Parallele Programmierung
Actor Model

Vorlesung 9

Mirko Stocker
Prof. Dr. Luc Bläser

Agenda

- Inhalt
  - Motivation
  - Actor Modell und CSP
  - Akka Actors
    - Senden und Empfangen
    - Verteilung
    - Supervision

Lernziele

- Prinzipien des Actor Modells verstehen
- Verstehen, welche Probleme sich mit Actors lösen lassen
- Lokale und verteilte Actor Systeme mit Akka implementieren

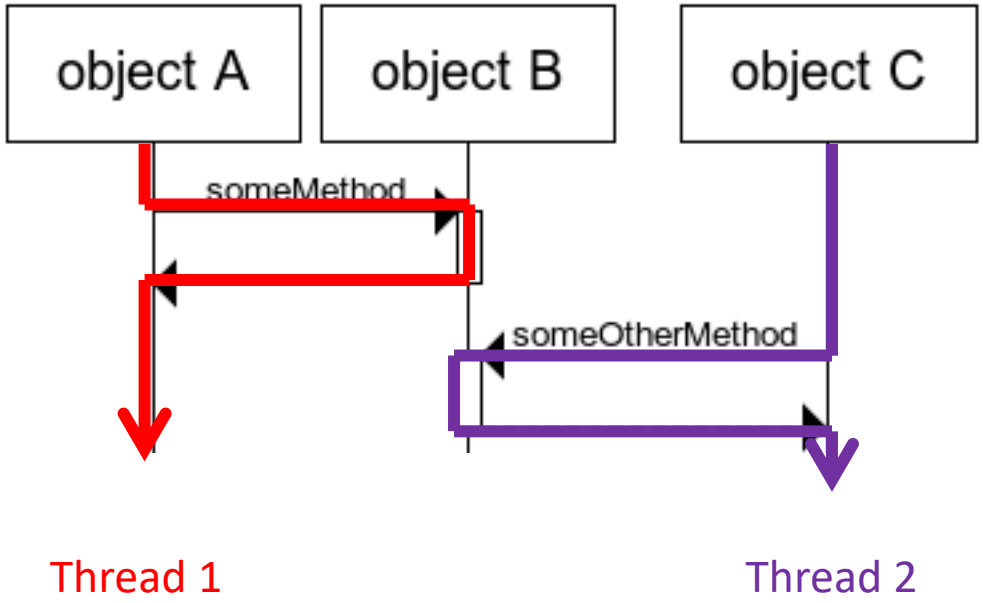
Motivation

- Herkömmliche Programmiersprachen sind nicht für Nebenläufigkeit entworfen
  - Optimiert für sequentielle / prozedurale Ausführung
  - Nebenläufigkeit / Threads sind «Second Class» Features
  - Oder gar nicht vorhanden: JavaScript
  - By Default: Speicher nicht thread-safe

- **Korrekte** nebenläufige Programme zu schreiben ist daher besonders schwierig!

Motivation

Threads operieren auf Modell von **passiven** Objekten



Konsequenzen

- Beschränkter Grad an Nebenläufigkeit
  - Concurrency sehr selektiv eingesetzt
  - Maschinenorientierte Concurrency (für Performance)
  - Viele unnötig blockierende Aufrufe
  - Resultat: «No Free Lunch»
- Fehleranfälligkeit
  - Race Conditions immanent
- Sehr schlecht verteilbar
  - Shared Memory Modell
  - Threads operieren auf gemeinsamen Speicher

