Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Ausarbeitung Anwendungen 2 – SoSe 2009

Dominik Charousset

Softwarearchitekturen für die Entwicklung verteilter Anwendungen

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

# Inhaltsverzeichnis

# 1 Einführung

Der aktuell zunehmende Architekturwechsel hin zu Mehrkernsystemen bedeutet für die Programmierung von Computerprogrammen, dass sie nebenläufig ausführbar sein müssen, um von der zunehmend nebenläufigen Hardware profitieren zu können:

> "multi-core processors make concurrency an essential ingredient of efficient program execution" (Haller und Odersky, 2009, S. 1)

Programmierung nebenläufiger Anwendungen bedeutet in Sprachen wie C++ oder Java heute meist das Auslagern einzelner Programmteile in *Threads*. Wobei alle Threads prinzipiell Zugriff auf den Speicher des Prozesses, zu dem sie gehören, haben.

Der Zugriff mehrerer Threads auf gemeinsam benutzte Speicherbereiche (*shared memory*) und Datenstrukturen muss vom Entwickler explizit synchronisiert werden. Eine fehlerhafte Implementierung kann zu Seiteneffekten, wie *Race Conditions*, *Deadlocks* und/oder *Lifelocks*, führen. Das korrekte Verhalten der Anwendung ist zudem schwierig zu verifizieren:

> "The main difficulty of multiprogramming is that concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing." (Hansen, 1973)

In einer multithreaded Anwendung muss jedes Objekt (bzw. Datenstruktur), auf das von mehr als einem Thread zugegriffen wird *thread safe* sein, d.h. es muss sichergestellt sein, dass bei parallelem Zugriff keine Seiteneffekte auftreten. Die Implementierung ist dabei um ein vielfaches aufwändiger und fehleranfälliger, als bei Anwendungen mit nur einem Thread (üblicherweise der *main*-Thread).

Selbst Implementierungen, die scheinbar thread safe (und in der Praxis anzutreffen) sind, können unter gewissen Umständen zu undefiniertem Verhalten führen, wie Meyers und Alexandrescu (2004) gezeigt haben.

Multithreaded Anwendungen laufen auf Mehrkernsystemen allerdings schneller, je *weniger* Zugriffe auf gemeinsame Speicherbereiche sie haben. Das liegt u.A. an der Hardwarearchitektur: RAM ist deutlich langsamer als die Prozessoren, weshalb diese einen lokalen, schnellen Cache haben. Je mehr ein Prozessor auf seinem Cache arbeiten kann, desto schneller ist die Ausführung und je öfter ein Thread auf gemeinsame Speicherbereiche zugreift, desto weniger

kann der Cache benutzt werden. Je mehr Locks eine Anwendung benutzt, desto wahrscheinlicher wird es zudem, dass mehrere Threads auf eine Ressource zeitgleich zugreifen wollen und auf Freigabe eines Locks warten müssen.

Ein alternatives Konzept zur Implementierung verteilter Anwendungen ist das *Aktormodell*. Aktoren sind nebenläufige Softwarekomponenten, die keine gemeinsame Sicht auf einen Speicherbereich haben und durch asynchronen Nachrichtenaustausch miteinander kommunizieren.

Auch wenn die zugrundeliegende Theorie aus den 1970er Jahren stammt, ist sie durch die aufkommende parallelisierung der Hardware wieder sehr aktuell. Die vorhandenen Implementierungen existieren entweder in bislang wenig bekannten, funktionalen Sprachen wie Erlang (Kap. 2.2) oder sind Bestandteil neuer Entwicklungen der letzten Jahre, wie z.B. Scala (Kap. 2.3).

Auch für “klassische” objektorientierte Sprachen wie Java oder C# existieren aktuelle Projekte, die eine Programmierung nach dem Aktormodell ermöglichen sollen (Kap. 2.4).

Dennoch ist das Aktormodell in der Praxis derzeit noch wenig verbreitet und insbesondere im C++ [1] Umfeld existiert keine (ausreichende) Implementierung.

Ziel dieser Arbeit ist daher die Entwicklung einer C++ Bibliothek zur Programmierung verteilter Anwendungen, die dem Aktormodell folgt und die den vorläufigen Projektnamen “acedia” trägt.

[1] Nach wie vor eine der am meisten genutzten Sprachen und bald in einer neuen Sprachversion mit einer neuen, zeitgemäßen Standardbibliohek verfügbar

# 2 Related Work

In diesem Kapitel wird zunächst das Aktormodell im Detail diskutiert. Anschließend werden zwei Sprachen vorgestellt, die die Entwicklung von acedia maßgeblich beeinflussen und Projekte, die in einem ähnlichen Kontext und ähnlicher Motivation entstanden sind.

## 2.1 Das Aktormodell

Unter einem Aktor versteht man:

> "self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing." (Agha, 1990, S. 128)

Aktoren können zu ihrer Laufzeit weitere Aktoren erschaffen. Da sie keine gemeinsame Sicht auf einen Speicherbereich haben, entspricht das Modell einer *shared nothing* Architektur (Stonebraker, 1986), in der alle Komponenten voneinander isoliert sind.

Jeder Aktor besitzt eine Mailbox, in der eingehende Nachrichten bis zu ihrer Verarbeitung zwischengespeichert werden und auf die nur er Zugriff hat. Dadurch werden (potentielle) Probleme einer *shared memory* Architektur – insbesondere *race conditions* – vermieden (Haller und Odersky, 2009, S. 214) und die Komplexität, sowie der Implementierungsaufwand für den Entwickler dadurch verringert.

Ein weiterer Vorteil des Aktormodells ist, dass es bei der Kommunikation konzeptionell gleichgültig ist, ob die Nachricht an einen lokalen, auf der gleichen Hardware ausgeführten Aktor oder über ein Netzwerk geschickt wird. Damit ist das Paradigma und die Art und Weise wie eine Anwendung verteilt wird prinzipiell die Gleiche – unabhängig davon, ob das System lokal auf mehrere Prozessoren oder über ein Netzwerk auf mehrere Rechner verteilt wird.

Die Architektur des Aktormodells hat zur Folge, dass ein Aktor nur mit ihm bereits bekannten anderen Aktoren kommunizieren kann. Dazu kann dem Aktor bei seiner Erschaffung eine Liste anderer Aktoren mitgegeben werden oder es muss einen zentralem Namensdienst geben, um bestimmte (besonders wichtige, bzw. zentrale) Aktoren finden zu können.

Ein Problem aller verteilter Anwendungen ist es, Fehler entdecken und beheben zu können. Implementierungen des Aktormodells wie z.B. in Erlang bieten die Möglichkeit, Aktoren miteinander zu verknüpfen. Fällt ein Aktor aus, werden alle mit ihm verknüpften Aktoren informiert. Durch diesen Mechanismus kann auf den Ausfall einzelner Komponenten reagiert werden und diese ggf. neugestartet oder ersetzt werden.

## 2.2 Erlang

Erlang (entwickelt seit 1985) ist eine funktionale, dynamisch typisierte Sprache, die für die Bedürfnisse verteilter Anwendungen entwickelt wurde. Das Aktormodell ist ein fundamentaler Bestandteil der Sprache:

> "Erlang is today the de facto implementation of an actor-model and it is used as reference for many of the newest actor-model implementations." (Corrêa, 2009, S. 23)

In Erlang gibt es keine Threads, sondern ausschließlich Prozesse. Des Weiteren gibt es keinen gemeinsamen Speicher. Kommunikation unter Prozessen findet ausschließlich über asynchronen Nachrichtenaustausch statt. Jeder Prozess ist also ein Aktor (Armstrong, 2007a, S. 1).

Nachrichten können mittels `receive` empfangen werden (mit optionalem, nach `after` angegebenem Timeout), dem eine Liste von Pattern[2] und zugehörige Anweisungen mitgegeben wird. Wenn die Nachricht ein Pattern erfüllt und ggf. der optionale Guard[3] erfüllt ist, werden die entsprechenden Anweisungen ausgeführt ((Armstrong, 2007b, S. 150)):

```
receive
    Pattern1 [when Guard1] -> Expressions1;
    Pattern2 [when Guard2] -> Expressions2;
    ...
    after Time -> Expressions
end
```

Das folgende Beispiel zeigt ein `receive` mit drei Pattern. Das Erste prüft, ob eine Ganzzahl empfangen wurde. Das Zweite prüft, ob die Empfangene Nachricht ein Tupel bestehend aus dem Atom[4] "hello" und einem beliebigen Wert ist. Das dritte Pattern hat keine Bedingungen und bindet die empfangene Nachricht an die Variable "Y". Wenn nach 50 Millisekunden noch keine Nachrichten empfangen wurde, wird der `after`-Block ausgeführt.

```
receive
  X when is_integer(X) -> io:format("Received an integer~n") ;
  {hello, What} -> io:format("Received: hello ~s~n", [What]) ;
  Y -> io:format("Received: ~p~n",[Y])
  after 50 -> io:format("Received nothing~n")
end
```

