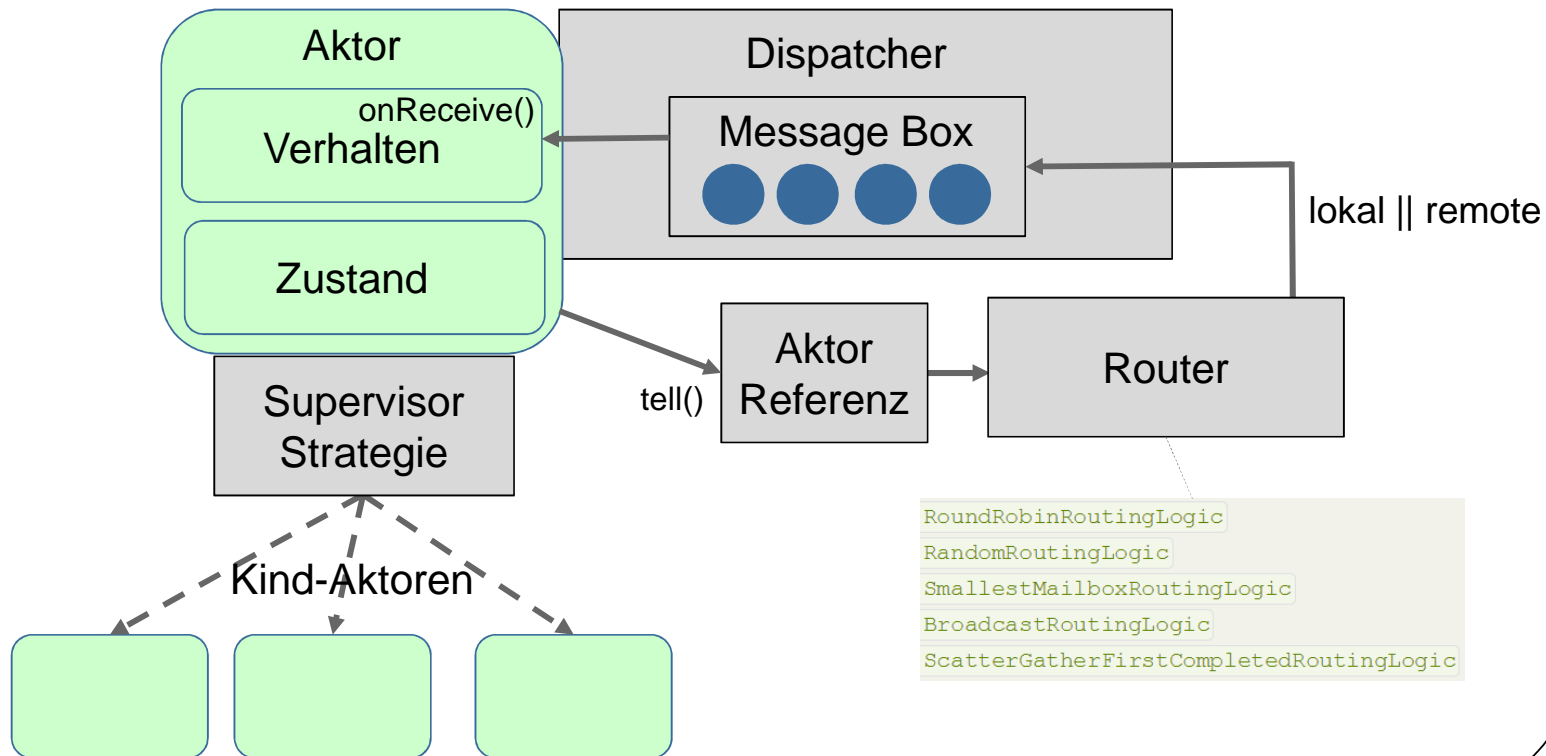
akka java
Suchbegriff



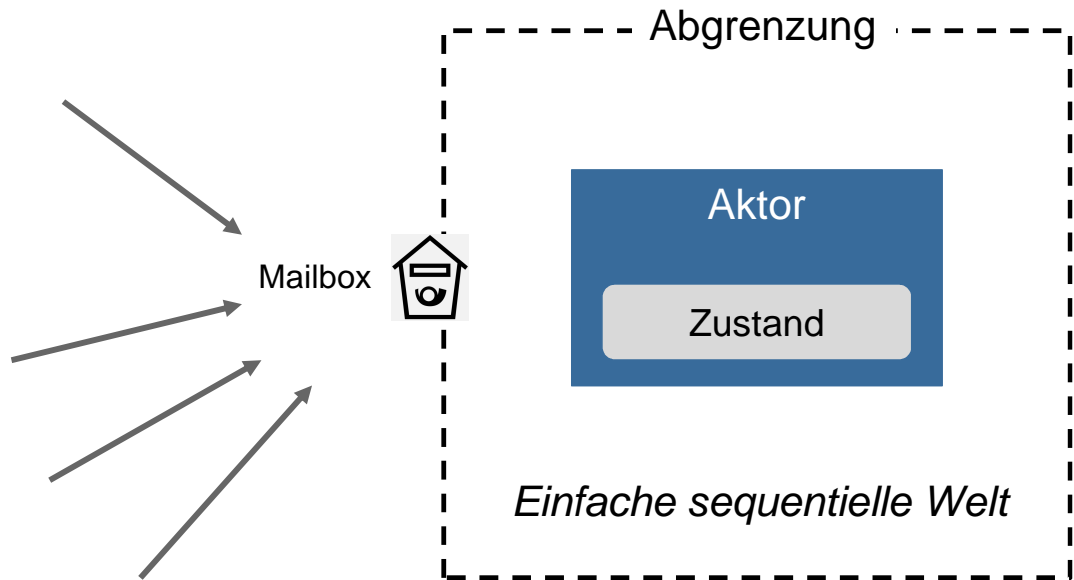
Die Grundkonzepte von akka

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

Aktorensystem: Kennt die Aktoren und ihre Konfigurationen



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



Große komplexe parallele Welt

```
private synchronized void updateSomeState() {  
    }  
}
```

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

Actor Best Practices

■ Do not block!

- Ein Akteur 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.**

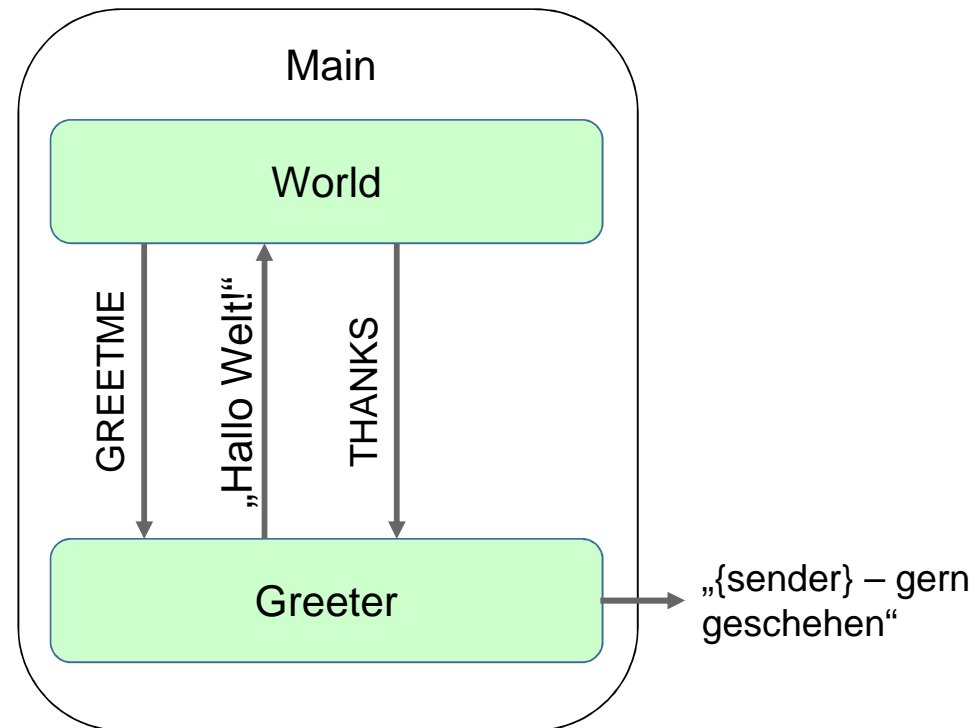
■ Pass immutable Messages. Do not close over actor state!

- Eine Nachricht darf syntaktisch ein beliebiges Objekt sein. Verknüpfe sie aber nicht mit dem Zustand des Akteurs. Sonst bist du zurück in der Hölle der unkontrollierten Parallelität (d.h., „lasset alle Hoffnung fahren“).

■ Let it crash!