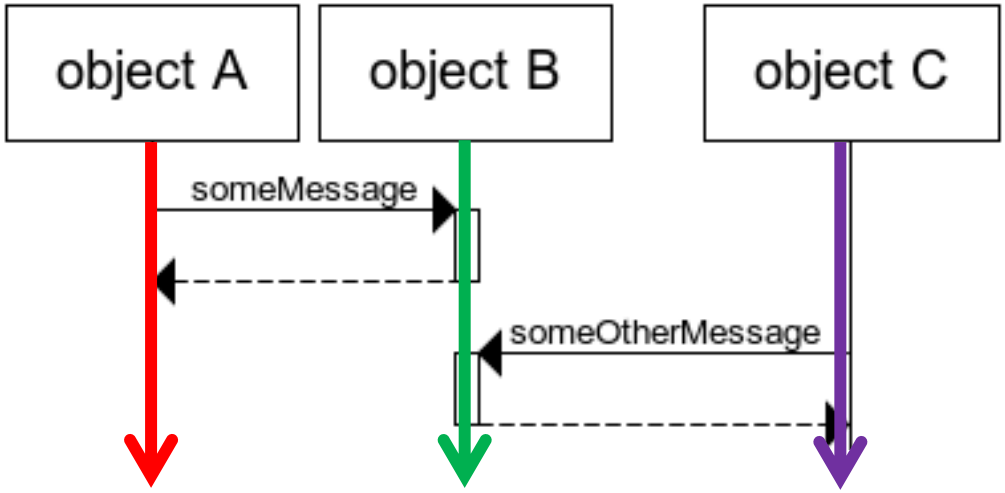
Actor Model

Substantiell anderes Programmierkonzept

- **Aktive** Objekte
  - Objekte haben nebenläufiges Innenleben
- Kommunikation
  - Objekte senden und empfangen Nachrichten
- Kein Shared Memory
  - Nur Austausch von Nachrichten über Kanäle/Mailboxen

Actor Model

Im Actor Modell sind die Objekte (Actors) **aktiv**



Historisches: Simula I

- Simula I (1965)
  - Geburt von OO in Norwegen von O.J. Dahl, K. Nygaard
  - Hatten schon aktive Objekte (ohne Kommunikation, nur Prozeduraufrufe)
  - Später in Simula 67 leider nicht mehr (nur passive Objekte)

Historisches: Actor Model

- Actor Model (1973)
  - Theorie begründet von C. Hewitt
  - http://bit.ly/hewitt-on-actors
- Ein Actor umfasst
  - Processing
  - Storage
  - Communication
- Ein Actor kann
  1. Neue Actors erstellen
  2. Nachrichten an Actors (andere und sich selber) senden
  3. Entscheiden, wie die nächste Nachricht behandelt werden soll (Zustandsänderung)

Historisches: CSP

- Communicating Sequential Processes (CSP)
  - von Sir C.A.R. Hoare im 1978
  - Prozesse kommunizieren indirekt über Channels miteinander
  - Austausch der Nachrichten erfolgt unmittelbar und synchron
  - In Occam (für Transputer) und Ada implementiert, inspirierte die Entwicklung von Go
- Prinzipiell gleiches Modell wie Actors
  - Unterschied: Actor hat keine Channels, Senden ist immer asynchron, keine garantierte Reihenfolge des Empfangs