[2] Ein Pattern ist eine Regel, die den Aufbau (bzw. Inhalt) einer Variablen beschreibt
[3] Guards sind Ausdrücke, die nach `true` oder `false` ausgewertet werden können
[4] Atome sind klein geschriebene, konstante Werte (Vergleichbar mit Enums in Java)

Nachrichten, die kein Pattern erfüllen, verbleiben in der Mailbox und werden beim nächsten `receive` erneut geprüft. Dadurch kann ein Aktor die Nachrichten empfangen, die er in seinem aktuellen Zustand, bzw. Verhalten verarbeiten kann, ohne Nachrichten zu verlieren, die nach einem Zustands-/Verhaltenswechsel verarbeitet werden sollen. `receive` kann geschachtelt werden, um z.B. eine Reihe semantisch zusammengehöriger Nachrichten geschlossen, nacheinander zu empfangen und zu verarbeiten.

Prozesse (also Aktoren) werden mit der Funktion `spawn` gestartet und lassen sich auch auf anderen Knoten (Rechner im Netzwerk) starten. Mit Prozessen auf anderen Knoten lassen sich alle Operationen ausführen, die auch auf lokalen Prozessen möglich sind.

`spawn` gibt eine Prozess-ID (kurz PID) zurück, über die dem gestarteten Prozess Nachrichten mit dem `!`-Operator geschickt werden können. Um in einer großen verteilten Anwendung diese PID systemweit bekannt zu machen, gibt es die Funktion `register`, der ein Atom und eine PID übergeben wird. Um einen registrierten Prozess zu erreichen, genügt es, das Atom zu kennen, mit dem er registriert wurde (es kann nachfolgend anstelle der PID verwendet werden):

```
Pid = spawn(...),
register(my_actor, Pid),
my_actor ! "Hello Actor".
```

Die Funktion `register` stellt also einen Namensdienst für Prozesse zur Verfügung.

Um Ausfallsicherheit implementieren zu können, kann jeder Prozess mit einem oder mehrerer anderer Prozesse verbunden werden:

> "A frequent assumption made when writing Erlang software is that any Erlang process may unexpectedly die (...) The runtime system provides a mechanism to notify selected processes of the fact that a certain other process has terminated; this is realized by a special message that arrives in the mailbox of processes that are specified to monitor the vanished process." (Earle u. a., 2005, S. 27)

Darüber hinaus stellt Erlang mit dem "*supervision tree*" dem Entwickler ein konfigurierbares Werkzeug zur Verfügung, mit dem er Prozesse gruppieren und überwachen kann:

> "A supervision tree is a tree of processes. The upper processes (supervisors) in the tree monitor the lower processes (workers) in the tree and restart the lower processes if they fail." (Armstrong, 2007b, S. 351)

Der *supervision tree* stellt auch Möglichkeiten bereit, um Loops zu erkennen (z.B. wenn Worker immer aus dem gleichen Grund terminieren und neugestartet werden) und Konfigurations-

möglichkeiten für Fehlerstrategien (für weitergehende Details und Konzepte siehe (Armstrong, 2003)).

## 2.3 Scala

Scala ist eine Hybridsprache, die sowohl objektorientiert, als auch funktional ist und seit 2001 entwickelt wird. Scala läuft auf der *Java Virtual Machine* (kurz JVM) und ist voll kompatibel zu Java. Alle bestehenden Java-Klassen, sowie Interfaces aus können direkt angesprochen und auch abgeleitet werden (Odersky, 2009).

Das Aktormodell ist kein Bestandteil der Sprache, aber seit Version 2.1.7 (2006) Bestandteil der Standardbibliothek von Scala. Die Implementierung orientiert sich sehr stark an Erlang und übernimmt die wesentlichen Charakteristika und Operationen (Haller und Odersky, 2009). Eine Möglichkeit einen Aktor zu definieren ist es, die Klase `scala.actors.Actor` abzuleiten und die Methode `act()` zu überschreiben (siehe Beispielcode).

Im Gegensatz zu Erlang kann die Aktorbibliothek in Scala allerdings nicht sicherstellen, dass Aktoren voneinander isoliert sind und ausschließlich durch asynchrone Nachrichten kommunizieren, da es in Scala-Anwendungen (wie bei Java) *shared memory* gibt. Wobei dazu gesagt werden muss, dass der von Scala propagierte, funktionale Stil sog. *immutable* (unveränderliche und daher auch bei parallelem Zugriff seiteneffektfreie) Objekte favorisiert.

