Prüfung Verteilte Softwaresysteme

Datum	:	12.07.2018, 12:30 Uhr
Bearbeitungszeit	:	90 Minuten
Prüfer	:	Prof. Dr. Oliver Braun
Hilfmittel	:	Keine
Erreichbare Punkte	:	80

Name: _____

Vorname: _____

Matrikelnummer: _____ Studiengruppe: _____

Hörsaal: _____ Platz Nr.: _____

Unterschrift: _____

Bitte kontrollieren Sie, ob Sie eine vollständige Angabe mit 4 Aufgaben auf 13 Seiten erhalten haben.

Aufgabe	1	2	3	4	Summe
max. Punkte	20	20	20	20	80

Anmerkungen:

- Schreiben Sie die Lösungen in die dafür vorgesehenen Kästchen. Sollte Ihnen der Platz dabei nicht reichen, benutzen Sie die Rückseite **und vermerken Sie das im dazugehörigen Kästchen!**

Aufgabe 1 (20 Punkte)

(a) Grenzen Sie die Begriffe *Netzwerkbetriebssystem* und *verteiltes Betriebssystem* voneinander ab. (4)

(b) Nennen Sie zwei Transparenzeigenschaften, welche die Verteilung verbergen und erklären Sie sie kurz. (4)

(c) Nennen Sie zwei Transparenzeigenschaften, welche die Verteilung zur Leistungssteigerung und Fehlertoleranz ausnutzen und erklären Sie sie kurz. (4)

(d) Grenzen Sie die *parallele Verarbeitung* und *nebenläufige Verarbeitung* voneinander ab. (4)

(e) Nennen Sie zwei Probleme die durch die Offenheit eines verteilten Systems entstehen können und erläutern Sie diese kurz. (4)

Aufgabe 2 (20 Punkte)

(a) Erläutern Sie kurz die Kernidee eines Actor-Systems. (4)

(b) Welche Schritte laufen beim Empfangen einer Nachricht in einem Actorsystem ab. (4)

(c) Gegeben sei folgender Scala-Code unter Nutzung des Akka-Frameworks. (12)
Die Aktoren haben einen eigenen Zustand (eine ganze Zahl), senden Ihren Zustand oder, auf Anforderung, alle Zustände, die sie sich in einer Map merken.
Zunächst werden vier Arten von Nachrichten definiert:

```
case class Msg(state: Int) // sendet eigenen Zustand
case object RequestStates // fordert Zustands-Map an
case class States(states: Map[ActorRef, Int]) // sendet Zustands-Map
case object DieImmediatly // beendet den Actor sofort
```

Die Klasse `StatesActor` ist folgendermaßen definiert:

```
class StatesActor extends Actor {
  // Zustandsmap initialisiert mit einem Eintrag für den Actor selbst
  var states = Map[ActorRef, Int](self -> 0)

  def receive = {
    case Msg(otherState) =>
      // Zustand des anderen in Map eintragen/updaten
      states += sender -> otherState
      // eigenen Zustand inkrementieren
      states += self -> (states(self) + 1)
      if ((states(self) + otherState) % 3 == 0) {
        sender ! RequestStates
        self ! States(states)
      } else {
        sender ! Msg(states(self))
      }
      if (states(self) == 3) {
        for (a <- states.keys)
          a ! DieImmediatly
      }

    case RequestStates => sender ! States(states)

    case States(statesFromOther) =>
      for((a,s) <- statesFromOther) {
        if (a != self && (!states.contains(a) || states(a) < s))
          states += a -> s
        if (a != self)
          a ! Msg(states(self))
      }

    case DieImmediatly =>
      // Stoppt diesen Actor sofort
      context.stop(self)
  }
}
```