CSP Modell

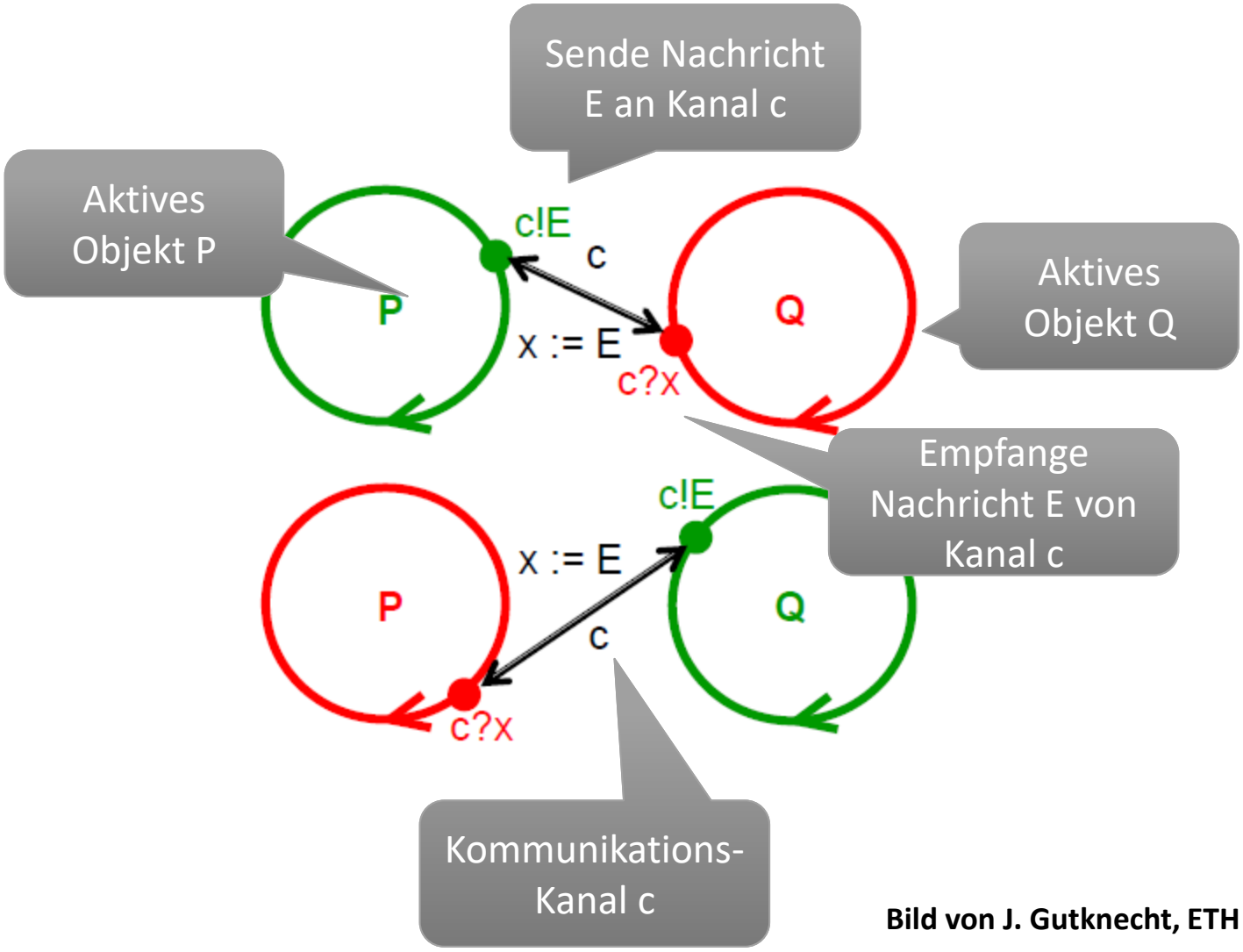


Bild von J. Gutknecht, ETH Zürich

Klassisch versus CSP

```
class A {
  void process(B b) {
      produce x;
      b.process(x);
   }
}
```

```
class B {
  …
   void process(x) {
      consume x;
   }
}
```


```
process A channel c {
  produce x;
  c!process(x)
}
```

```
process B channel c {

  c?process(x);
  consume x
}
```

B kennt A nicht!

Vorteile: Actor, CSP

- Inhärente Nebenläufigkeit
  - Alle Objekte (Actors) laufen nebenläufig
  - Maschine kann Grad an Nebenläufigkeit ausnutzen
- Keine Race Conditions
  - Kein Shared Memory
  - Nachrichtenaustausch synchronisiert implizit
- Gute Verteilbarkeit
  - Kein Shared Memory
  - Nachrichtenaustausch für Netz prädestiniert

Actor Implementierungen

- Erlang (1987)
  - Entwickelt von Joe Armstrong bei Ericsson für Telekommunikation (Telefonie-Switches)
  - Eigenes Laufzeitsystem, Lightweight Threads
- Akka
  - Für die JVM: Java, Scala, JRuby, etc.
  - Meta-Programmiermodell auf «herkömmlicher Sprache»
- Java Communicating Sequential Processes
  - JCSP von Ukent, veraltet
- .NET
  - Concurrency and Coordination Runtime CCR (low-level)
  - Orleans (Microsoft Research, für Xbox eingesetzt)
  - Akka.NET Portierung des Akka-Sourcecodes nach C#

Akka

Implementiert in Scala

- zusätzliches Java-API

Zur Entwicklung

- nebenläufiger,
- verteilter,
- fehlertoleranter und
- event-basierter Systeme

Neben Erlang wohl das verbreitetste Actor-Framework

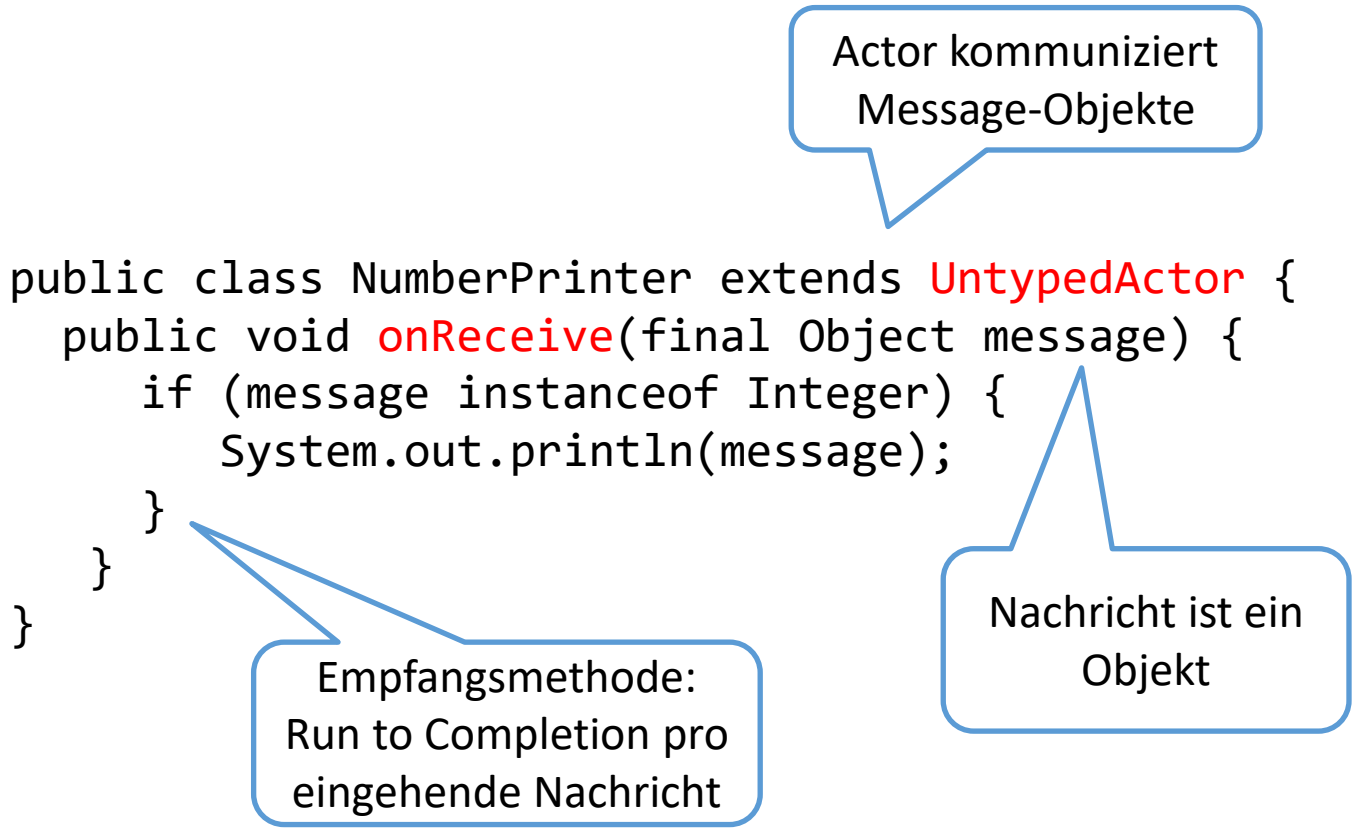
Akkas Actor Konzept

- Actor sind aktive Objekte
  - Laufen **konzeptionell** nebenläufig zueinander
- Privater Zustand
  - Kann auf der JVM nicht enforced werden; aufpassen, dass per Java Referenzen kein Shared State entsteht
- Eine Mailbox pro Actor
  - Ein Buffer für alle ankommenden Nachrichten
  - Asynchrones Senden

Akka Empfangsverhalten

- Reaktion auf ankommende Nachricht
  - Spezielle Behandlungs-Methode wird ausgeführt
- Effekte per Behandlung
  - Ändere privater Zustand
  - Sende Nachrichten
  - Erzeuge neue Actors
- Intern sequentiell
  - Nur eine Nachricht auf einmal bedienbar

Einfacher Akka Actor



Erzeugen und Senden

Spezielle Referenz

```
ActorSystem system = ActorSystem.create("System");

ActorRef printer = system.actorOf(Props.create(NumberPrinter.class));

for (int i = 0; i < 100; i++) {
  printer.tell(i, ActorRef.noSender());
}

system.shutdown();
```

Erzeugung per Reflection

Einfaches asynchrones Senden

Gebe «End-Signal» an alle Actors

Wieso diese komplizierte Erzeugung und ActorRef?

