

Reaktive Programmierung
Vorlesung 15 vom 29.06.15: Robustheit und Entwurfsmuster

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Fahrplan

- ▶ Einführung
- ▶ Monaden als Berechnungsmuster
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Functional Reactive Programming
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ **Robustheit und Entwurfsmuster**
- ▶ Theorie der Nebenläufigkeit, Abschluss

Rückblick: Konsistenz

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Strong Eventual Consistency
 - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
 - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Conflict-Free replicated Data Types:
 - ▶ Zustandsbasiert: CvRDTs
 - ▶ Operationsbasiert: CmRDTs
- ▶ Operational Transformation
 - ▶ Strong Eventual Consistency auch ohne kommutative Operationen

Robustheit in verteilten Systemen

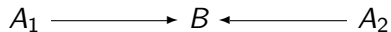
Lokal:

- ▶ Nachrichten gehen nicht verloren
- ▶ Aktoren können abstürzen - Lösung: Supervisor

Verteilt:

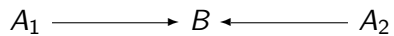
- ▶ Nachrichten können verloren gehen
- ▶ Teilsysteme können abstürzen
 - ▶ Hardware-Fehler
 - ▶ Stromausfall
 - ▶ Geplanter Reboot (Updates)
 - ▶ Naturkatastrophen / Höhere Gewalt
 - ▶ Software-Fehler

Zwei-Armeen-Problem

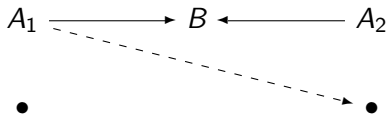


- ▶ Zwei Armeen A_1 und A_2 sind jeweils zu klein um gegen den Feind B zu gewinnen.
- ▶ Daher wollen sie sich über einen Angriffszeitpunkt absprechen.

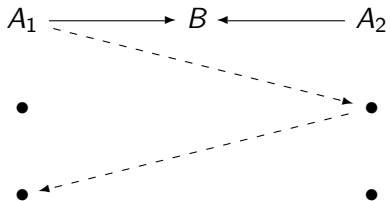
Zwei-Armeen-Problem



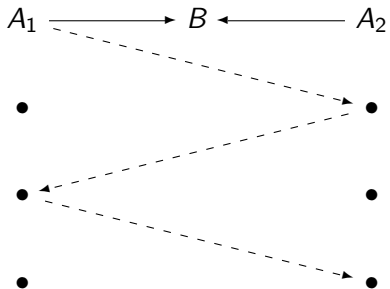
Zwei-Armeen-Problem



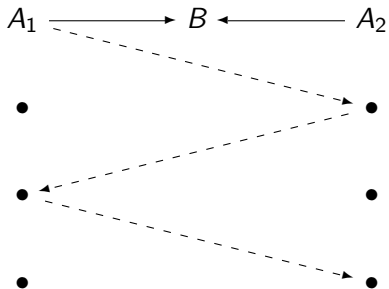
Zwei-Armeen-Problem



Zwei-Armeen-Problem



Zwei-Armeen-Problem



- ▶ Unlösbar – Wir müssen damit leben!

Unsichere Kanäle

- ▶ Unsichere Kanäle sind ein generelles Problem der Netzwerktechnik
- ▶ Lösungsstrategien:
 - ▶ Redundanz – Nachrichten mehrfach schicken
 - ▶ Indizierung – Nachrichten numerieren
 - ▶ Timeouts – Nicht ewig auf Antwort warten
 - ▶ Heartbeats – Regelmäßige „Lebenszeichen“
- ▶ Beispiel: TCP
 - ▶ Drei-Wege Handschlag
 - ▶ Indizierte Pakete

Gossiping

N_1

N_2

N_3

N_4

N_5

N_6

N_7

N_8

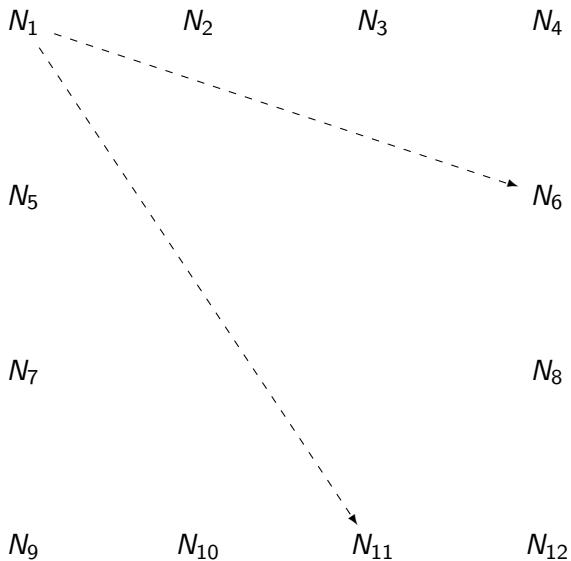
N_9

N_{10}

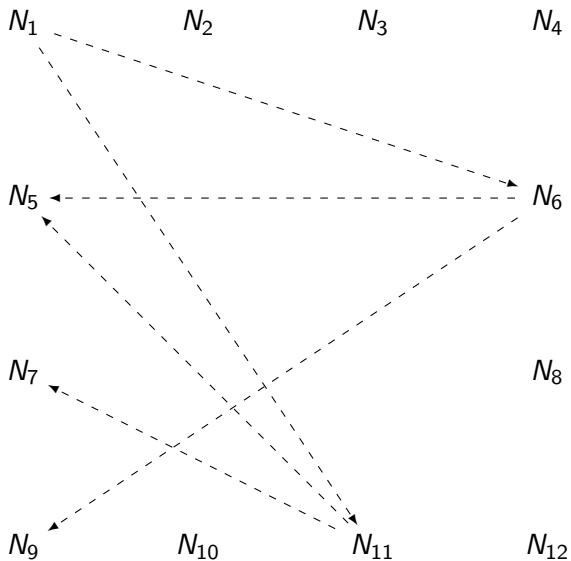
N_{11}

N_{12}

Gossiping



Gossiping

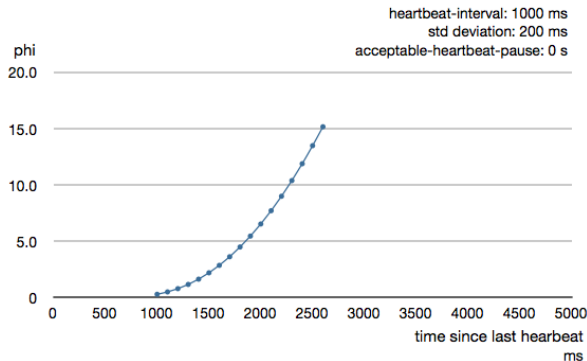


Gossiping

- ▶ Jeder Knoten verbreitet Informationen periodisch weiter an zufällige weitere Knoten
- ▶ Funktioniert besonders gut mit CvRDTs
 - ▶ Nachrichtenverlust unkritisch
- ▶ Anwendungen
 - ▶ Ereignis-Verteilung
 - ▶ Datenabgleich
 - ▶ Anti-entropy Protokolle
 - ▶ Aggregate, Suche

Heartbeats

- ▶ Kleine Nachrichten in regelmäßigen Abständen
- ▶ Standardabweichung kann dynamisch berechnet werden
- ▶ $\Phi = -\log_{10}(1 - F(\text{timeSinceLastHeartbeat}))$



Akka Clustering

- ▶ Verteiltes Aktorsystem
 - ▶ Infrastruktur wird über gossiping Protokoll geteilt
 - ▶ Ausfälle werden über Heartbeats erkannt
- ▶ **Sharding**: Horizontale Verteilung der Ressourcen
 - ▶ In Verbindung mit Gossiping mächtig

(Anti-)Patterns: Request/Response

- ▶ Problem: Warten auf eine Antwort — Benötigt einen Kontext der die Antwort versteht
- ▶ Pragmatische Lösung: Ask-Pattern

```
import akka.patterns.ask

(otherActor ? Request) map {
  case Response => //
}
```

- ▶ Eignet sich nur für sehr einfache Szenarien
- ▶ Lösung: Neuer Aktor für jeden Response Kontext

(Anti-)Patterns: Nachrichten

- ▶ Nachrichten sollten **typisiert** sein

```
otherActor ! "add 5 to your local state" // NO
otherActor ! Modify(_ + 5) // YES
```

- ▶ Nachrichten dürfen **nicht** veränderlich sein!

```
val state: scala.collection.mutable.Buffer
otherActor ! Include(state) // NO
otherActor ! Include(state.toList) // YES
```

- ▶ Nachrichten dürfen **keine Referenzen** auf veränderlichen Zustand enthalten

```
var state = 7
otherActor ! Modify(_ + state) // NO
val stateCopy = state
otherActor ! Modify(_ + stateCopy) // YES
```

(Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

(Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

- ▶ Besser?

```
(otherActor ? Request) map { case Response =>
  state ≠ 1; RequestComplete
} pipeTo sender
```

(Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

- ▶ Besser?

```
(otherActor ? Request) map { case Response =>
  state += 1; RequestComplete
} pipeTo sender
```

- ▶ So geht's!

```
(otherActor ? Request) map { case Response =>
  self ! IncState
  RequestComplete
} pipeTo sender
```

(Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

(Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

- ▶ Welche Strategie bei DivByZeroException?

(Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

- ▶ Welche Strategie bei `DivByZeroException`?
- ▶ Ein Aktor sollte immer nur **eine** Aufgabe haben!

(Anti-)Patterns: Akteur-Beziehungen

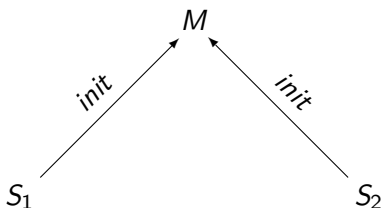
M

S₁

S₂

- ▶ Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?

(Anti-)Patterns: Aktor-Beziehungen



- ▶ Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?
- ▶ Slaves sollten sich beim Master registrieren!
 - ▶ Flexibel / Dynamisch
 - ▶ Einfachere Konfiguration in verteilten Systemen

(Anti-)Patterns: Aufgabenverteilung

- ▶ Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?

(Anti-)Patterns: Aufgabenverteilung

- ▶ Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?
- ▶ **Wichtige** Informationen und zentrale Aufgaben sollten möglichst nah an der Wurzel sein.
- ▶ **Gefährliche** bzw. unsichere Aufgaben sollten immer Kindern übertragen werden.

(Anti-)Patterns: Zustandsfreie Aktoren

- ▶ Ein Aktor ohne Zustand

```
class Calculator extends Actor {  
  def receive = {  
    case Divide(x,y) => sender ! Result(x / y)  
  }  
}
```

(Anti-)Patterns: Zustandsfreie Aktoren

- ▶ Ein Aktor ohne Zustand

```
class Calculator extends Actor {  
  def receive = {  
    case Divide(x,y) => sender ! Result(x / y)  
  }  
}
```

- ▶ Ein Fall für Käpt'n Future!

```
class UsesCalculator extends Actor {  
  def receive = {  
    case Calculate(Divide(x,y)) =>  
      Future(x/y) pipeTo self  
    case Result(x) =>  
      println("Got it: " + x)  
  }  
}
```

(Anti-)Pattern: Initialisierung

- ▶ Problem: Akteur benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann

(Anti-)Pattern: Initialisierung

- ▶ Problem: Akteur benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann
- ▶ Lösung: Parametrisierter Zustand

```
class Robot extends Actor {  
  def receive = uninitialized  
  def uninitialized: Receive = {  
    case Init(pos,power) =>  
      context.become(initialized(pos,power))  
  }  
  def initialized(pos: Point, power: Int): Receive = {  
    case Move(North) =>  
      context.become(initialized(pos + (0,1), power - 1))  
  }  
}
```

(Anti-)Patterns: Kontrollnachrichten

- ▶ Problem: Akteur mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich

(Anti-)Patterns: Kontrollnachrichten

- ▶ Problem: Akteur mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich
- ▶ Lösung: Verkettete partielle Funktionen

```
class Obstacle extends Actor {  
  def rejectMoveTo: Receive = {  
    case MoveTo => sender ! Reject  
  }  
  def receive = uninitialized orElse rejectMoveTo  
  def uninitialized: Receive = ...  
  def initialized: Receive = ...  
}
```

(Anti-)Patterns: Circuit Breaker

- ▶ Problem: Wir haben eine elastische, reaktive Anwendung aber nicht genug Geld um eine unbegrenzt große Server Farm zu betreiben.
- ▶ Lösung: Bei Überlastung sollten Anfragen nicht mehr verarbeitet werden.

```
class DangerousActor extends Actor with ActorLogging {  
  val breaker =  
    new CircuitBreaker(context.system.scheduler,  
      maxFailures = 5,  
      callTimeout = 10.seconds,  
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())  
  
  def notifyMeOnOpen(): Unit =  
    log.warning("My CircuitBreaker is now open, and will  
      not close for one minute")  
}
```

(Anti)-Patterns: Message Transformer

```
class MessageTransformer(from: ActorRef, to: ActorRef,  
  transform: PartialFunction[Any,Any]) extends Actor {  
  
  def receive = {  
    case m => to forward transform(m)  
  }  
}
```

Weitere Patterns

- ▶ Lange Aufgaben unterteilen
- ▶ Aktor Systeme sparsam erstellen
- ▶ Futures sparsam einsetzen
- ▶ `Await.result()` **nur** bei Interaktion mit Nicht-Aktor-Code
- ▶ Dokumentation Lesen!

Zusammenfassung

- ▶ Nachrichtenaustausch in verteilten Systemen ist unzuverlässig
- ▶ Zwei Armeen Problem
- ▶ Lösungsansätze
 - ▶ Drei-Wege Handschlag
 - ▶ Nachrichtennummerierung
 - ▶ Heartbeats
 - ▶ Gossiping Protokolle
- ▶ Patterns und Anti-Patterns
- ▶ Nächstes mal: Theorie der Nebenläufigkeit