Das Actorsystem startet drei Actors und speichert die `ActorRefs` mit jeweils dem initialen Zustand `0` in einer Map. Diese Map wird dann dem `statesActor1` geschickt.

```
object StatesApp extends App {
  val system = ActorSystem("states")

  val statesActor1 = system.actorOf(Props[StatesActor], "statesActor1")
  var states = Map[ActorRef, Int](statesActor1 -> 0)
```

```
  for (i <- 2 to 3) {
    states += system.actorOf(Props[StatesActor], "statesActor" + i) -> 0
  }

  import system.dispatcher

  system.scheduler.scheduleOnce(500 millis) {
    statesActor1 ! States(states)
  }

}
```

Geben Sie für die 3 Actors `statesActor1`, `statesActor2` und `statesActor3` getrennt an, welche Nachrichten von welchem Absender der Reihe nach in dessen Mailbox landen. Nachrichten, die nach einer `DieImmediatly`-Nachricht in der Mailbox landen, müssen Sie nicht angeben.

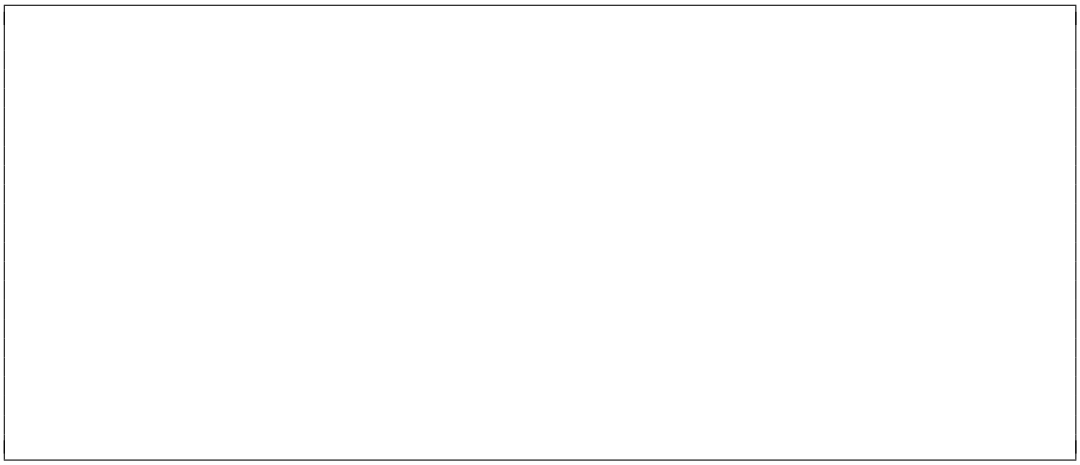
Geben Sie außerdem an wie sich der Zustand der `states`-Map über die Zeit ändert. Geben Sie die Zustände einfach als Map in der Form

`Map(statesActor13 -> 20, statesActor27 -> 108, statesActor32 -> 99)`

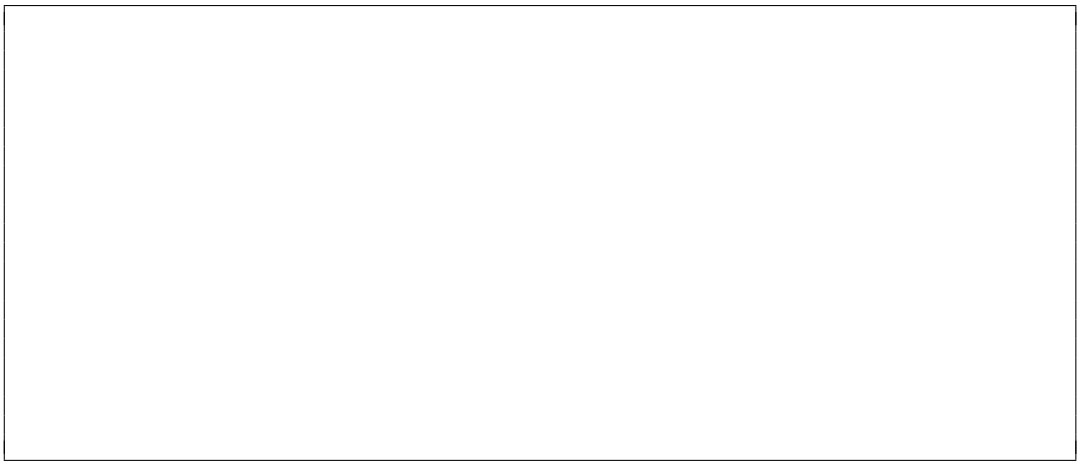
an.

Gehen Sie dabei davon aus, dass die Actors im Round-Robin-Verfahren in der Reihenfolge `statesActor1`, `statesActor2`, `statesActor3` gescheduled werden, nie parallel laufen und solange weiterarbeiten bis die Mailbox leer ist.

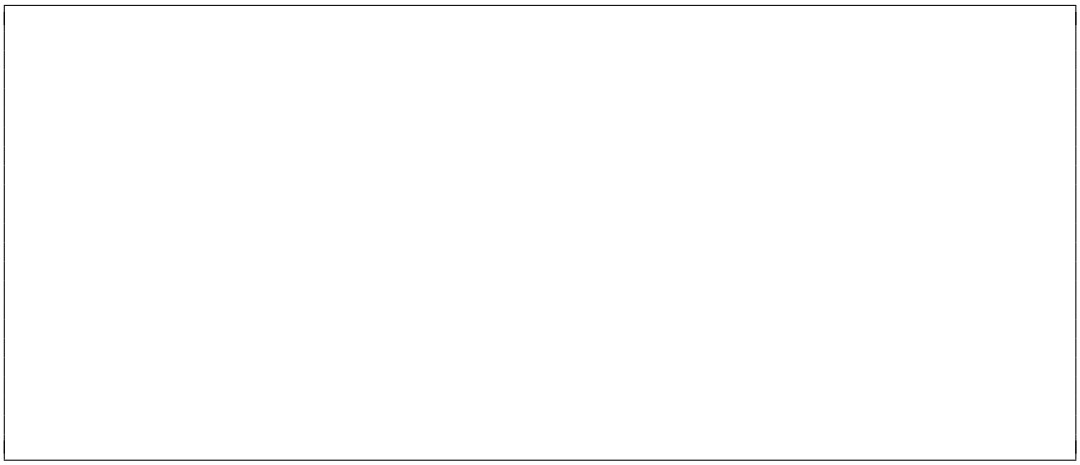
`statesActor1` — Mailbox:



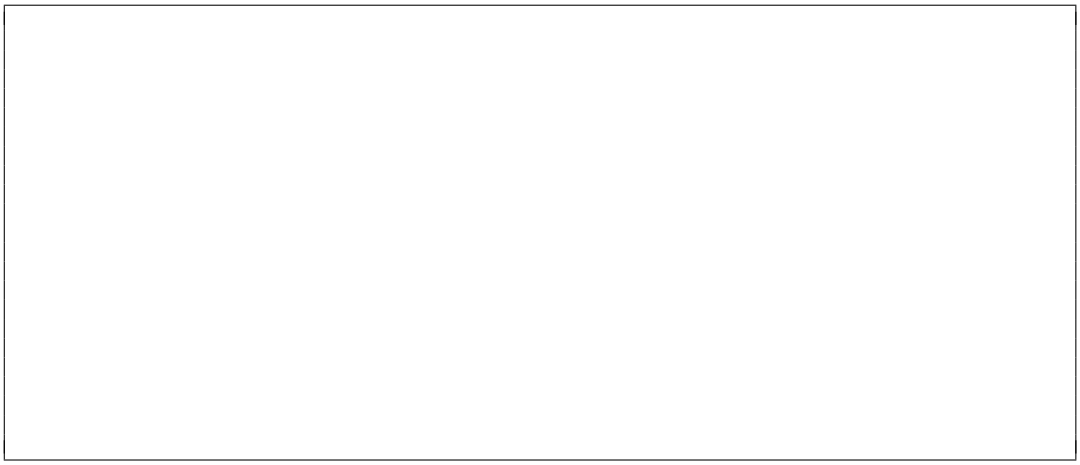
`statesActor1` — `states`-Map:



