

Akka und Aktoren - Implementierung der Ameisen

Michael Walz & Fabian Mog

HTWG Konstanz - Big Data - 03.04.2017

Inhalt

- Aufgabenstellung
- Einführung Akka
- Ants Actor System
 - Implementierung
 - Demo
 - Metriken
 - Probleme
 - Ausblick
- Alternativen

Aufgabenstellung

- Ameisen sollen autonom auf einem Feld von einem definierten Startpunkt an einen definierten Zielpunkt laufen
- Die Ameisen schlagen dabei selbst eine zufällige nächste Position vor, die sie näher ans Ziel bringt
- Die Anfrage der Ameisen geht an einen Navigator, der prüft, ob die angefragte Position zulässig ist, d.h. diese Position noch nicht von einer anderen Ameise belegt wird
- Ameisen und Navigator werden mittels Aktoren implementiert (Scala/Akka)
- Fragestellungen:
 - Welche Eigenschaften müssen Ameisen besitzen?
 - Wie bestimmen Ameisen eine neue Wunschposition/Richtung, die sie ans Ziel bringt?
 - Wie sieht eine konkrete Implementierung aus?
 - Wie performant arbeitet diese Implementierung und von welchen Parametern hängt das ab?

Aktoren

- Eigenständige “Arbeiter”
- Kommunikation über asynchrone Nachrichten mit anderen Aktoren
- Auf verschiedene Nachrichten unterschiedlich reagieren
- Neue Aktoren erstellen
- Akteur arbeitet se-quen-zi-ell
- 1973 von Carl Hewitt vorgestellt

Akka

- Implementierung des Aktorenmodells
- Open Source Bibliothek für die JVM
- In Scala geschrieben
- Seit Scala 2.10 in Standardbibliothek vorhanden

Aktoren Nachrichten

- Nachrichten asynchron als “fire and forget”
- Nachrichten immutable
- Eintreffende Nachrichten in Queue (FiFo)
- Können verloren gehen, aber nie Duplikate
- Muss nicht in selber Reihenfolge ankommen, wie abgeschickt

Skalierbarkeit: Synchronized vs Asynchron

- Synchronized Methoden oder Lock-Objekte

vs

- Asynchrone Nachrichten

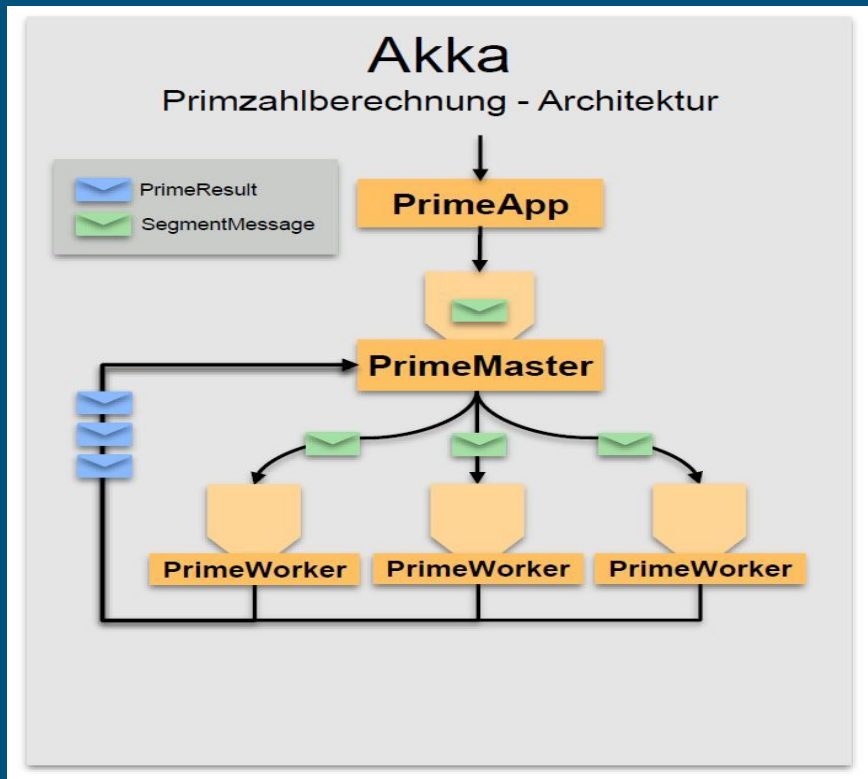
Lokal vs remote

```
akka {  
  actor {  
    provider = akka.remote.RemoteActorRefProvider  
  }  
  remote {  
    enabled-transport = [akka.remote.netty.tcp]  
    netty.tcp {  
      hostname = localhost  
      port = 2552  
    }  
  }  
}
```