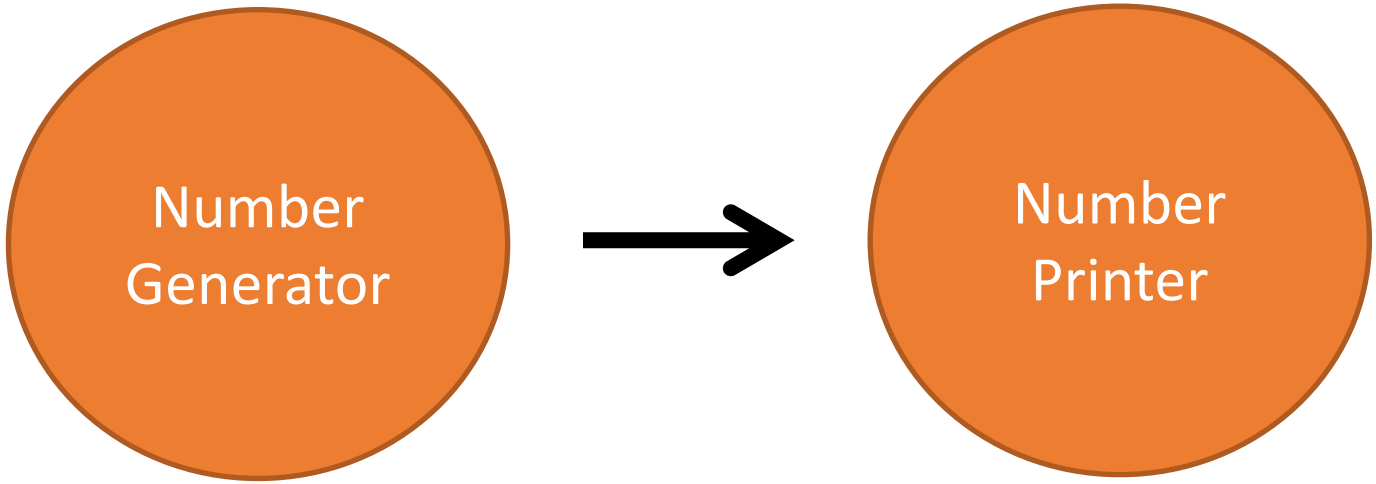
Actor-Referenzen

- `ActorRef`: Adresse eines Actors
  - Actor bei Fehlverhalten neu startbar (behält Adresse)
- Entkopplung von «Interface» und Instanz
  - Vorbereitet für Verteilung
  - ActorRef ist immutable (in Message verschickbar)
- Verhindert Methodenaufrufe/Variablenzugriffe
  - Reiner Nachrichtenaustausch

Beispiele für Actor Anwendungen

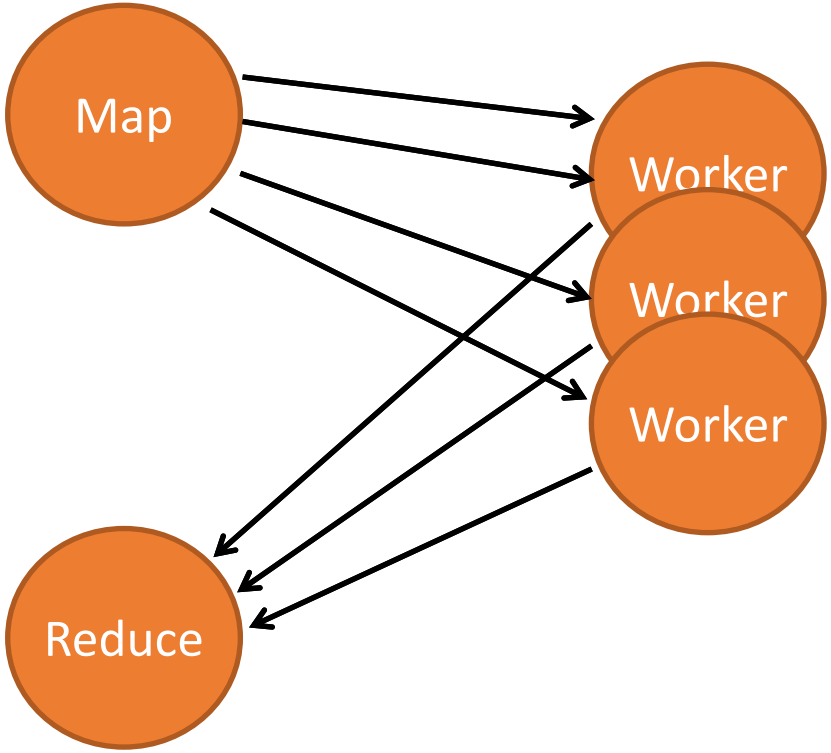
- Alternative zu Threads
  - Explizite Kommunikation statt Shared Memory
- Transaction-Processing
  - Koordination über verschiedene Systeme
- Backend für Service
  - REST, SOAP, Websockets
- Kommunikations-Hub
  - Router, Load-Balancer

