

Reaktive Programmierung
Vorlesung 13 vom 22.07.14: Robustheit, Entwurfsmuster und
Theorie der Nebenläufigkeit

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- ▶ Teil I: Grundlegende Konzepte
- ▶ Teil II: Nebenläufigkeit
- ▶ Teil III: Fortgeschrittene Konzepte
 - ▶ Bidirektionale Programmierung: Zippers and Lenses
 - ▶ Robustheit, Entwurfsmuster und Theorie der Nebenläufigkeit

Robustheit in verteilten Systemen

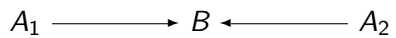
Lokal:

- ▶ Aktoren können abstürzen - Lösung: Supervisor

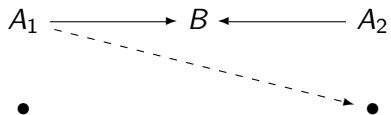
Verteilt:

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

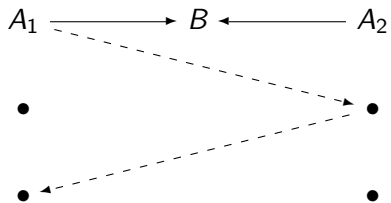
Zwei-Armeen-Problem



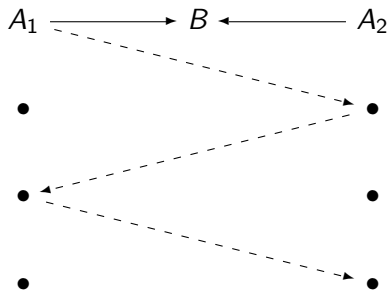
Zwei-Armeen-Problem



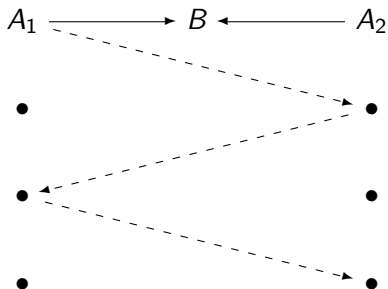
Zwei-Armeen-Problem



Zwei-Armeen-Problem



Zwei-Armeen-Problem



- ▶ Unlösbar – Wir müssen damit leben!
- ▶ TCP: Drei-Wege-Handschlag + Nummerierte Pakete