ActorRef

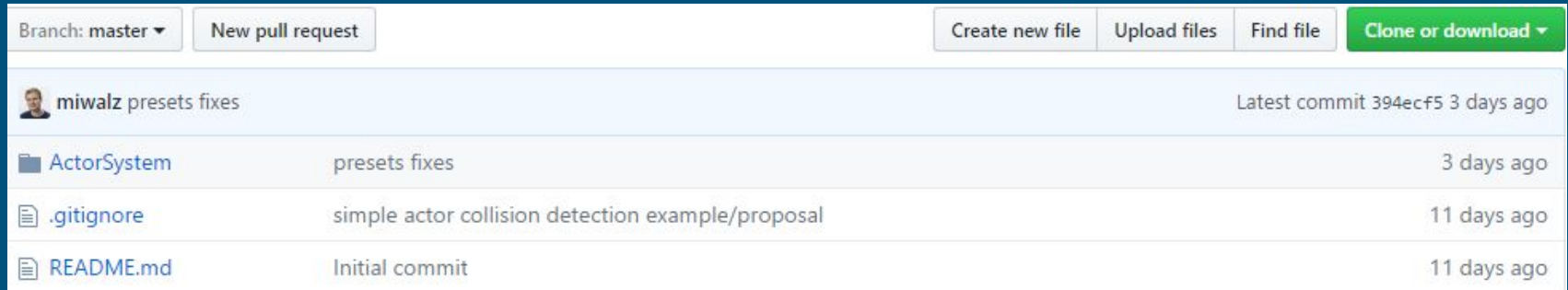
```
"akka://my-sys/user/service-a/worker1"           // purely local  
"akka.tcp://my-sys@host.example.com:5678/user/service-b" // remote
```

Beispiel







Ants Actor System - Repository

<https://github.com/miwalz/htwg-bigdata>



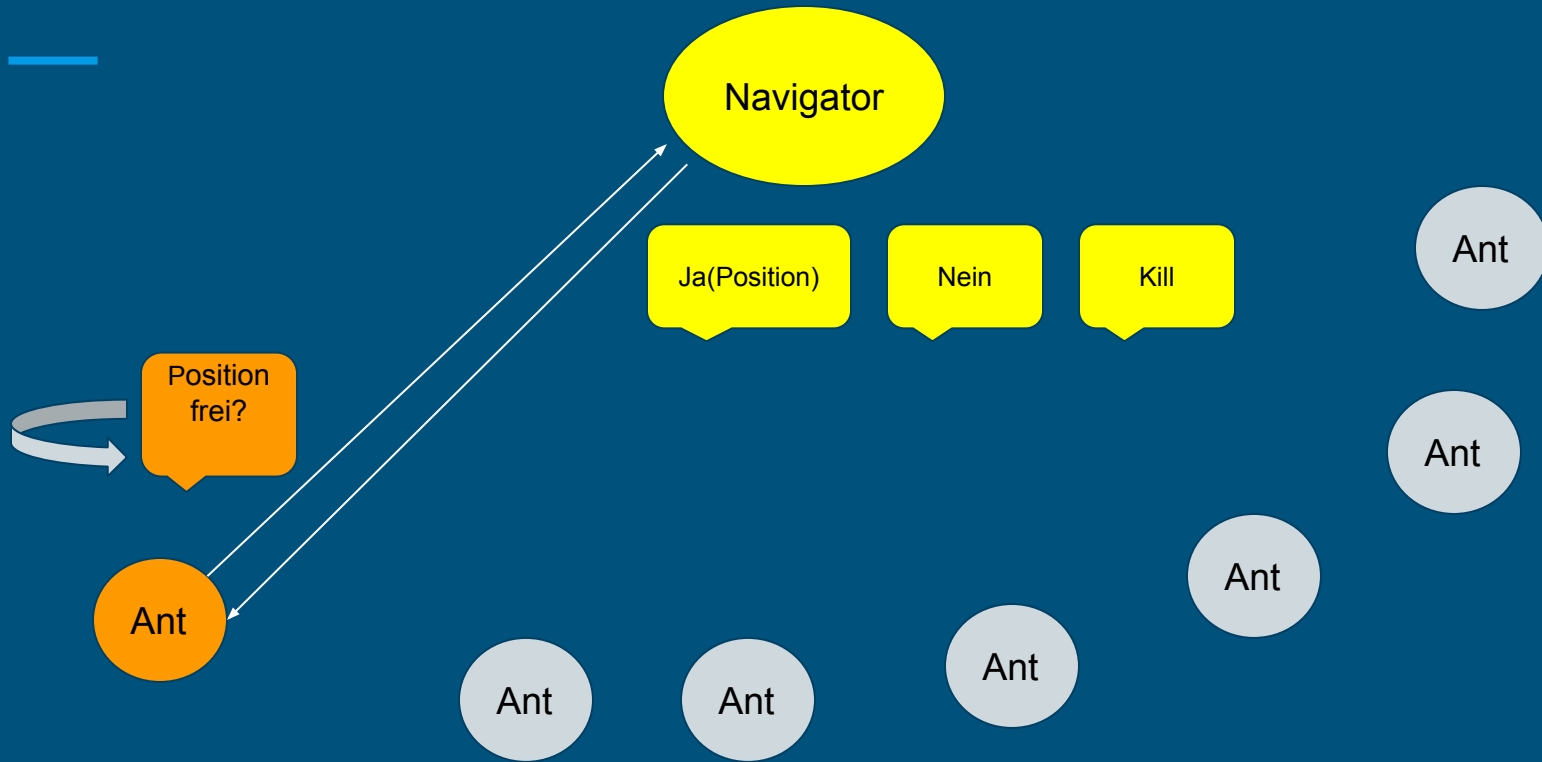
Branch: master ▾ New pull request Create new file Upload files Find file Clone or download ▾