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

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()}); }  
}
```


API: *UntypedActor*

ActorContext

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, reversed).
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 "\$".

ActorRefs

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. Iterable).
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).

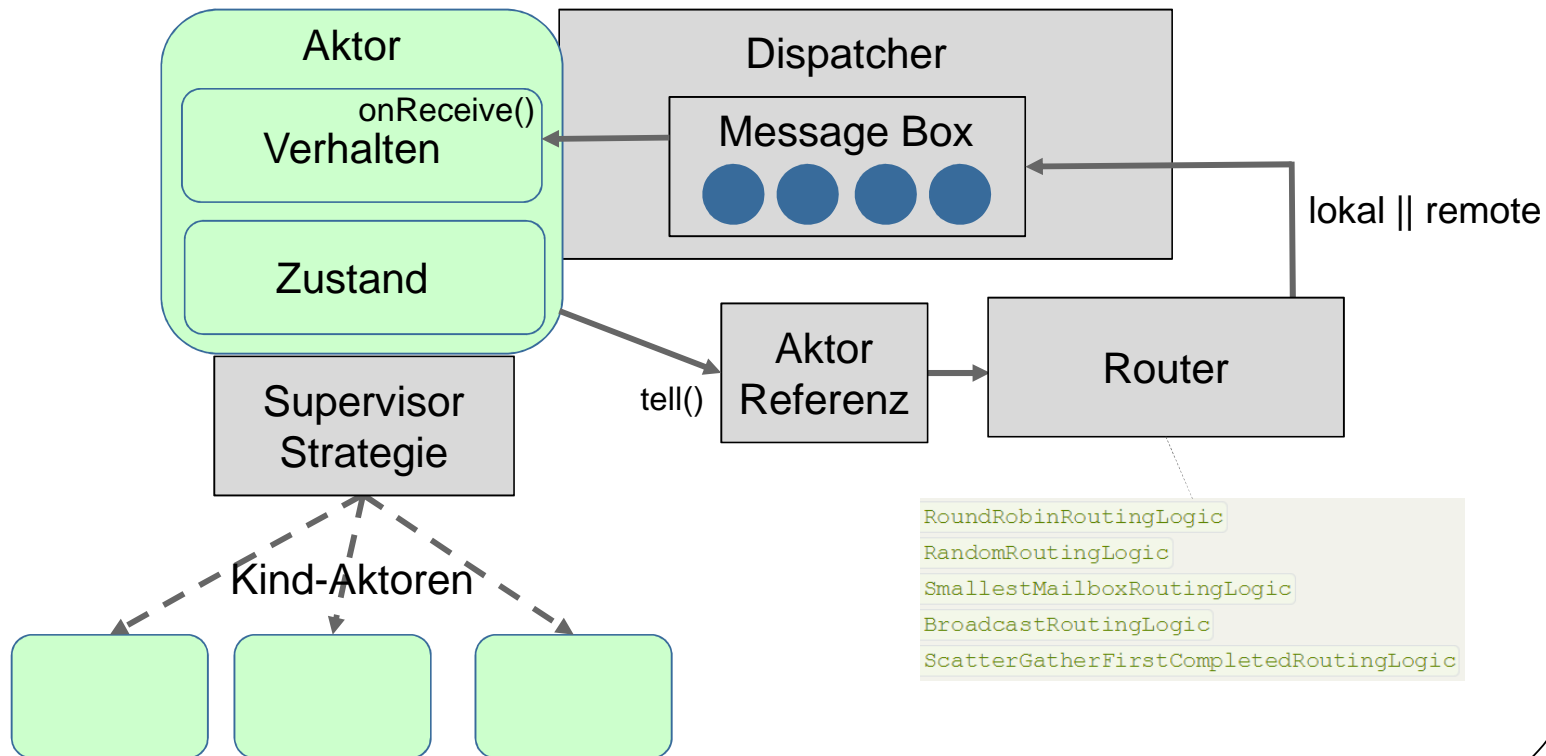
[`tell`](#)(java.lang.Object msg, [`ActorRef`](#) sender)

Sends the specified message to the sender, i.e.

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