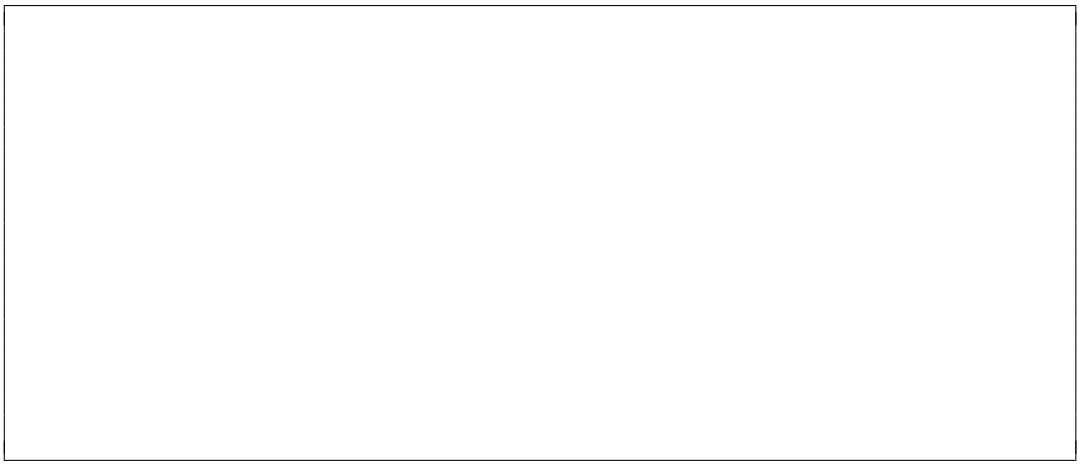
`statesActor2` — Mailbox:



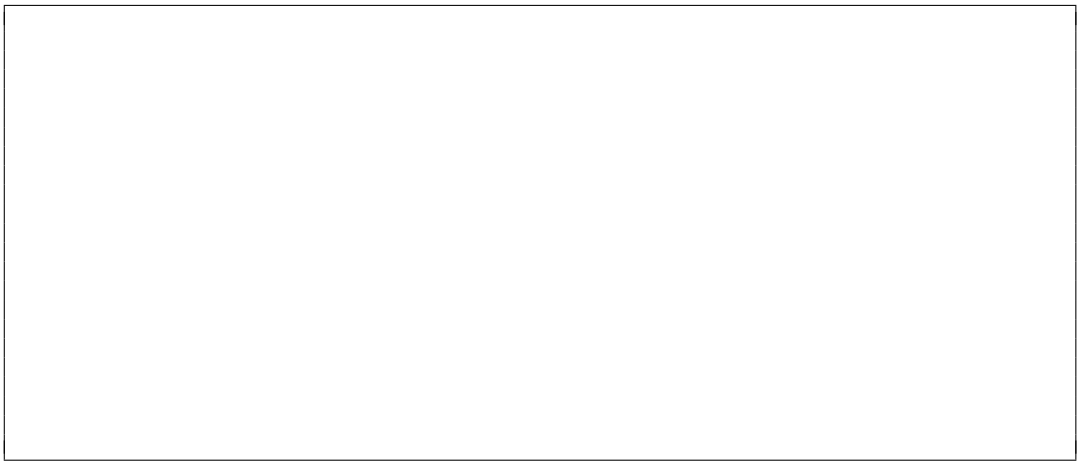
`statesActor2` — `states`-Map:



`statesActor3` — Mailbox:



`statesActor3` — `states`-Map:



Aufgabe 3 (20 Punkte)