Gossiping

N_1

N_2

N_3

N_4

N_5

N_6

N_7

N_8

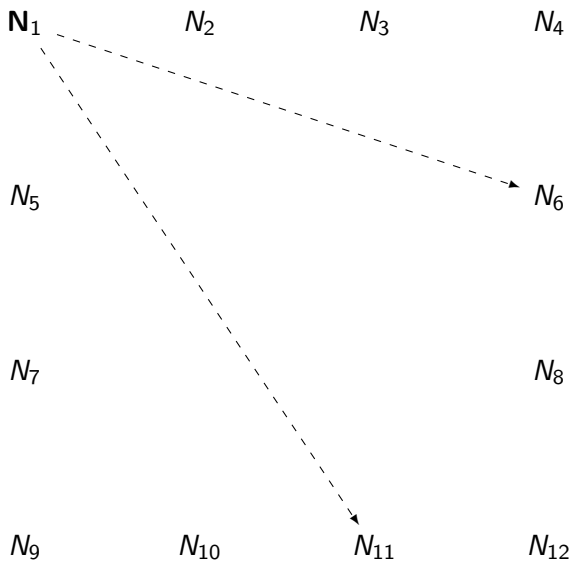
N_9

N_{10}

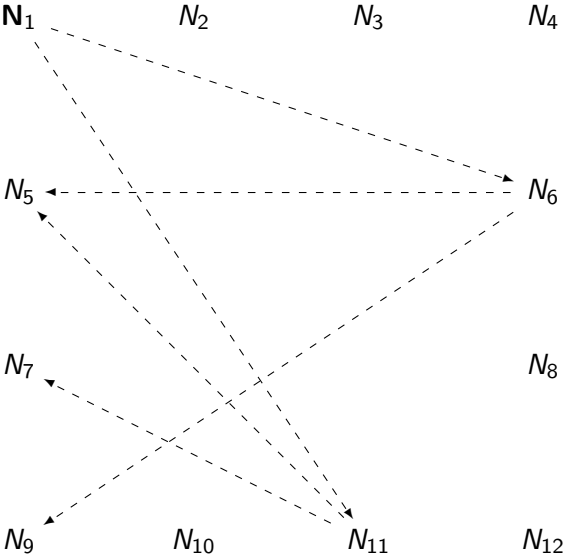
N_{11}

N_{12}

Gossiping

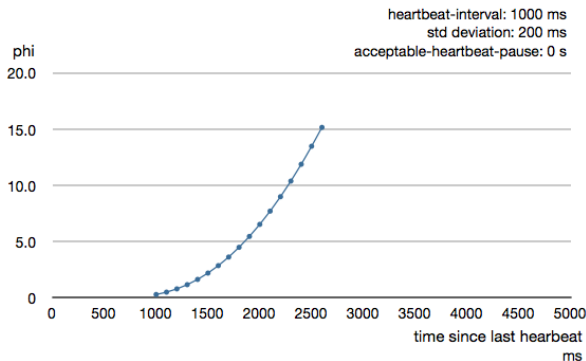


Gossiping



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 Event-Sourcing sehr 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: Aktor-Beziehungen

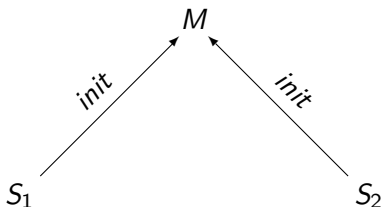
M

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

Patterns: ...

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

Prozessalgebren und Modelchecking

- ▶ Prozessalgebren und temporale Logik beschreiben **Systeme** anhand ihrer **Zustandsübergänge**
- ▶ Ein System ist dabei im wesentlichen eine **endliche Zustandsmaschine** $\mathcal{M} = \langle S, \Sigma, \rightarrow \rangle$ mit Zustandsübergang $\rightarrow \subseteq S \times \Sigma \times S$
- ▶ Prozessalgebren erlauben **mehrere** Zustandsmaschinen und ihre **Synchronisation**
- ▶ Damit modellieren wir **nebenläufige Systeme**
 - ▶ Zur **Spezifikation** (CSP) aber auch als **Berechnungsmodell** (π -Kalkül)
- ▶ Der Trick ist **Abstraktion**: mehrere interne Zustandsübergänge werden zu einem Zustandsübergang zusammengefaßt

Beispiel: ein Flugbuchungssystem

- ▶ Operationen des Servers:
 - ▶ Nimmt Anfragen an, schickt Resultate (mit flid)
 - ▶ Nimmt Buchungsanfragen an, schickt Bestätigung (ok) oder Fehler (fail)
 - ▶ Nimmt Stornierung an, schickt Bestätigung
- ▶ Unterscheidung zwischen **interner** Auswahl \sqcap (Server trifft Entscheidung) und **externer** Auswahl \square (Server reagiert)

$SERVER = query?(from, to) \rightarrow result!flid \rightarrow SERVER$
 $\square booking?flid \rightarrow (ok \rightarrow SERVER \sqcap fail \rightarrow SERVER)$
 $\square cancel?flid \rightarrow ok \rightarrow SERVER$

Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$CLIENT = query!(from, to) \rightarrow result?flid \rightarrow \\ (booking!flid \rightarrow ok \rightarrow CLIENT$$
$$\sqcap CLIENT)$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$CLIENT = query!(from, to) \rightarrow result?flid \rightarrow \\ (booking!flid \rightarrow ok \rightarrow CLIENT$$
$$\sqcap CLIENT)$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

- ▶ Problem: **Deadlock**

- ▶ Es gibt **Werkzeuge** (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden

Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$\begin{aligned} CLIENT &= query!(from, to) \rightarrow result?flid \rightarrow \\ &\quad (booking!flid \rightarrow (ok \rightarrow CLIENT \\ &\quad \quad \square fail \rightarrow CLIENT)) \\ &\quad \square CLIENT) \end{aligned}$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

- ▶ Problem: **Deadlock**

- ▶ Es gibt **Werkzeuge** (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden