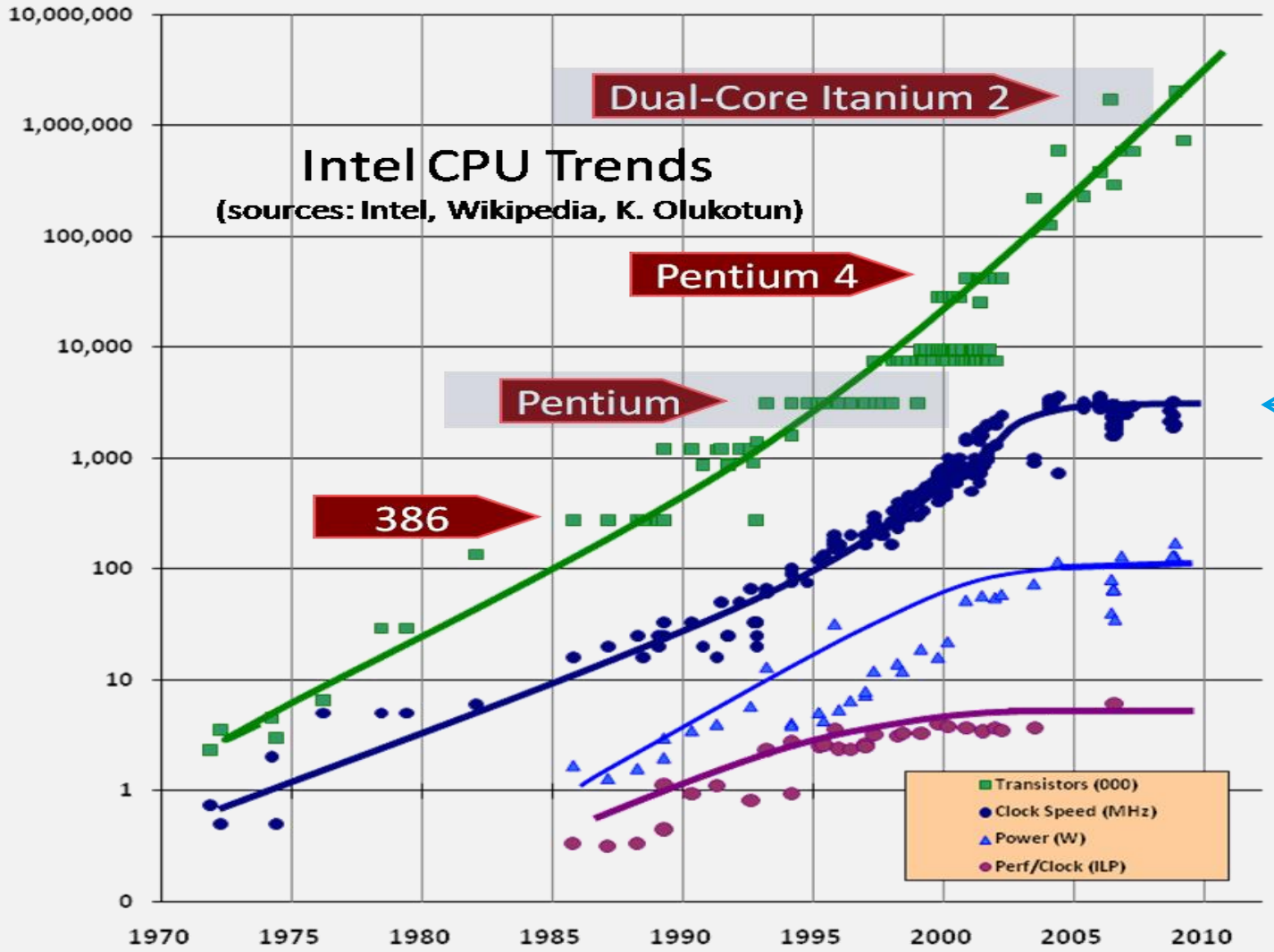
QA|WARE
SOFTWARE ENGINEERING

Programmiermodelle

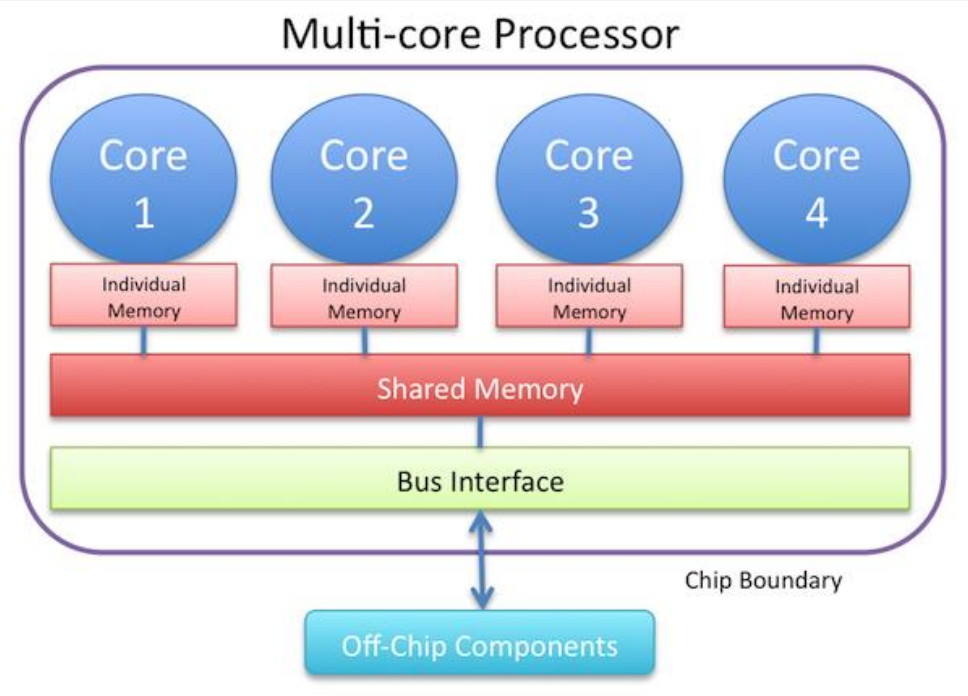
„The free lunch is over“: Es