 miwalz	presets fixes	Latest commit 394ecf5 3 days ago
 ActorSystem	presets fixes	3 days ago
 .gitignore	simple actor collision detection example/proposal	11 days ago
 README.md	Initial commit	11 days ago

Für Schreibzugriff auf das Repository bitte Mail an:

michaelwalz.rt@googlemail.com

Architektur



Main- Erzeugung von Ameisen und Navigation

```
private val system = ActorSystem("antSystem")

// create navigator actor
val navigator = system.actorOf(Props(new Navigator(positions)), "navigatorActor")

// create ants and schedule them to ask navigator for collisions
for (it <- 1 to Presets.MaxAnts) {
    val antPosition = Position(random.nextInt(Presets.SpawnWidth + 1), random.nextInt(Presets.SpawnWidth + 1))
    val antActor = system.actorOf(Props(new Ant(navigator, antPosition)), name = "ant_" + it)
    positions.put(antActor, antPosition)
}
```

Navigator - Implementierung + Datenhaltung

```
class Navigator(val antPositions: mutable.HashMap[ActorRef, Position]) extends Actor {
```

Ameise - Implementierung

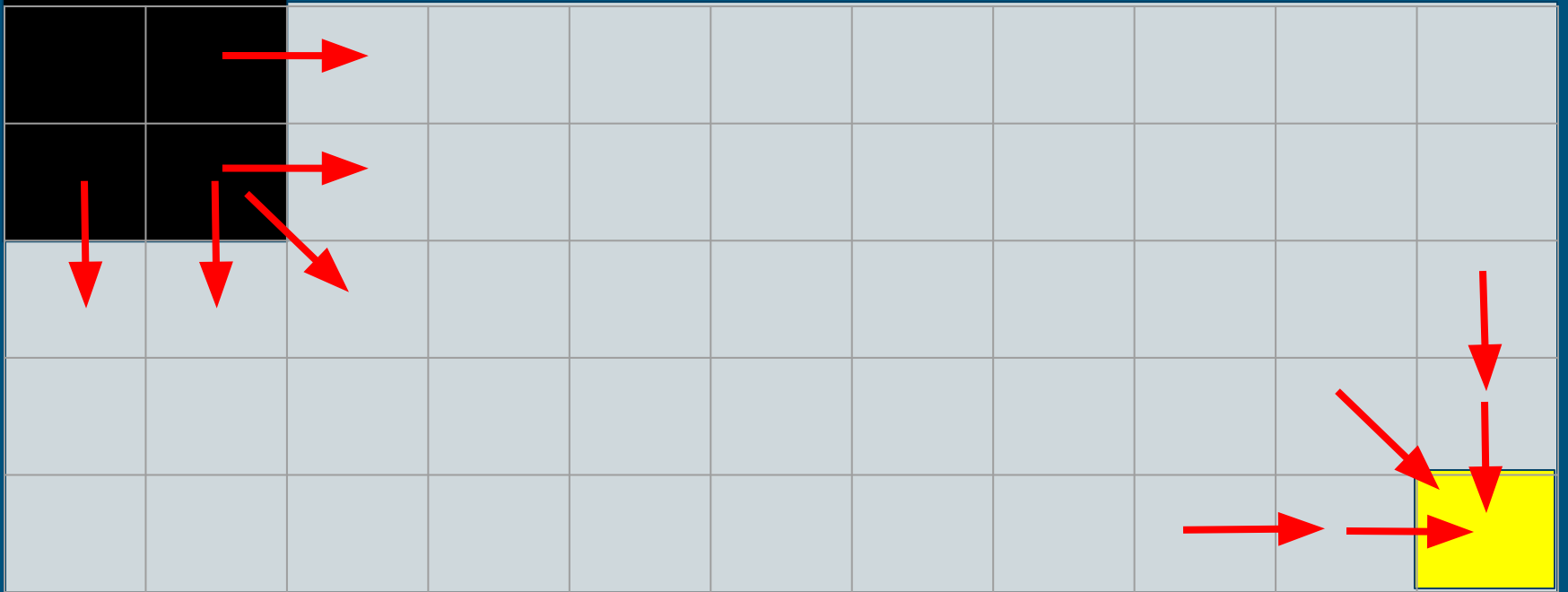
```
class Ant(val navigatorRef: ActorRef, var position: Position) extends Actor {
```

Ameise - Scheduler

```
private val duration = if (Presets.MaxDuration == 0) 0 else {  
    random.nextInt(Presets.MaxDuration - Presets.MinDuration) + Presets.MinDuration  
}  
  
private val cancellable =  
    system.scheduler.schedule(Duration.Zero, Duration(duration, "millis"))(tellNewPosition)
```


Bewegung der Ameise

Start



Ziel

Ameise - neue Position bestimmen

```
def tellNewPosition = {  
    // init result position with current position  
    var result = new Position(position.x, position.y)  
  
    // increase x OR y randomly OR (increase x and y)  
    var randomInt = random.nextInt(3)  
  
    // x already on border  
    if (position.x >= Presets.FinalPosition.x) {  
        randomInt = 2  
    }  
  
    // y already on border  
    if (position.y >= Presets.FinalPosition.y) {  
        randomInt = 1  
    }  
  
    if (randomInt == 0) {  
        if (position.x < Presets.FinalPosition.x && position.y < Presets.FinalPosition.y) {  
            result = new Position(position.x + 1, position.y + 1)  
        }  
    } else if (randomInt == 1) {  
        if (position.x < Presets.FinalPosition.x) {  
            result = new Position(position.x + 1, position.y)  
        }  
    } else if (randomInt == 2) {  
        if (position.y < Presets.FinalPosition.y) {  
            result = new Position(position.x, position.y + 1)  
        }  
    }  
  
    // ask navigator if position is empty  
    navigatorRef ! result  
}
```

Navigator - Nachrichtenempfang (1/2)

```
override def receive = {  
  
  case demandedPosition: Position => {  
  
    // check if ant did not reaches finish position  
    if (demandedPosition != Presets.FinalPosition) {  
  
      // check if position is empty  
      if (causesCollisions(demandedPosition)) {  
        collisions.incrementAndGet  
  
        sender ! Messages.FieldOccupied  
  
      } else {  
        antPositions.put(sender, demandedPosition)  
        movesDone.incrementAndGet  
        draw(antPositions, collisions, kills, failedKills, movesDone)  
  
        sender ! demandedPosition  
      }  
    }  
  }  
}
```

Navigator - Kollisionserkennung

```
def causesCollisions(position: Position): Boolean = {  
    antPositions.values.exists(pos => pos == position)  
}
```

Navigator - Nachrichtenempfang(2/2)

```
} else {  
  // ant demands finish position --> kill ant  
  val removed = antPositions.remove(sender)  
  
  if (!removed.isDefined) {  
    failedKills.incrementAndGet  
  } else {  
    kills.incrementAndGet  
  
    if (Presets.ShowProgressBar) println("\nAnt finished! Finished ants count is: " + kills)  
    draw(antPositions, collisions, kills, failedKills, movesDone)  
  
    sender ! Messages.Finished  
  
    // shutdown actor system if all ants have finished  
    if (antPositions.isEmpty) {  
      ActorSystem("antSystem").terminate  
      AntsSimulation.exitSimulation(antPositions, collisions, kills, failedKills, movesDone)  
    }  
  }  
}  
}  
}  
  
case _ => // do nothing  
}
```

Ameise - Nachrichtenempfang

```
override def receive: PartialFunction[Any, Unit] = {  
  
  case pos: Position =>  
    // set new position  
    position = pos  
  
  case Messages.FieldOccupied => // do nothing  
  
  case Messages.Finished =>  
    // final position reached  
    cancellable.cancel  
    context.stop(self)  
  
  case _ => println("unknown message")  
}
```

Live Demo

Metriken - Parameter

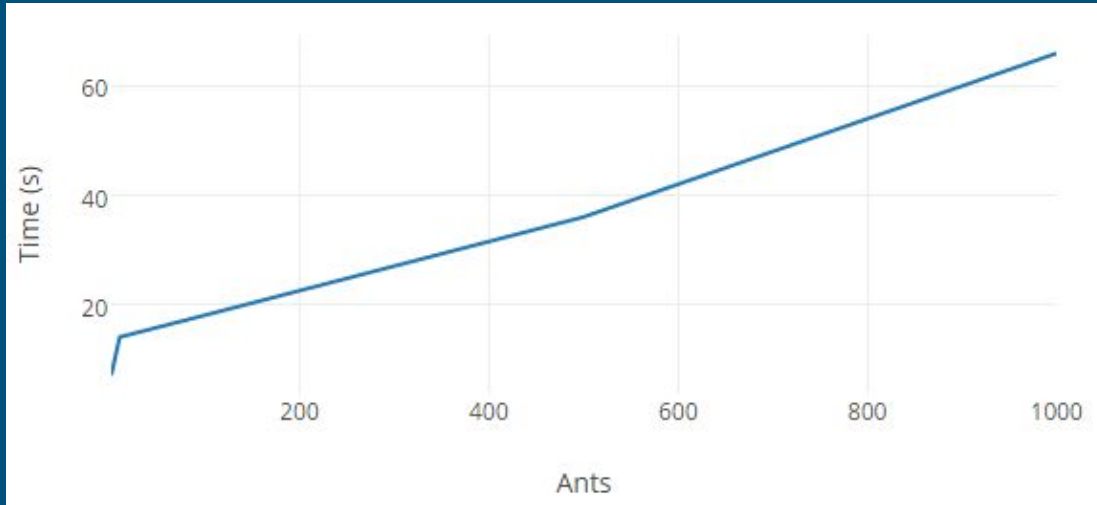
- Parameter werden über **Presets.scala** gesetzt:
 - **FieldWidth:** Seitenlänge des Simulationsfeldes
 - **SpawnWidth:** Seitenlänge des "Spawn"-Bereichs
 - **MaxAnts:** Anzahl der zu generierenden Ameisen
 - **MinDuration:** Kürzeste Zeit, die zwischen zwei Anfragen einer Ameise vergeht
 - **MaxDuration:** Längste Zeit, die zwischen zwei Anfragen einer Ameise vergeht
 - **ShowBoard:** Spielfeld in TextualUI ausgeben?
 - **ShowStats:** Live-Daten während Simulation ausgeben?
 - **ShowProgressBar:** Zeigt nur, ob Simulation noch aktiv
 - **WriteToFile:** Ergebnisse der Simulation in Datei schreiben?

Metriken - Daten

- Daten, die während der Simulation gesammelt werden:
 - **Moves:** Anzahl der Schritte aller Ameisen
 - **Collisions:** Anzahl der Kollisionen aller Ameisen
 - **AntsFinished:** Anzahl der Ameisen, die Ziel erreicht haben
 - **Time:** Dauer der Simulation
 - **Result:** "OK" wenn Schritte gemacht wurden und alle Ameisen im Ziel sind, ansonsten "Not OK"

Metriken - Ergebnisse

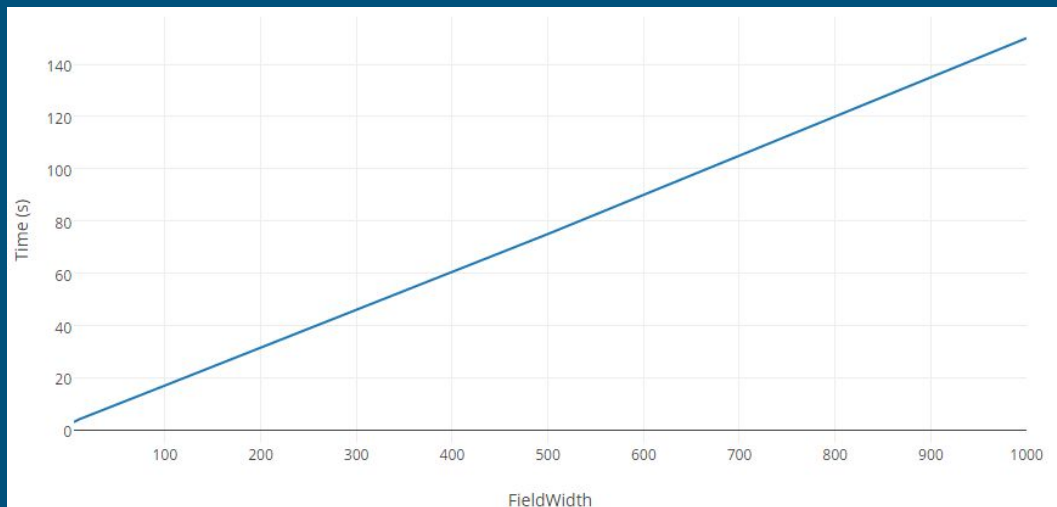
Erhöhen von MaxAnts (bei Feldgröße 100):