Pattern: Producer-Consumer



Pattern: Map-Reduce

Aufteilen der Arbeit an Worker

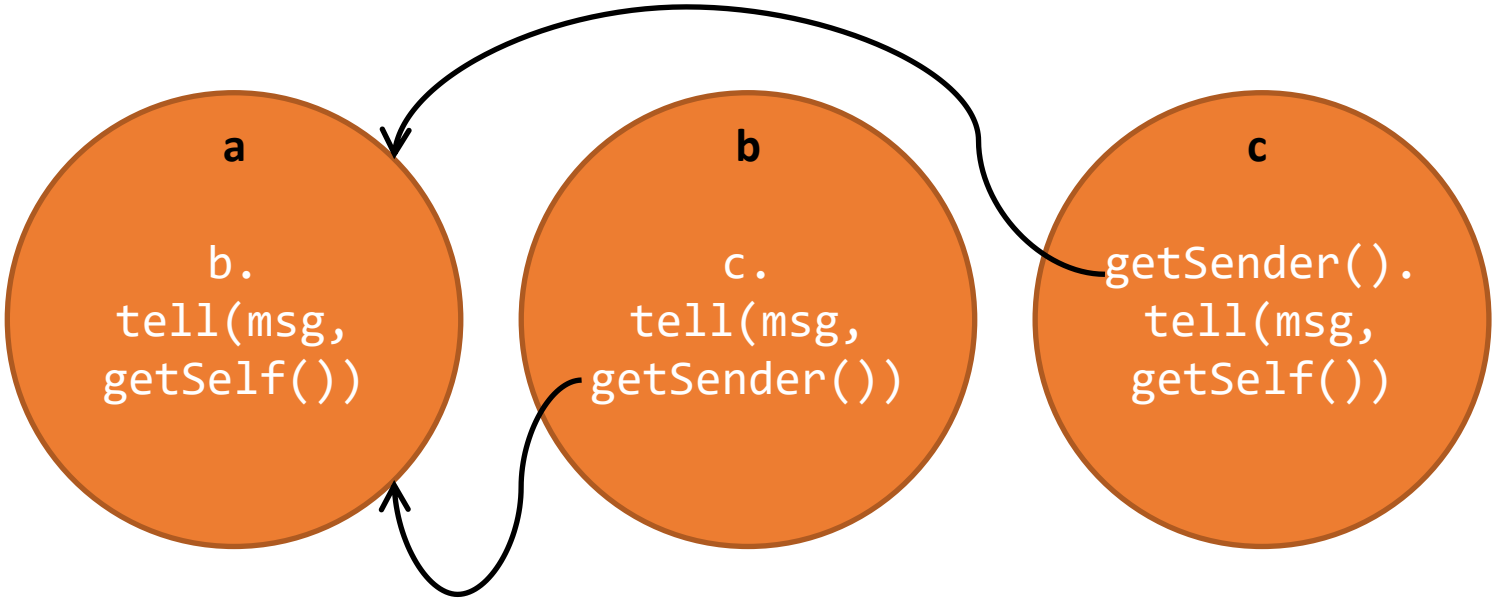


Resultate einsammeln und zusammenfügen

Akka Sender

- `tell(msg, sender)`
  - Sender der Message wird mitgegeben
  - Nützlich bei Antwort an Sender-Actor
  - Typischerweise `getSelf()`, bzw. `getSender()` bei Forward

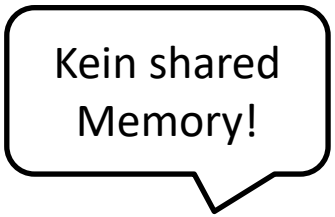
Methoden von Actor



Verteilbarkeit

```
printer.tell(i);
```

```
NumberPrinter ... {
  onReceive(Object i) {
    ...
  }
}
```



Senden und Empfangen von **serialisierten immutable** Nachrichten

Verteilte Actors – Server

Genau gleich

```
ActorSystem system = ActorSystem.create("System");

ActorRef printer = system.actorOf(Props.create(...), "printer");
```