gibt keine kostenlose Performanzsteigerung mehr – Nebenläufigkeit zählt.



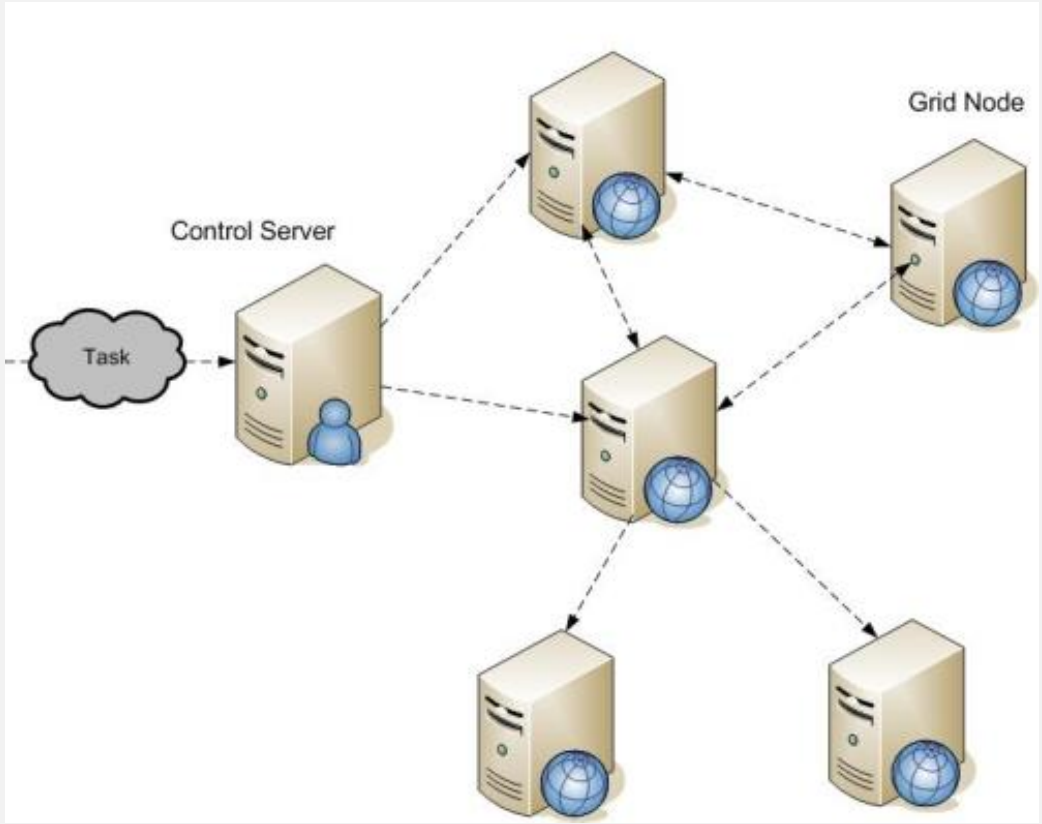
Anzahl Transistoren
Moore‘s Law gilt weiterhin