Zeit wächst konstant mit Ameisen, bis zu viele Ameisen → Zu viele Threads

Metriken - Ergebnisse

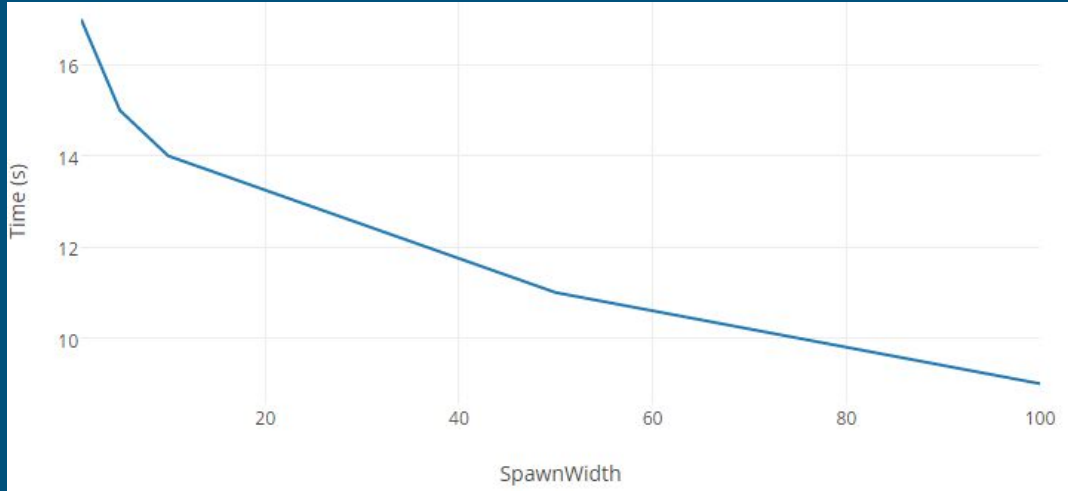
Ändern der Feldgröße (bei MaxAnts von 50):



Feldgröße und Zeit verhalten sich (wie erwartet) proportional zueinander

Metriken - Ergebnisse

Ändern des Spawn-Bereichs (bei Feldgröße 100 und 50 Ants):



Bei großem Spawn-Bereich können mehr Ameisen gleichzeitig starten

Metriken - Ergebnisse

Ändern der Anfragehäufigkeit der Ameisen:

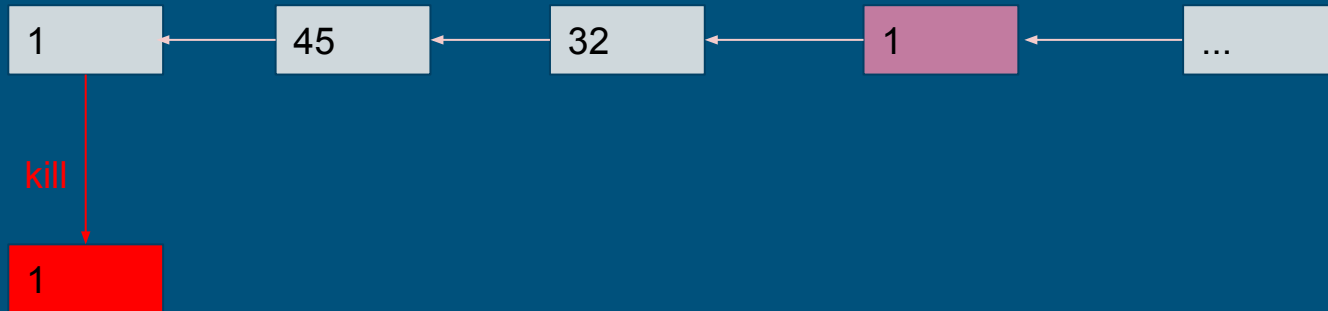
- Konfigurationen für MinDuration / MaxDuration:
 - **0 ms / 0 ms:**
 - Alle Ants fragen ununterbrochen an!
 - Es werden neue Positionsanfragen gestellt, bevor die Antwort auf die alte Anfrage von der Ameise verarbeitet wurde
 - Navigator kommt nicht nach
 - **20 ms / 20 ms:**
 - Auch hier kann es Probleme geben: Alle Ameisen feuern im gleichen Takt! → Überlasten des Navigators

Problem - Actor stoppen

- Laufende Scheduler verhindern das Beenden eines Actors

Problem - Nachrichten Queue

- Löschen einer Ameise, welche bereits gelöscht ist
- Nachricht an einen Akteur, der nicht (mehr) existiert -> **Dead Letters**