(a) Bei der Bewertung von parallelen Programmen spielt die Laufzeit $T_p(n)$ eine wesentliche Rolle. Geben Sie die sechs Zeiten an, aus denen sie sich zusammen setzt und erläutern Sie diese kurz. (6)

(b) Was ist die Effizienz eines parallelen Programms, wie wird sie berechnet und welche Werte nimmt sie im Normalfall an? (5)

(c) Gegeben sei folgender, allgemein verständlicher, Go-Code zur Multiplikation zweier, quadratischer Matrizen `a` und `b`: (9)

```
for i := 0; i < size; i++ {
    for j := 0; j < size; j++ {
        value := 0
        for k := 0; k < size; k++ {
            value += a[i][k] * b[k][j]
        }
        res[i][j] = value
    }
}
```

Wie könnten Sie diesen Code *sinnvoll* parallelisieren?

Beschreiben Sie Ihr Vorgehen, schreiben Sie keinen Code. Wieviele Prozesse würden Sie in Abhängigkeit von der Problemgröße $n = size \cdot size$ verwenden um sie möglichst gleichmäßig auszulasten? Was müssten Sie den einzelnen Prozesse als Parameter geben und was müsste ein Master-Prozess dann noch machen? Wäre ein gemeinsamer Speicher hilfreich?

Aufgabe 4 (20 Punkte)

(a) Gegeben sind die 4 Prozesse P_1, P_2, P_3 und P_4 die mit Hilfe der Lamportzeit synchronisiert werden sollen. Alle Uhren stehen zu Beginn auf 0. Die 4 Prozesse verarbeiten die folgenden Ereignisse in der angegebenen Reihenfolge: (10)

Prozess P_1

- Bearbeitungsschritt
- Senden der Nachricht N_1
- Bearbeitungsschritt
- Empfangen der Nachricht N_5
- Senden der Nachricht N_2
- Bearbeitungsschritt
- Empfangen der Nachricht N_9

Prozess P_2

- Bearbeitungsschritt
- Bearbeitungsschritt
- Senden der Nachricht N_3
- Empfangen der Nachricht N_1
- Senden der Nachricht N_4
- Senden der Nachricht N_5
- Empfangen der Nachricht N_2
- Empfangen der Nachricht N_7
- Empfangen der Nachricht N_8
- Senden der Nachricht N_6