Taktfrequenz
Seit 2004 ist die Taktfrequenz von CPUs konstant

Nebenläufigkeit kann im Kleinen und im Großen betrieben werden.

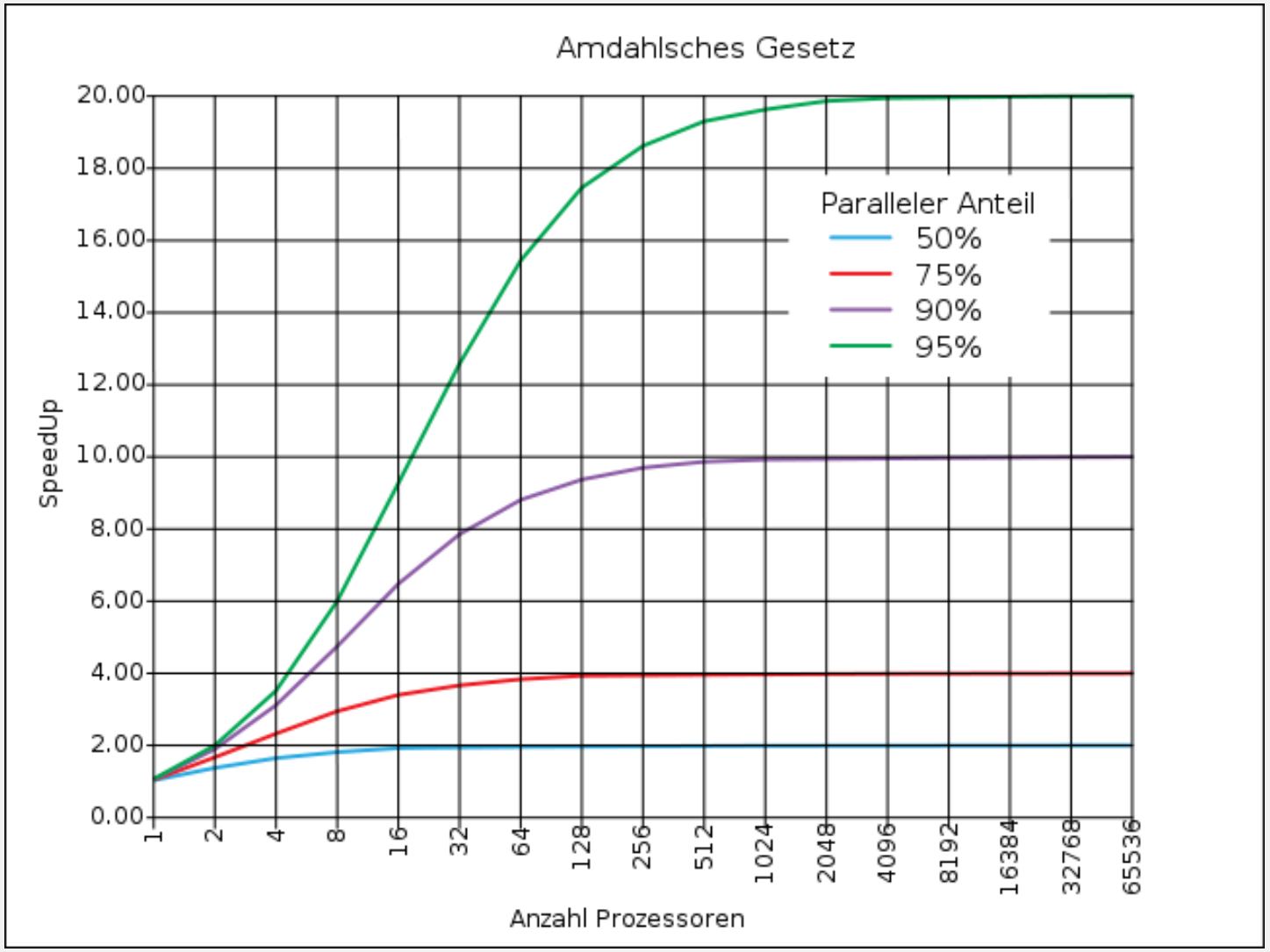


Multi Core



Multi Node
(Cluster, Grid, Cloud)