Probleme - Actor Timeout

- Existieren zu viele Ameisen/Aktoren kommt es zu einem Actor Timeout
- Flaschenhals ist dabei der Navigator-Actor, da sich dort alle Anfragen sammeln
- **Mögliche Lösung:**
 - Die Arbeit des Navigator-Aktors muss aufgeteilt und delegiert werden (Divide & Conquer)
 - Mehr dazu unter "Ausblick"

Probleme - Actor Timeout

Untersuchung mit Java VisualVM:

CPU samples		Thread CPU Time				
 		 Snapshot		Thread Dump		
Hot Spots - Method		Self Time [%] ▼	Self Time	Self Time (CPU)	Total Time	Total Time (...) 
scala.concurrent.forkjoin.ForkJoinPool. idleAwaitWork ()			1.100.734 ms (43,9%)	3.069 ms	1.100.734 ms	3.069 ms 
scala.concurrent.forkjoin.ForkJoinPool. scan ()			976.632 ms (38,9%)	2.369 ms	2.165.368 ms	93.441 ms 
 Method Name Filter (Contains)						

- Bei 10.000 Ameisen werden >10.000 Threads gestartet!
- **ForkJoinPool.idleAwaitWork**: Navigator kommt nicht nach, Ants warten
- **ForkJoinPool.scan**: Zu viele Threads, zu viele Context-Switches

Ausblick - Navigator Workers

Problem: Alle Ameisen befeuern den selben Navigator → Bottleneck!

Lösung: Aufteilen der Zuständigkeit:

- Erstelle Master Navigator, der alle Positionsanfragen annimmt
- Das Feld wird vom Master in Abschnitte eingeteilt
- Jeder Feldabschnitt wird von einem Worker Navigator bearbeitet
- Master teilt Anfragen je nach Position den passenden Workern zu
- So hält jeder Worker nur einen kleinen Teil des Feldes → Daten müssen nicht geteilt werden, da Anfrage für eine bestimmte Position unabhängig von der restlichen Feldbelegung bearbeitet werden kann

Ausblick - Navigator Workers

```
case class WorkerRequest(antRequest: AntRequest, antRef: ActorRef)
case class AntRequest(demandedPosition: Position, currentPosition: Position)

class Master(val numberOfWorkers: Int) extends Actor {

  val workers: HashMap[Int, ActorRef] = HashMap()

  override def receive: Receive = {
    case antRequest: Messages.AntRequest => {
      val targetWorkerNumber = antRequest.demandedPosition.y % numberOfWorkers
      if (!workers.contains(targetWorkerNumber)) {
        // create worker
        workers.put(targetWorkerNumber, context.actorOf(Props[Worker], name = "worker" + targetWorkerNumber))
      }
      // divide and conquer
      workers.get(targetWorkerNumber).get ! Messages.WorkerRequest(antRequest, sender)
    }

    [...]
  }
}
```

Ausblick - Navigator Workers

```
class Worker extends Actor {  
  var antPositions: Set[Position] = Set()  
  
  override def receive = {  
  
    case workerRequest: Messages.WorkerRequest => {  
      val demandedPosition = workerRequest.antRequest.demandedPosition  
      val currentPosition = workerRequest.antRequest.currentPosition  
      val antRef = workerRequest.antRef  
  
      if (demandedPosition.x < Presets.FinalPosition.x || demandedPosition.y < Presets.FinalPosition.y) {  
        if (causesCollisions(demandedPosition)) {  
          antRef ! Messages.FieldOccupied  
        } else {  
          antPositions -= currentPosition  
          antPositions += demandedPosition  
          antRef ! demandedPosition  
  
          // draw(...)  
        }  
      } else {  
        // ant demands finish position --> kill ant  
        antPositions -= currentPosition  
        antRef ! Messages.Kill  
  
        // draw(...)   
        // TODO: shutdown worker and tell master  
      }  
    }  
  }  
  [...]
```

Ausblick - Navigator Workers

Herausforderungen:

- Spielfeldzustand wird über mehrere Aktoren verteilt...
 - Wie kann Spielfeld ausgegeben werden?
 - Hierzu müssen alle Aktoren dem Master ihren aktuellen Zustand mitteilen; Der Master kann die Einzelfelder zusammensetzen und die *draw()*-Funktion aufrufen
 - **Aber:** Die Gültigkeit des zusammengesetzten Feldes kann nicht mehr garantiert werden, da die Worker ihr Teilfeld nicht synchronisiert an den Master schicken
- Dadurch wird es sehr schwer die Korrektheit des Programms nachzuvollziehen

Ausblick - Ameisen

- Weitere Ideen für die Ameisen:
 - Andere (komplexere) Laufwege
 - Evtl. "Fette Ameisen", die mehrere Felder belegen, um die Rechenzeit am Navigator zu erhöhen
 - Evtl. komplexere Funktion zur Berechnung der neuen Wunschposition, um die Rechenzeit der Ameisen zu erhöhen
- Weitere Ideen für Wege:
 - Bei einem kleinen Spawn-Bereich kommt es am Anfang zu Stau und schon erzeugte Ameisen müssen evtl. sehr lange Warten, bis diese loslaufen können
→ Man könnte neue Ameisen erst erzeugen, wenn Startposition frei ist (geordneter Start)
 - Da das Ziel aus einer Position besteht, kommt es im Zielbereich zu Stau
→ Man könnte einen Bereich als Ziel definieren, um dies zu verhindern

Alternative Technologien

- Apache Spark
 - Entwickelt mit Scala
 - Ähnliches Konzept wie Akka
 - Echtzeitverarbeitung von Daten, z.B.:
 - E-Health: DNA-Vergleiche
 - Bildverarbeitung: Feature-Extraction
 - Map-Reduce basierte Computations
 - Einfache Anbindung an Hadoop, Cassandra, etc.
 - Spark eher geeignet für Datenanalysen aus riesigen heterogenen Datenmengen
 - **Akka eher geeignet für Gaming und Simulationen**

Alternative Technologien

- Erlang (Ericsson Language)
 - Zuverlässig, schnell lernbar
 - Bündelt alles, um hochskalierbare Anwendungen zu erstellen
 - Sehr ausgereift, viel im Einsatz
 - **Beispiele:** WhatsApp Backend, Facebook Messaging, CouchDB, Amazon SimpleDB, u.v.m.
 - Bietet Hot-Swap von Code Modulen
 - Erlang Aktoren übergeben Zustandsvariablen als Parameter, in Akka kapselt jeder Aktor einen eigenen Zustand
 - Extrem leistungsfähig bei Concurrency Problemen, aber bei aufwändigen Berechnungen schwächer als die JVM
 - Keine große Auswahl an Bibliotheken wie bei JVM → Ansprechen einer Datenbank in Erlang beispielsweise sehr viel aufwändiger als in Java/Scala

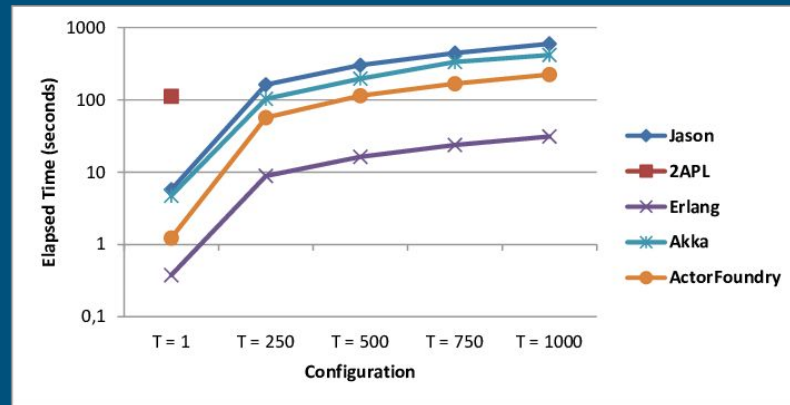
Alternative Technologien

- Clojure
 - Funktionale Sprache auf der Java Virtual Machine
 - Vereinfacht Nebenläufige Programmierung sehr stark!
 - Nutzt für die asynchrone Kommunikation/Messaging die Clojure Bibliothek core.async, welche nach dem Channelsystem, das dem Aktorensystem von Akka ähnelt
 - **Aber:** nicht gut geeignet für verteilte Anwendungen → Kein Kandidat für Big Data!
- Multiagentensysteme
 - spezialisierte Software-Einheiten lösen gemeinsam ein Problem mit Hilfe verteilter, künstlicher Intelligenz
 - Beispiel aus der Biologie: Ameisen
 - Beispiele: Jason, 2APL

Alternative Technologien

Metriken

- Tokens werden von einem Actor zum nächsten gesendet; Dabei wird jeweils eine Berechnung vorgenommen
- Testet Performance für Parallelität und Concurrency verschiedener Programmiersprachen
- Erlang liefert hier die beste Performance!
Aber: Artikel von 2013, Akka hat vermutlich aufgeholt!



Quelle: Rafael C. Cardoso, Maicon R. Zatelli, Jomi F. Hübner, Rafael H. Bordini:
Towards Benchmarking Actor- and Agent-Based Programming Languages;
October 2013