Prozess P_3

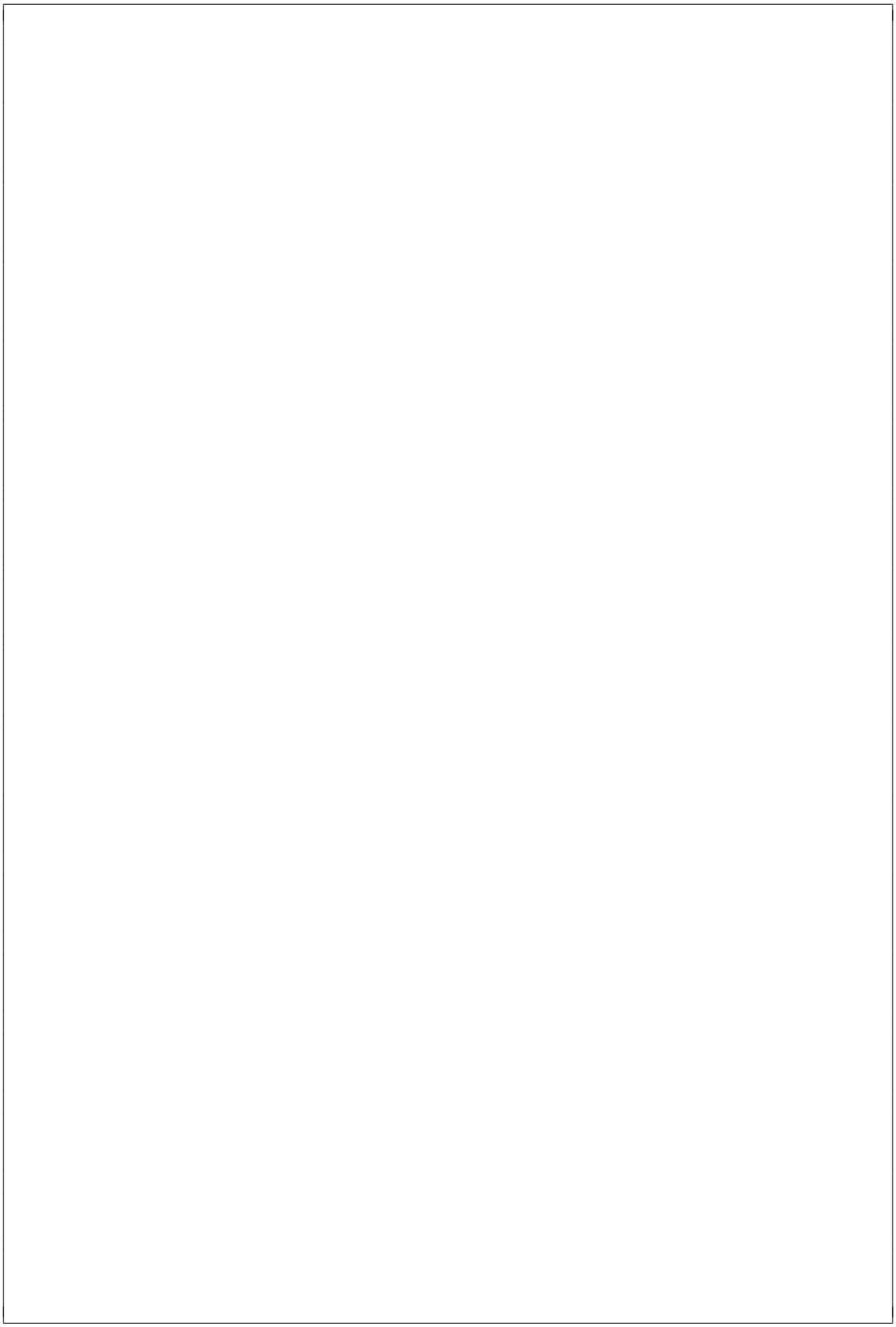
- Bearbeitungsschritt
- Empfangen der Nachricht N_3
- Bearbeitungsschritt
- Senden der Nachricht N_7
- Bearbeitungsschritt
- Empfangen der Nachricht N_6
- Empfangen der Nachricht N_{10}

Prozess P_4

- Bearbeitungsschritt
- Empfangen der Nachricht N_4
- Senden der Nachricht N_8
- Senden der Nachricht N_9
- Bearbeitungsschritt
- Senden der Nachricht N_{10}

Zeichnen Sie die Zeitachsen für die 4 Prozesse, tragen Sie die Ereignisse als Punkte, die Nachrichten als Pfeile ein und beschriften Sie die Nachrichten. Tragen Sie dann die Lamportzeit für jedes Ereignis ein.

Tipp: Es bietet sich an das Blatt quer zu nutzen.



(b) Welchen Nachteil hat die Lamportzeit und wie kann dieser durch Vektoruhren vermieden werden? (5)

(c) Nehmen wir an in Beispiel von Teilaufgabe (a) werden statt der Lamportzeit Vektoruhren genutzt. Die lokale Uhr soll wieder bei allen Prozessen bei 0 beginnen. Geben Sie die 10 Vektoruhren an, wie sie jeweils unmittelbar **nach** dem Empfangen einer Nachricht beim **empfangenden** Prozess vorliegen. Geben Sie zur besseren Übersichtlichkeit zuerst die Nachrichtennummer, gefolgt von der Vektoruhr als **Zeilenvektor** in der Reihenfolge P_1, P_2, P_3, P_4 an, z.B. (5)

$N_{12} : (230, 12, 34, 120)$