

Reaktive Programmierung
Vorlesung 5 vom 26.04.17: Das Aktorenmodell

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

Das Aktorenmodell



- ▶ Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- ▶ Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- ▶ Theoretisches Berechnungsmodell

Das Aktorenmodell



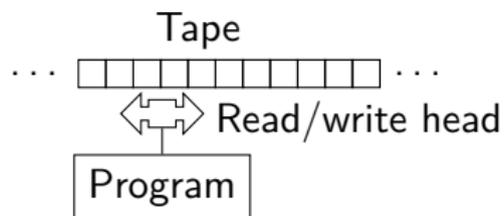
- ▶ Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- ▶ Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- ▶ Theoretisches Berechnungsmodell

Warum ein weiteres Berechnungsmodell? Es gibt doch schon die Turingmaschine!

Die Turingmaschine



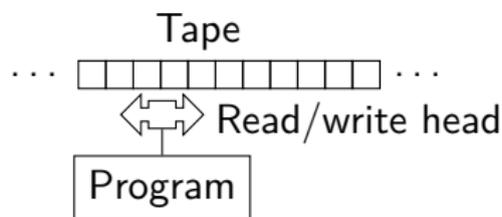
“the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his ‘state of mind’ at that moment”
— Alan Turing



Die Turingmaschine



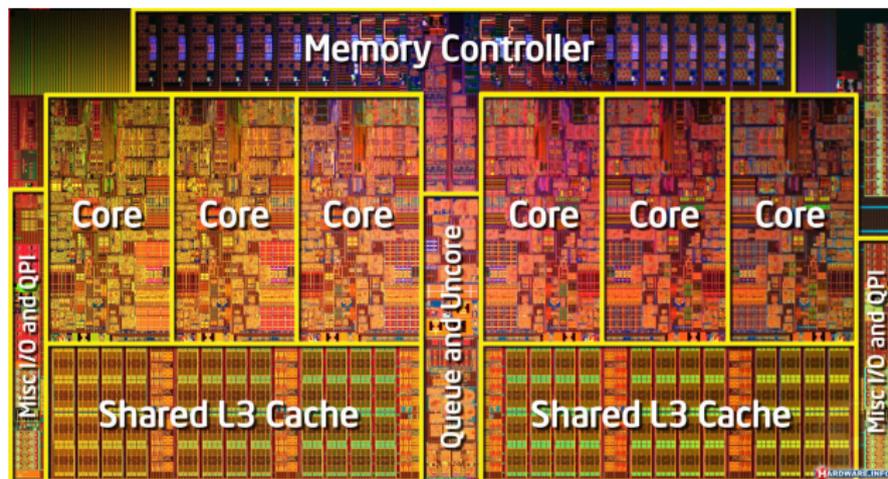
“the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his ‘state of mind’ at that moment”
— Alan Turing



It is “absolutely impossible that anybody who understands the question [What is computation?] and knows Turing’s definition should decide for a different concept.” — Kurt Gödel



Die Realität



- ▶ $3\text{GHz} = 3'000'000'000\text{Hz} \implies \text{Ein Takt} = 3,333 * 10^{-10}\text{s}$
- ▶ $c = 299'792'458 \frac{\text{m}}{\text{s}}$
- ▶ Maximaler Weg in einem Takt $< 0,1\text{m}$ (Physikalische Grenze)

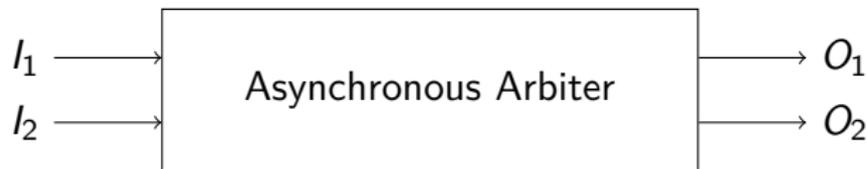
Synchronisation



- ▶ Während auf ein Signal gewartet wird, kann nichts anderes gemacht werden
- ▶ Synchronisation ist nur in engen Grenzen praktikabel! (Flaschenhals)

Der Arbiter

- ▶ Die Lösung: **Asynchrone Arbiter**



- ▶ Wenn I_1 und I_2 fast ($\approx 2fs$) gleichzeitig aktiviert werden, wird entweder O_1 oder O_2 aktiviert.
- ▶ Physikalisch unmöglich in konstanter Zeit. Aber Wahrscheinlichkeit, dass keine Entscheidung getroffen wird nimmt mit der Zeit exponentiell ab.
- ▶ Idealer Arbiter entscheidet in $O(\ln(1/\epsilon))$
- ▶ kommen in modernen Computern überall vor

Unbounded Nondeterminism

- ▶ In Systemen mit Arbitern kann das Ergebnis einer Berechnung **unbegrenzt** verzögert werden,
- ▶ wird aber **garantiert** zurückgegeben.
- ▶ Nicht modellierbar mit (nichtdeterministischen) Turingmaschinen.

Beispiel

Ein Arbitr entscheidet in einer Schleife, ob ein Zähler inkrementiert wird oder der Wert des Zählers als Ergebnis zurückgegeben wird.

Das Aktorenmodell

Quantum mechanics indicates that the notion of a universal description of the state of the world, shared by all observers, is a concept which is physically untenable, on experimental grounds. — Carlo Rovelli

- ▶ Frei nach der relationalen Quantenphysik

Drei Grundlagen

- ▶ Verarbeitung
 - ▶ Speicher
 - ▶ **Kommunikation**
-
- ▶ Die (nichtdeterministische) Turingmaschine ist ein Spezialfall des Aktorenmodells
 - ▶ Ein **Aktorensystem** besteht aus **Aktoren** (Alles ist ein Aktor!)

Aktoren

- ▶ Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- ▶ neue Aktoren erzeugen
 - ▶ Nachrichten an bekannte Aktor-Referenzen versenden
 - ▶ festlegen wie die nächste Nachricht verarbeitet werden soll
-
- ▶ Aktor \neq (Thread | Task | Channel | ...)

Aktoren

- ▶ Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- ▶ neue Aktoren erzeugen
- ▶ Nachrichten an bekannte Aktor-Referenzen versenden
- ▶ festlegen wie die nächste Nachricht verarbeitet werden soll

- ▶ Aktor \neq (Thread | Task | Channel | ...)

Ein Aktor kann (darf) nicht

- ▶ auf globalen Zustand zugreifen
- ▶ veränderliche Nachrichten versenden
- ▶ irgendetwas tun während er keine Nachricht verarbeitet

Aktoren (Technisch)

- ▶ $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$
(Verhalten)

Aktoren (Technisch)

- ▶ $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$
(Verhalten)
- ▶ $\text{Behavior} : (\text{Msg}, \text{State}) \rightarrow \text{IO State}$
- ▶ oder $\text{Behavior} : \text{Msg} \rightarrow \text{IO Behavior}$

Aktoren (Technisch)

- ▶ $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$
(Verhalten)
- ▶ $\text{Behavior} : (\text{Msg}, \text{State}) \rightarrow \text{IO State}$
- ▶ oder $\text{Behavior} : \text{Msg} \rightarrow \text{IO Behavior}$
- ▶ Verhalten hat Seiteneffekte (IO):
 - ▶ Nachrichtenversand
 - ▶ Erstellen von Aktoren
 - ▶ Ausnahmen

Verhalten vs. Protokoll

Verhalten

Das Verhalten eines Aktors ist eine seiteneffektbehaftete Funktion
Behavior : $Msg \rightarrow IO\ Behavior$

Protokoll

Das Protokoll eines Aktors beschreibt, wie ein Aktor auf Nachrichten reagiert und resultiert implizit aus dem Verhalten.

► Beispiel:

```
case (Ping, a) =>  
  println("Hello")  
  counter += 1  
  a ! Pong
```

$\exists a(b, Ping) \mathcal{U} \diamond b(Pong)$

Kommunikation

- ▶ Nachrichten sind **unveränderliche** Daten, **reine** Funktionen oder **Futures**
- ▶ Die Zustellung von Nachrichten passiert höchstens einmal (Best-effort)
 - ▶ Wenn z.B. die Netzwerkverbindung abbricht, wird gewartet, bis der Versand wieder möglich ist
 - ▶ Wenn aber z.B. der Computer direkt nach Versand der Nachricht explodiert (oder der Speicher voll läuft), kommt die Nachricht möglicherweise niemals an
- ▶ Über den Zeitpunkt des Empfangs kann keine Aussage getroffen werden (Unbounded indeterminacy)
- ▶ Über die Reihenfolge der Empfangenen Nachrichten wird im Aktorenmodell keine Aussage gemacht (In vielen Implementierungen allerdings schon)
- ▶ Nachrichtenversand \neq (Queue | Lock | Channel | ...)

Kommunikation (Technisch)

- ▶ Der Versand einer Nachricht M an Aktor A bewirkt, dass zu **genau einem** Zeitpunkt in der Zukunft, das Verhalten B von A mit M als Nachricht ausgeführt wird.
- ▶ Über den Zustand S von A zum Zeitpunkt der Verarbeitung können wir begrenzte Aussagen treffen:
 - ▶ z.B. Aktor-Invariante: Vor und nach jedem Nachrichteneingang gilt $P(S)$
- ▶ Besser: Protokoll
 - ▶ z.B. auf Nachrichten des Typs T reagiert A immer mit Nachrichten des Typs U

Identifikation

- ▶ Aktoren werden über **Identitäten** angesprochen

Aktoren kennen Identitäten

- ▶ aus einer empfangenen Nachricht
 - ▶ aus der Vergangenheit (Zustand)
 - ▶ von Aktoren die sie selbst erzeugen
-
- ▶ Nachrichten können weitergeleitet werden
 - ▶ Eine Identität kann zu mehreren Aktoren gehören, die der Halter der Referenz äußerlich nicht unterscheiden kann
 - ▶ Eindeutige Identifikation bei verteilten Systemen nur durch Authentisierungsverfahren möglich

Location Transparency

- ▶ Eine Akteuridentität kann irgendwo hin zeigen
 - ▶ Gleicher Thread
 - ▶ Gleicher Prozess
 - ▶ Gleicher CPU Kern
 - ▶ Gleiche CPU
 - ▶ Gleicher Rechner
 - ▶ Gleiches Rechenzentrum
 - ▶ Gleicher Ort
 - ▶ Gleiches Land
 - ▶ Gleicher Kontinent
 - ▶ Gleicher Planet
 - ▶ ...

Sicherheit in Aktorsystemen

- ▶ Das Aktorenmodell spezifiziert nicht wie eine Aktoridentität repräsentiert wird
- ▶ In der Praxis müssen Identitäten aber **serialisierbar** sein
- ▶ Serialisierbare Identitäten sind auch **synthetisierbar**
- ▶ Bei Verteilten Systemen ein potentiellles Sicherheitsproblem
- ▶ Viele Implementierungen stellen **Authentisierungsverfahren** und **verschlüsselte** Kommunikation zur Verfügung.

Inkonsistenz in Aktorsystemen

- ▶ Ein Aktorsystem hat **keinen** globalen Zustand (Pluralismus)
- ▶ Informationen in Aktoren sind global betrachtet **redundant**, **inkonsistent** oder **lokal**
- ▶ Konsistenz \neq Korrektheit
- ▶ Wo nötig müssen duplizierte Informationen konvergieren, wenn "*längere Zeit*" keine Ereignisse auftreten (**Eventual consistency**)

Eventual Consistency

Definition

In einem verteilten System ist ein repliziertes Datum **schließlich Konsistent**, wenn über einen längeren Zeitraum keine Fehler auftreten und das Datum nirgendwo verändert wird

- ▶ Konvergente (oder Konfliktfreie) Replizierte Datentypen (CRDTs) garantieren diese Eigenschaft:
 - ▶ $(\mathbb{N}, \{+\})$ oder $(\mathbb{Z}, \{+, -\})$
 - ▶ Grow-Only-Sets
- ▶ Strategien auf komplexeren Datentypen:
 - ▶ Operational Transformation
 - ▶ Differential Synchronization
- ▶ dazu später mehr ...

Fehlerbehandlung in Aktorsystemen

- ▶ Wenn das Verhalten eines Aktors eine unbehandelte Ausnahme wirft:
 - ▶ Verhalten bricht ab
 - ▶ Aktor existiert nicht mehr
- ▶ Lösung: Wenn das Verhalten eine Ausnahme nicht behandelt, wird sie an einen überwachenden Aktor (**Supervisor**) weitergeleitet (**Eskalation**):
 - ▶ Gleiches Verhalten wird wiederbelebt
 - ▶ oder neuer Aktor mit gleichem Protokoll kriegt Identität übertragen
 - ▶ oder Berechnung ist fehlgeschlagen

"Let it Crash!" (Nach Joe Armstrong)

- ▶ Unbegrenzter Nichtdeterminismus ist statisch kaum analysierbar
- ▶ **Unschärfe** beim Testen von verteilten Systemen
- ▶ Selbst wenn ein Programm fehlerfrei ist kann Hardware ausfallen
- ▶ Je verteilter ein System umso wahrscheinlicher geht etwas schief
- ▶ Deswegen:
 - ▶ Offensives Programmieren
 - ▶ Statt Fehler zu vermeiden, Fehler behandeln!
 - ▶ Teile des Programms kontrolliert abstürzen lassen und bei Bedarf neu starten



Das Aktorenmodell in der Praxis

- ▶ Erlang (Aktor-Sprache)
 - ▶ Ericsson - GPRS, UMTS, LTE
 - ▶ T-Mobile - SMS
 - ▶ WhatsApp (2 Millionen Nutzer pro Server)
 - ▶ Facebook Chat (100 Millionen simultane Nutzer)
 - ▶ Amazon SimpleDB
 - ▶ ...
- ▶ Akka (Scala Framework)
 - ▶ ca. 50 Millionen Nachrichten / Sekunde
 - ▶ ca. 2,5 Millionen Aktoren / GB Heap
 - ▶ Amazon, Cisco, Blizzard, LinkedIn, BBC, The Guardian, Atos, The Huffington Post, Ebay, Groupon, Credit Suisse, Gilt, KK, ...

Zusammenfassung

- ▶ Das Aktorenmodell beschreibt **Aktorensysteme**
- ▶ Aktorensysteme bestehen aus **Aktoren**
- ▶ Aktoren kommunizieren über **Nachrichten**
- ▶ Aktoren können überall liegen (**Location Transparency**)
- ▶ Inkonsistenzen können nicht vermieden werden: **Let it crash!**
- ▶ Vorteile: Einfaches Modell; keine Race Conditions; Sehr schnell in Verteilten Systemen
- ▶ Nachteile: Informationen müssen dupliziert werden; Keine vollständige Implementierung