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



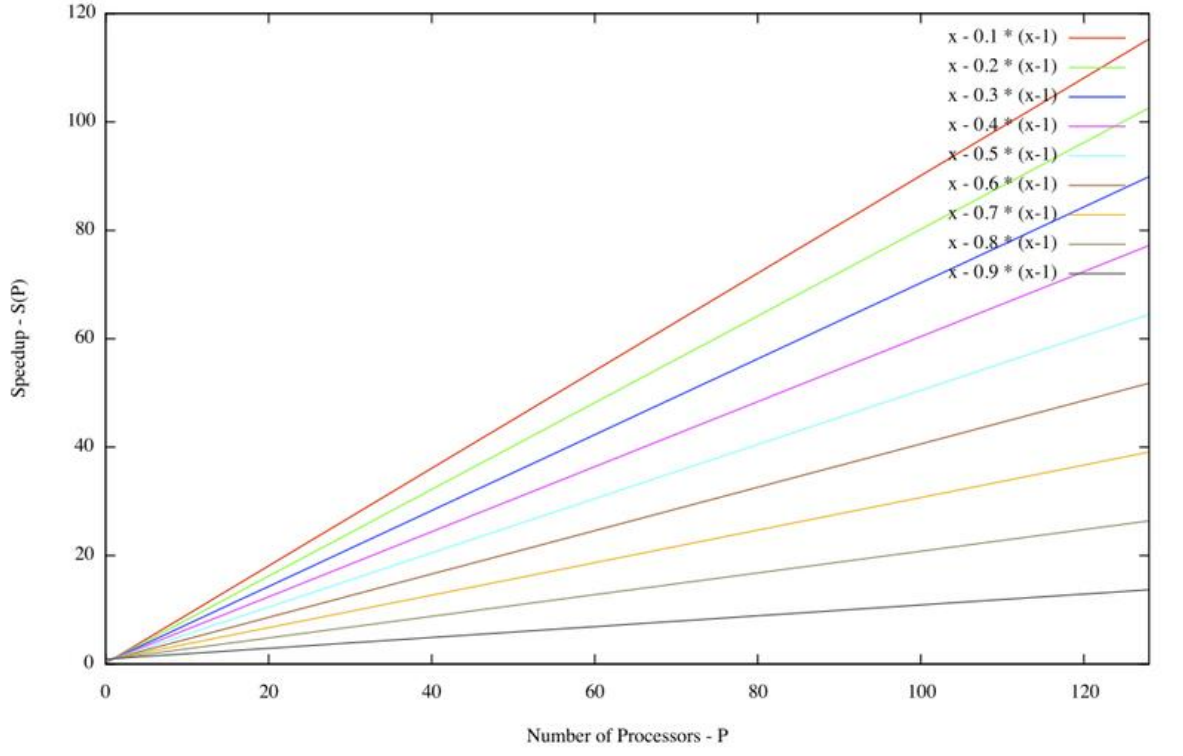
P = Paralleler Anteil (0,0 – 1,0)
S = Sequenzieller Anteil (0,0 – 1,0)
N = Anzahl der Prozessoren (1 ..)∞
Speedup = Maximale Beschleunigung

$$Speedup = \frac{1}{1-P}$$

für N = ∞

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

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



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

- **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

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

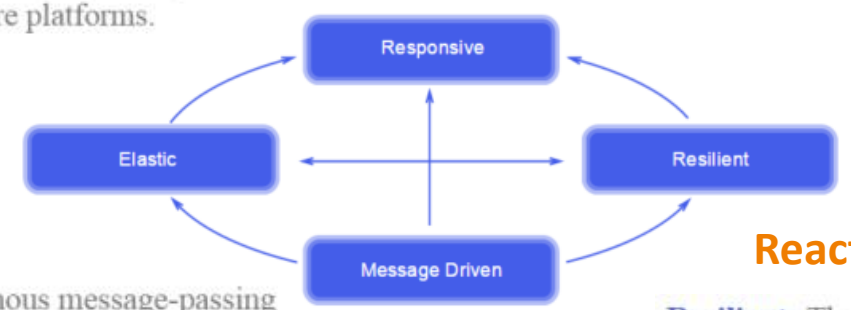
Das Reactive Manifesto

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.

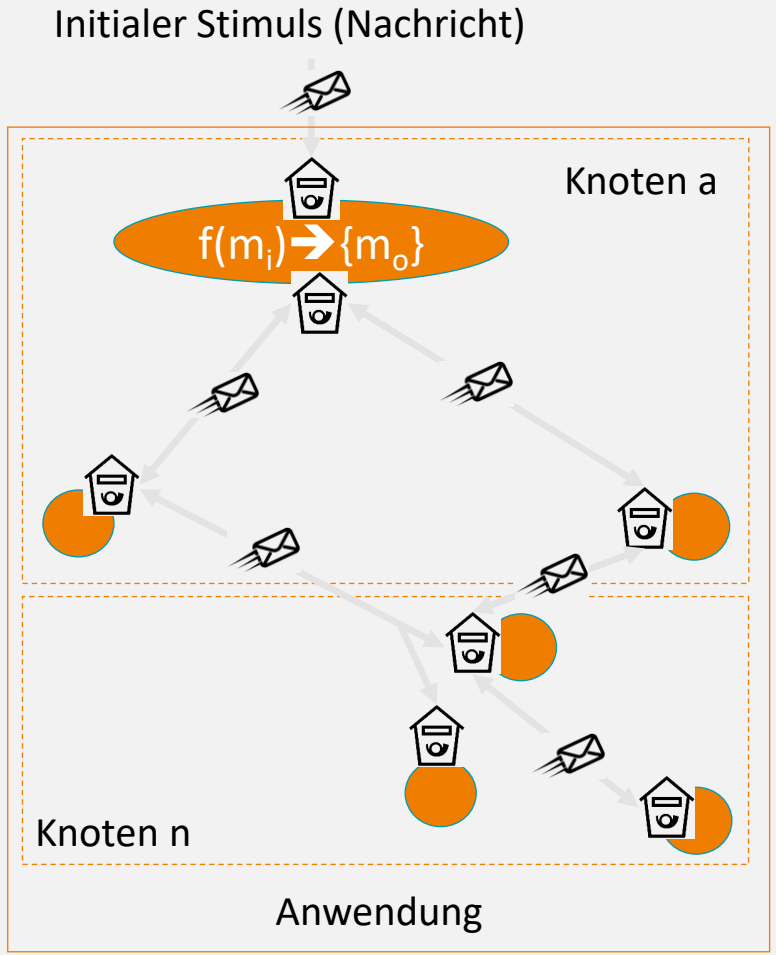
React to failures

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. (I)

Dekomposition in Funktionen (auch Aktoren genannt)

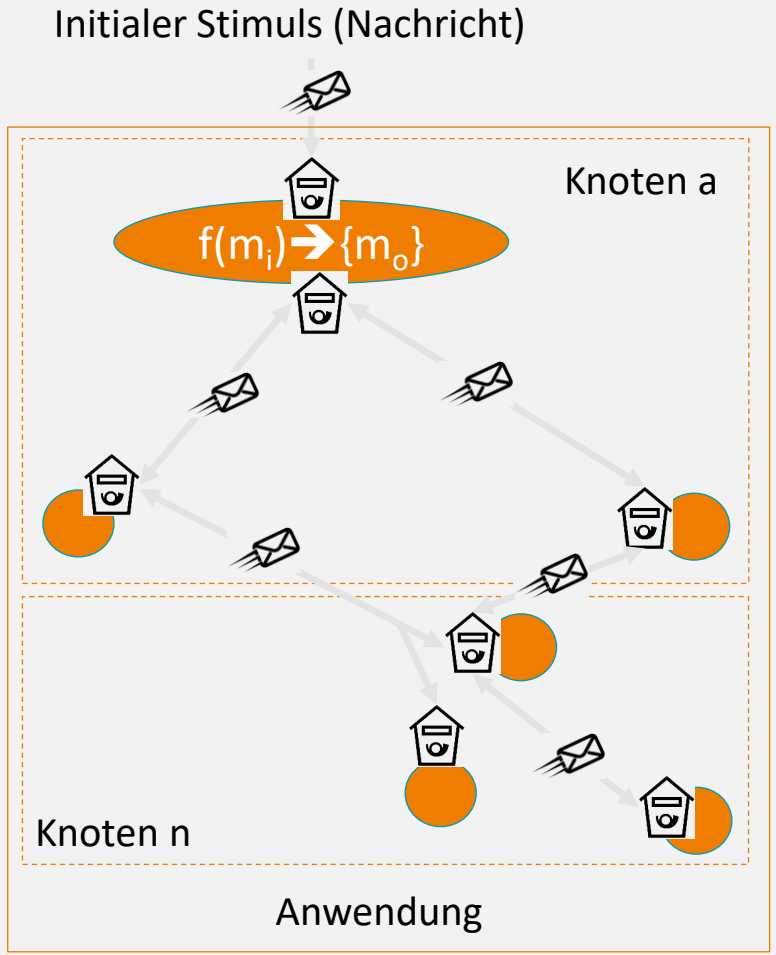
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wiederaufsetzbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)



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

Kommunikation zwischen den Funktionen über Nachrichten

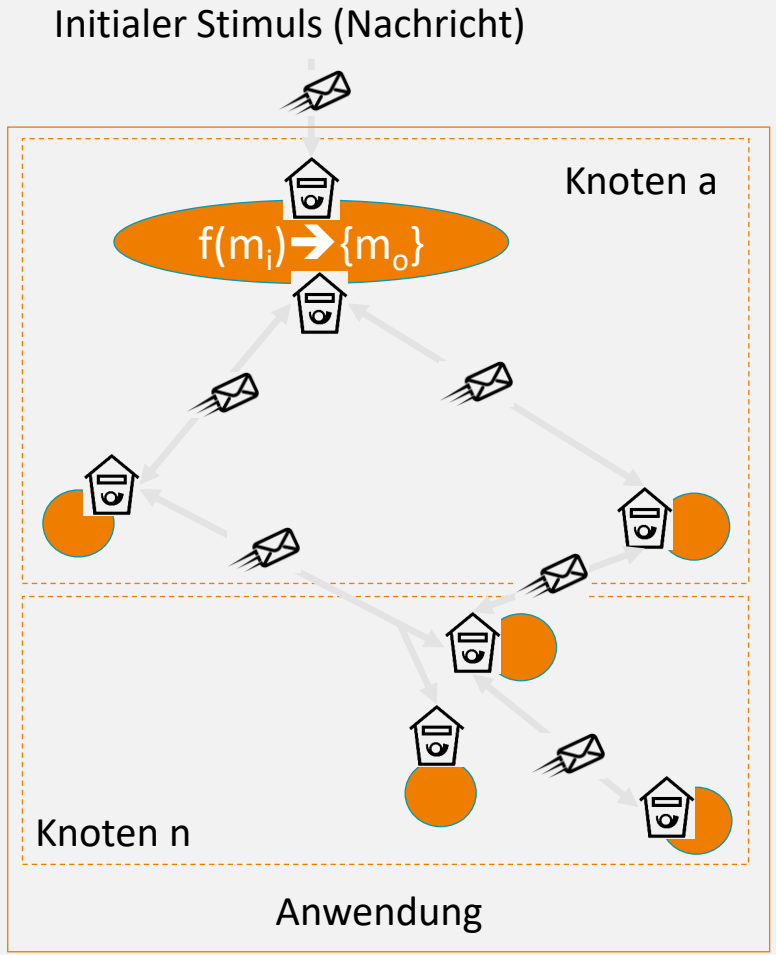
- asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber nicht auf sie.
- Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Producern und 1 Consumer)
- Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und sind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



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

Elastischer Kommunikationskanal

- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden (mehr oder minder) zuverlässig zugestellt (Garantie: „at-most-once“)
- 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

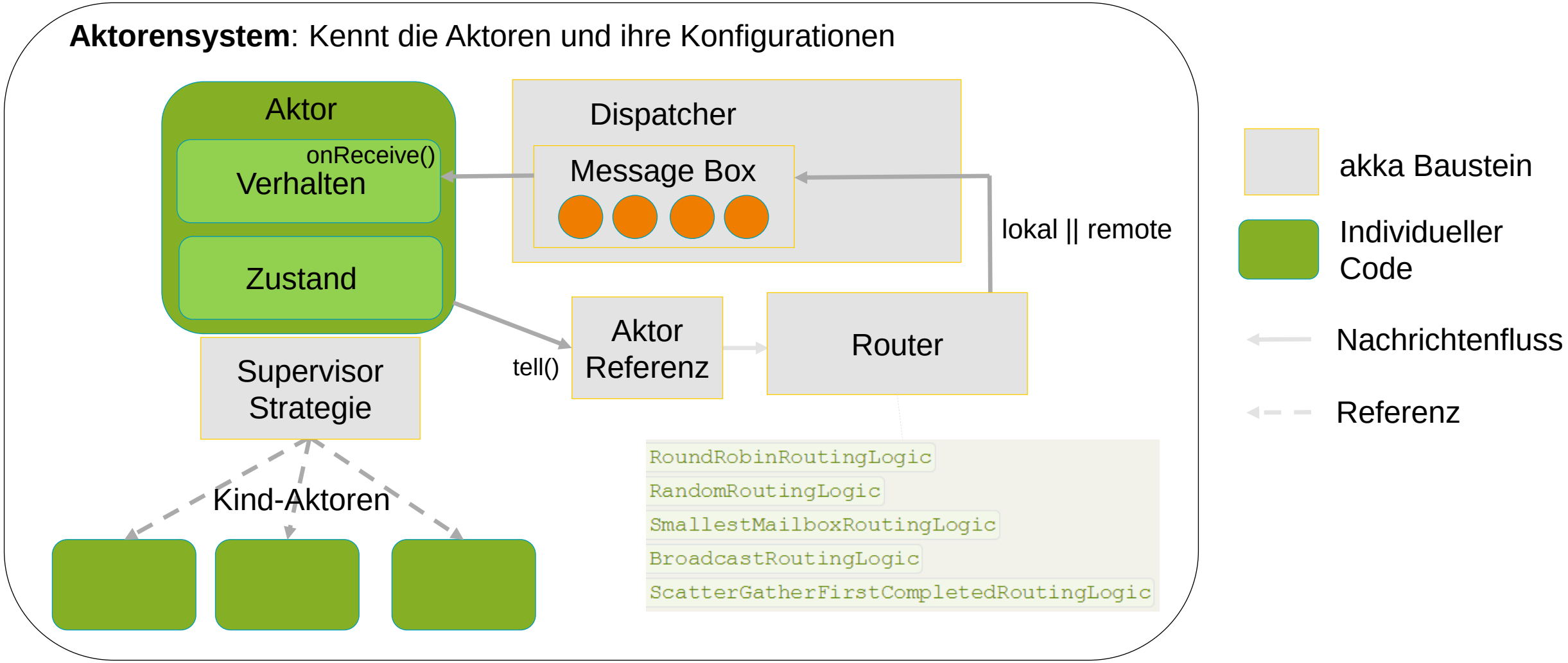
- funktionierender nebenläufiger,
- elastisch skalierbarer
- und selbst-heilender fehlertoleranter Software.

http://akka.io

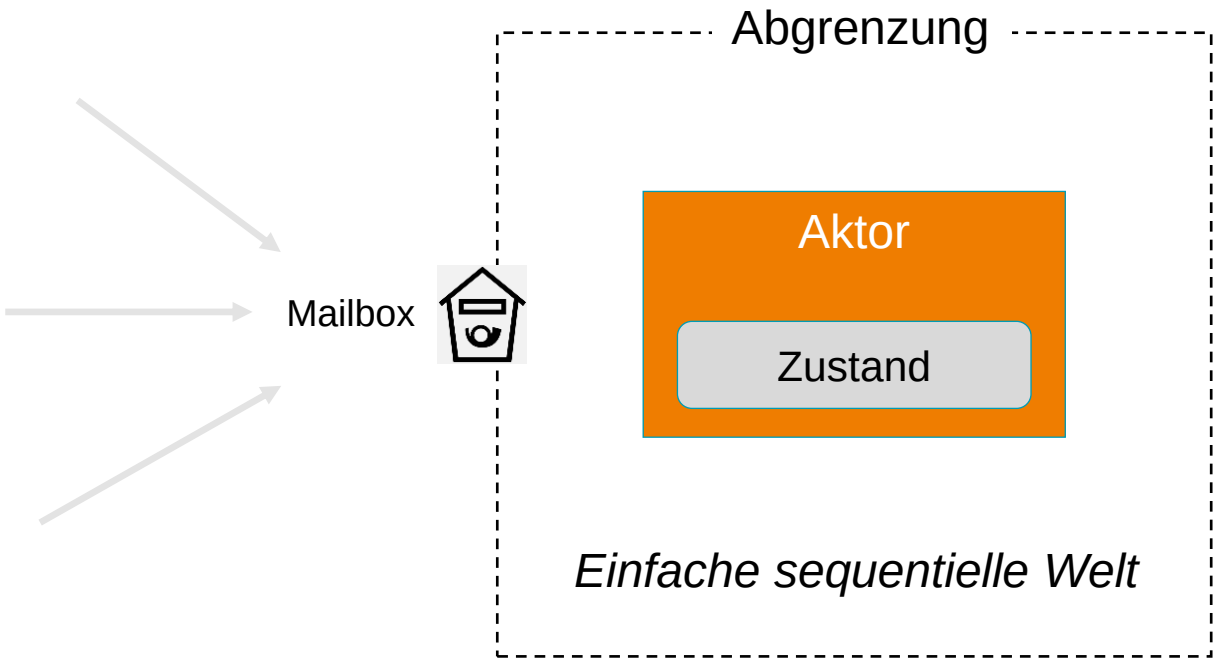
Start der Entwicklung 2009 durch Jonas Bonér im Umfeld Scala inspiriert durch das Aktor-Modell der Programmiersprache Erlang.