Actor Name

application.conf

```
akka {
  actor {
     provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
     enabled-transports = ["akka.remote.netty.tcp"]
     netty.tcp {
         hostname = "server"
         port = 2552
     }
  }
}
```

Verteilte Actors – Client

```
ActorSystem system = ActorSystem.create("producer");
```

ActorSelection, nicht ActorRef

```
ActorSelection printer = system.actorSelection(
  "akka.tcp://System@server:2552/user/printer");
```

Actor Adresse

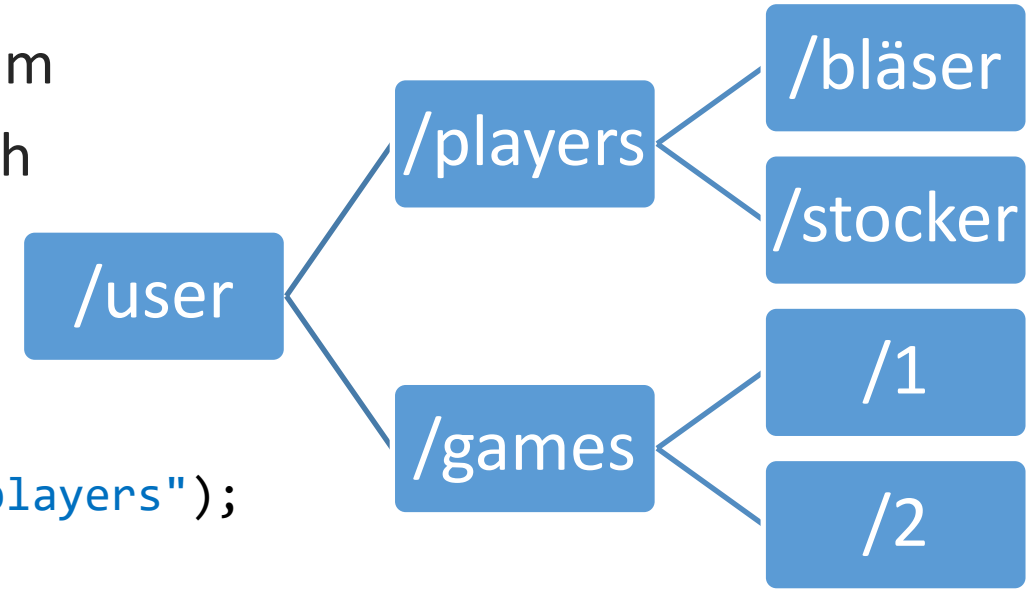
```
printer.tell(123, ActorRef.noSender());
```

Actor Remoting

- Remote Lookup
  - `system.actorSelection` mit URL
- `ActorSelection`
  - Leichtgewichtiger als ActorRef
  - Kann 0..n Actors umfassen
  - Kann zu ActorRef aufgelöst werden
- Remote Erzeugen
  - `system.actorOf(...)`
  - `application.conf` spezifiziert, wo Actor erstellt wird
  - Keine Codeänderungen

Actor Hierachies

- Hierarchie von Actors
  - Passend zu URL Adressierungsschema
  - Supervision von Actors
  - Erzeuger ist Parent
- `ActorSelection`
  - Selektiert Teilbaum
  - Broadcast möglich




```
actorSelection("/user/players");
```

System Shutdown

- Wann kann ein Actor System beendet werden?
  - Alle Mailboxen leer? Actor könnte noch beschäftigt sein
- Applikation muss Actors selber stoppen

```
getContext().stop(actorRef);
getContext().stop(getSelf());
getContext().system().terminate();

actor.tell(PoisonPill.getInstance(), sender);
```

Stoppt nach Bearbeitung der akuellen Message

immer Rekursiv

Stoppt bei Behandlung der Poison Pill

Rückblick: Lernziele

- Prinzipien des Actor Modells verstehen
- Verstehen, welche Probleme sich mit Actors lösen lassen
- Lokale und verteilte Actor Systeme mit Akka implementieren

Literatur

- Akka – https://akka.io
- Buch Programming Concurrency on the JVM , Chapter 8
- C. A. R. Hoare: Communicating Sequential Processes
  - https://dl.acm.org/citation.cfm?id=359585
- Hewitt, Meijer and Szyperski: The Actor Model
  - https://bit.ly/hewitt-on-actors