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).

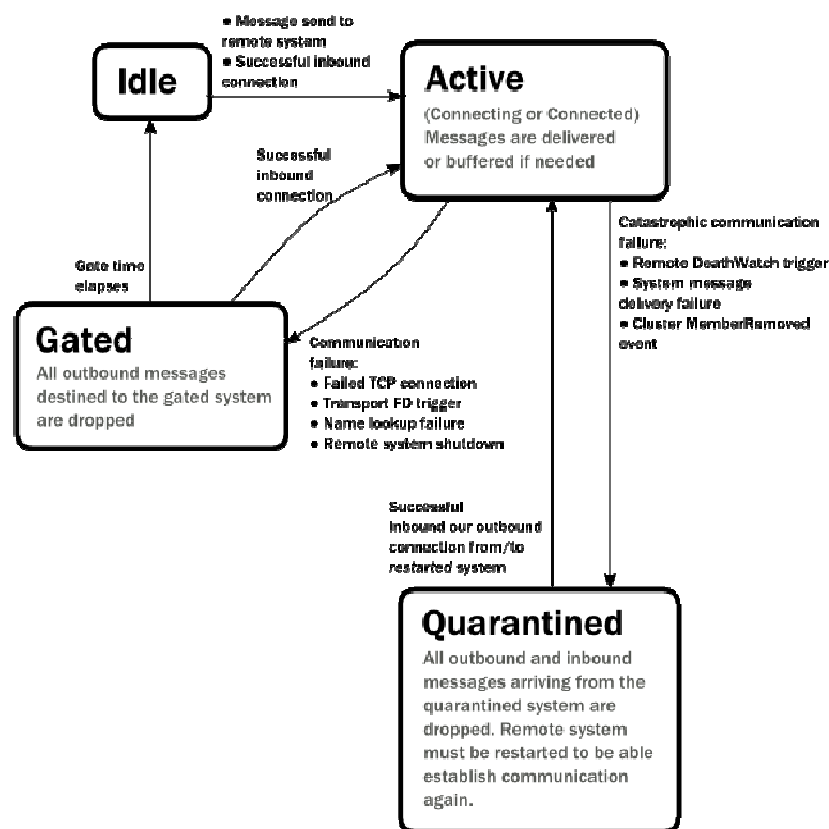
Akka selbst bietet nur wenige Garantien

- 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 Aktorenhierarchie 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.

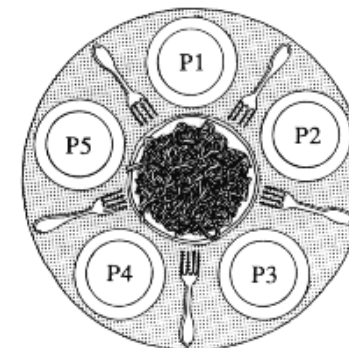


akka Beispiel: Das Philosophenproblem

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

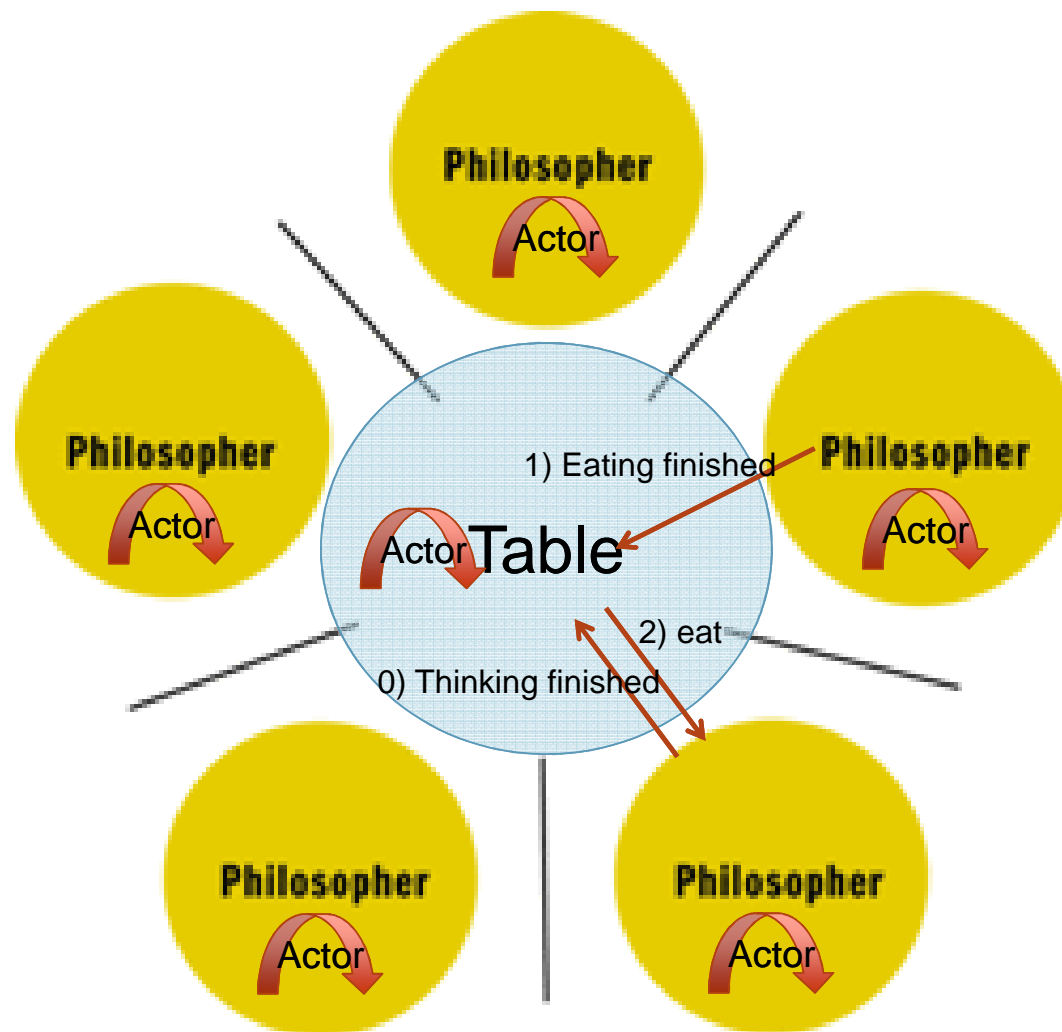
Umsetzung mit akka:

- 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.

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));
        }
    }
}
```

Functional Reactive Programming am Beispiel *RxJava*



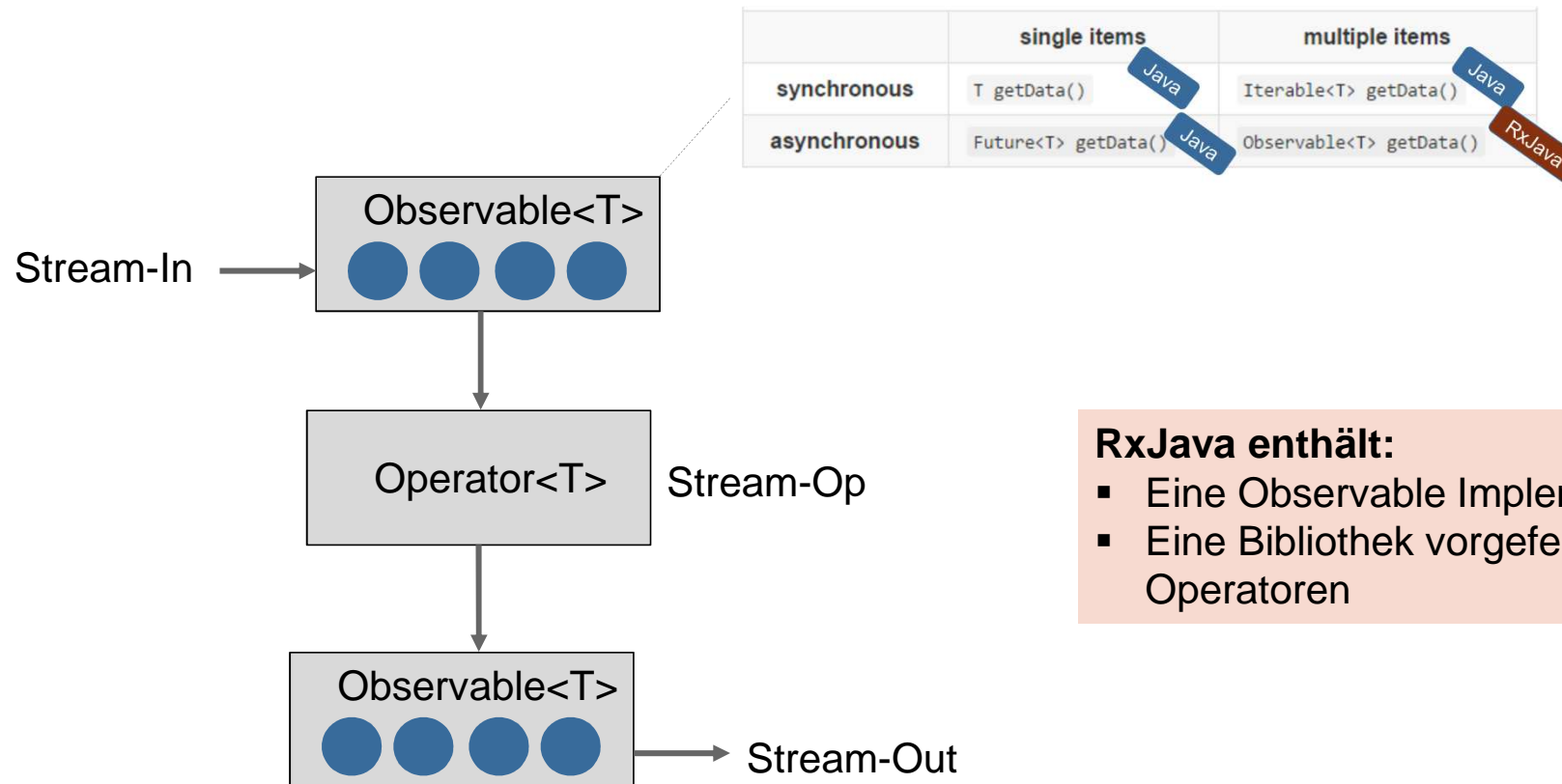
Darf ich vorstellen: RxJava

- Projekt-URL: <https://github.com/ReactiveX/RxJava>
- Open-Source-Projekt unter der Apache 2.0 Lizenz
- Entwickelt seit 2010 durch Netflix
- Portierung der Reactive Extensions (Rx) Bibliothek von .NET auf Java (siehe <https://rx.codeplex.com>)



<https://www.google.de/trends>

Die Grundkonzepte von RxJava



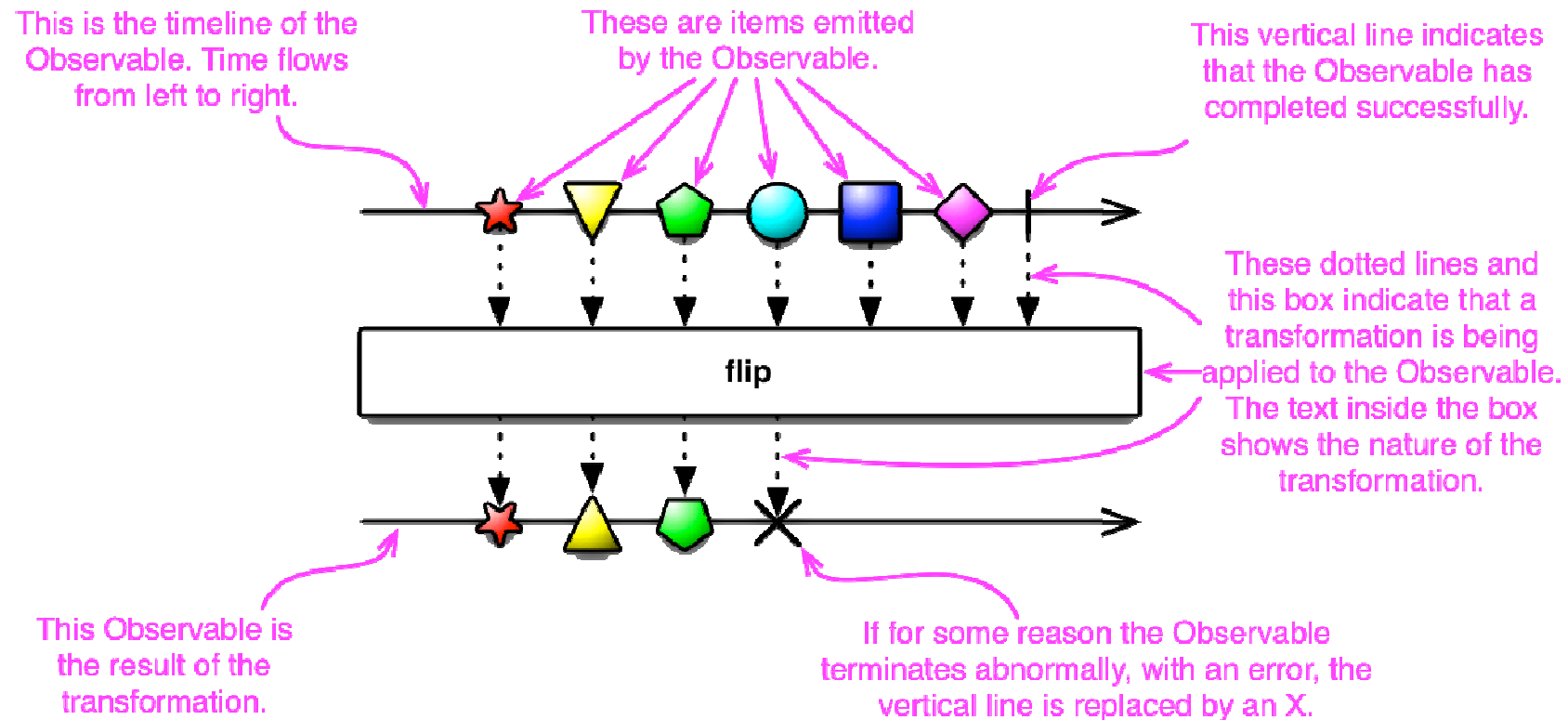
RxJava enthält:

- Eine Observable Implementierung
- Eine Bibliothek vorgefertigter Operatoren

RxJava: Ein einfaches Beispiel mit eigenem Operator.

```
public class Main {  
  
    public static void main(String[] args) { hello("Josef", "Maria", "Hias", "Franz"); }  
  
    public static void hello(String... names) {  
        Observable.from(names).subscribe((Action1) (s) -> { System.out.println("Hallo " + s); });  
    }  
}
```

Die Dokumentation der vorgefertigten Funktionen erfolgt über Marble Diagrams.



Siehe auch: <http://rxwiki.wikidot.com/marble-diagrams>

Die Funktionsbibliothek

- Async
- Blocking Observable
- Combining
- Conditional & Boolean
- Connectable Observable
- Error Handling
- Filtering
- Mathematical and Aggregate
- Observable Creation
- String
- Transformational
- Utility Operators
- Implementing Custom Operators

Filtering Observables

- `filter()` — filter items emitted by an Observable
- `takeLast()` — only emit the last n items emitted by an Observable
- `last()` — emit only the last item emitted by an Observable
- `lastOrDefault()` — emit only the last item emitted by an Observable, or a default value if the source Observable is empty
- `takeLastBuffer()` — emit the last n items emitted by an Observable, as a single list item
- `skip()` — ignore the first n items emitted by an Observable
- `skipLast()` — ignore the last n items emitted by an Observable
- `take()` — emit only the first n items emitted by an Observable
- `first()` **and** `takeFirst()` — emit only the first item emitted by an Observable, or the first item that meets some condition
- `firstOrDefault()` — emit only the first item emitted by an Observable, or the first item that meets some condition, or a default value if the source Observable is empty
- `elementAt()` — emit item n emitted by the source Observable
- `elementAtOrDefault()` — emit item n emitted by the source Observable, or a default item if the source Observable emits fewer than n items
- `sample()` **or** `throttleLast()` — emit the most recent items emitted by an Observable within periodic time intervals
- `throttleFirst()` — emit the first items emitted by an Observable within periodic time intervals
- `throttleWithTimeout()` **or** `debounce()` — only emit an item from the source Observable after a particular timespan has passed without the Observable emitting any other items
- `timeout()` — emit items from a source Observable, but issue an exception if no item is emitted in a specified timespan
- `distinct()` — suppress duplicate items emitted by the source Observable
- `distinctUntilChanged()` — suppress duplicate consecutive items emitted by the source Observable
- `ofType()` — emit only those items from the source Observable that are of a particular class
- `ignoreElements()` — discard the items emitted by the source Observable and only pass through the error or completed notification

Combining Observables

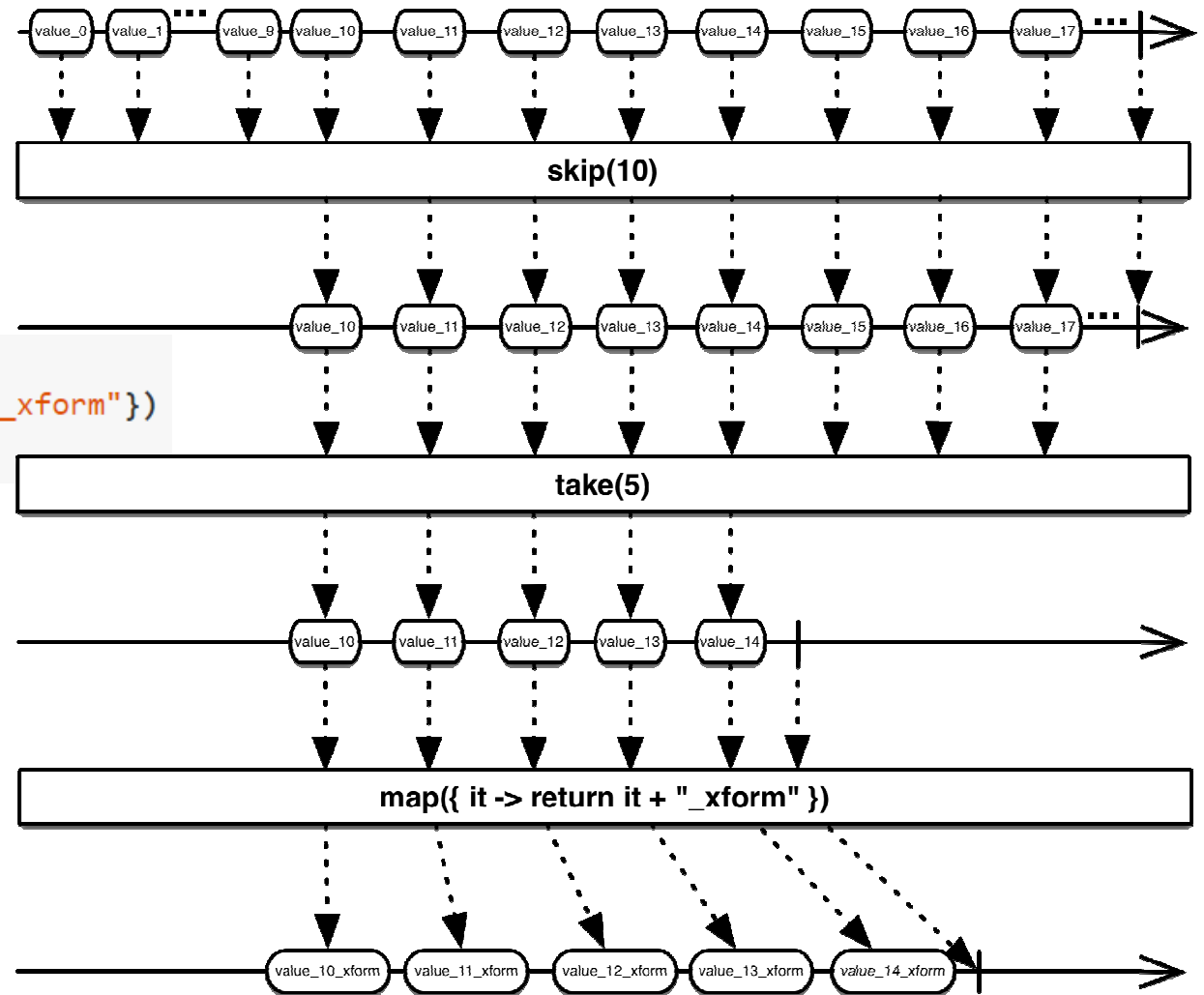
- `startWith()` — emit a specified sequence of items before beginning to emit the items from the Observable
- `merge()` — combine multiple Observables into one
- `mergeDelayError()` — combine multiple Observables into one, allowing error-free Observables to continue before propagating errors
- `zip()` — combine sets of items emitted by two or more Observables together via a specified function and emit items based on the results of this function
- `(rxjava-joins)` `and()`, `then()`, **and** `when()` — combine sets of items emitted by two or more Observables by means of `Pattern` and `Plan` intermediaries
- `combineLatest()` — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
- `join()` **and** `groupJoin()` — combine the items emitted by two Observables whenever one item from one Observable falls within a window of duration specified by an item emitted by the other Observable
- `switchOnNext()` — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently emitted of those Observables

`(rxjava-joins)` — indicates that this operator is currently part of the optional `rxjava-joins` package under `rxjava-contrib` and is not included with the standard RxJava set of operators

...

RxJava: Ein komplexeres Beispiel mit den vorgefertigten Operatoren.

```
customObservableNonBlocking().skip(10).take(5)  
    .map({ stringValue -> return stringValue + "_xform"})  
    .subscribe({ println "onNext => " + it})
```



akka vs. RxJava

akka vs. RxJava

	akka	RxJava
Funktionen		
Funktionale Bausteine	ja	ja, inkl. Bibliothek fertiger Bausteine
Idempotente Logik	Implementierungssache	Implementierungssache
Nachrichten		
Asynchron	ja	ja
Mailboxen	ja	ja
Einziges Sync-Mittel	ja	ja
Kanäle		
Kanalportabilität	ja	nein
Geringer Overhead	ja	ja
Load Balancing	ja	nein
Zuverlässigkeit	ja	nein
Circuit Breaker	ja	nein

Literatur

Links & Literatur

■ Functional Reactive Programming

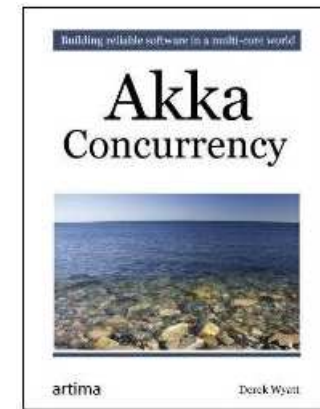
- <https://speakerdeck.com/cmeiklejohn/functional-reactive-programming>
- <https://speakerdeck.com/mnxfst/reactive-programming-on-example-of-the-basar-platform>
- <https://speakerdeck.com/peschlowp/reactive-programming>

■ 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>

■ RxJava

- <https://github.com/ReactiveX/RxJava/wiki>
- <https://speakerdeck.com/duarten/reactive-programming-with-rxjava-1>
- <https://speakerdeck.com/benjchristensen/reactive-programming-with-rx-at-nytimes>
- <https://speakerdeck.com/daschl/reactive-programming-with-rxjava-for-efficient-data-access>
- <https://speakerdeck.com/benjchristensen/reactive-streams-with-rx-at-javaone-2014>
- <http://www.grahamlea.com/2014/07/rxjava-threading-examples>
- <http://www.reactive-streams.org>



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