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.

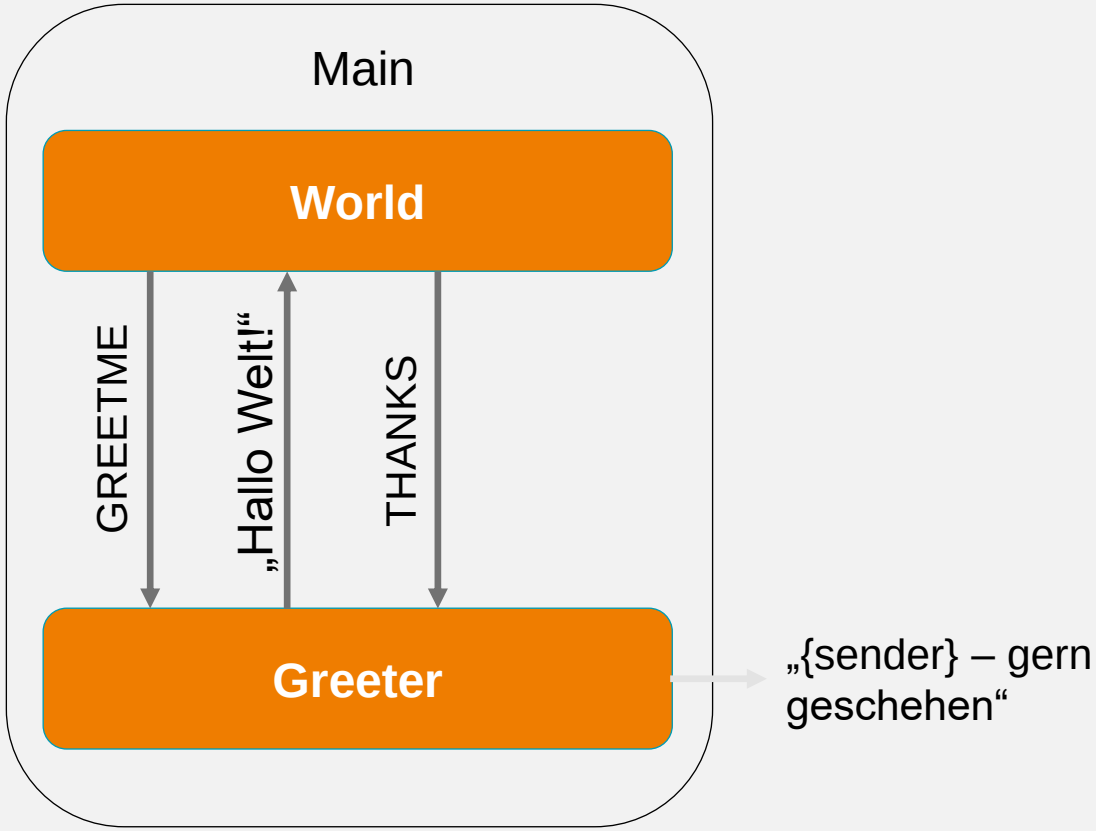
Eine Million Aktoren? Kein Problem!

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()}); }
}
```