Aus diesem Grund kann die Implementierung in Scala gut mit acedia verglichen werden, da sie dem Entwickler zwar die Möglichkeit gibt, seine Anwendung mit dem Aktormodell zu realisieren, aber es kann nicht ausgeschlossen werden, dass *shared memory* Zugriffe stattfinden.

Zur Implementierung der Aktoren in Scala wird auf die Threads der JVM zurückgegriffen, die letztlich auf native Threads des Betriebssystems abgebildet werden (in Erlang werden Prozesse nicht auf native Threads des Betriebssystem abgebildet, sondern die Verwaltung und Verteilung auf Hardwareressourcen übernimmt die Erlang VM). Jedem Aktor einen eigenen Thread zuzuweisen wäre bei einer hohen Anzahl an Aktoren ineffizient und bedeutet einen hohen Verwaltungsaufwand für das Betriebssystem (Haller und Odersky, 2009).

Um die Möglichkeit zu schaffen, Aktoren (ohne *Inversion of Control*[5] (IOC)) eventbasiert zu programmieren, gibt es neben dem an Erlang angelehnten `receive` eine zweite, eventbasierte Möglichkeit: `react`. Zu beachten ist, dass der Aktor beendet wird , sobald die Ausführung von `react` abgeschlossen ist (Haller und Odersky, 2006), wie das folgende Beispiel zeigt:

[5] *Inversion of Control* bezeichnet den Umstand, dass in eventbasierten Architekturen die Kontrolle beim Erzeuger eines Events und nicht bei dem Empfänger liegt, da der Erzeuger einen vom Empfänger definierten Callback ausführt, der in seinem Kontext – und nicht in dem des Empfängers – ausgeführt wird

```
val reactActor = new Actor {
  override def act() : Unit = {
    react {
      case _ : Int => println("Received an integer") ; act()
      case ("hello", what) => println("hello " + what) ; act()
      case "quit" => None // act nicht mehr aufrufen; Aktor beenden
      case y => println("Received: " + y) ; act()
    }
    println("Diese Zeile wird nie erreicht, auch nicht nach 'quit'")
  }
}
```

Scala benutzt *type inference*, um den Typ einer Variablen oder eines Ausdrucks zu ermitteln, ohne dass der Entwickler den Typ einer Variable explizit angeben muss. Ähnlich zu Erlang werden beim Nachrichtenempfang Pattern angegeben. Um z.B. die Nachricht nur auf ihren Typ zu prüfen, ohne sie an eine Variable zu binden, kann man statt eines Variablennamens "_" gefolgt von "`: [Typ]`" schreiben. Entspricht die Nachricht einem der angegebenen Pattern, wird der entsprechende Code ausgeführt.

Zur Verteilung der Aktoren über ein Netzwerk steht das Package `scala.actors.remote` zur Verfügung.

## 2.4 Related Projects

Nachfolgend werden drei Projekte vorgestellt, die in ähnlichem Kontext und mit ähnlicher Motivation wie acedia entstanden sind. Dabei handelt es sich nur um eine Auswahl verwandter Projekte und nicht um eine vollständige Liste.

### 2.4.1 Kilim

Kilim (aktuelle Version: 0.6) ist ein Aktorframework für Java, das im Vergleich zu Scala Aktoren weder auf JVM Threads abbildet, noch eine Eventbasierte Architektur benutzt. Statt dessen benutzt Kilim eine eigene Implementierung leichtgewichtiger Threads und stellt durch statische Analyse sicher, dass keine *shared memory* Zugriffe zwischen Aktoren stattfinden.

Dafür wurde ein Postcompiler (*Weaver* genannt) entwickelt, der den erzeugten Bytecode des Java-Compilers analysiert (u.A. auf Isolation der Aktoren) und Modifizierungen an ihm vornimmt. Um einen Aktor zu definieren, wird die Klasse `kilim.Actor` abgeleitet. Die ausführ-

baren Methoden des Aktors müssen mit der Annotation[6], `@pausable` gekennzeichnet sein, damit der Postcompiler die nötigen Modifikationen am Bytecode (CPS[7] Transformation) vornehmen kann. Als Nachricht können nicht beliebige Java-Objekte benutzt werden, sondern nur Subklassen von `kilim.Message`, die bei ihrer Verwendung ebenfalls für den Postcompiler mit Annotationen versehen werden müssen. Laut eigener Aussage ist der Nachrichtenversand bei Kilim 3x schneller als bei Erlang (Srinivasan und Mycroft, 2008).

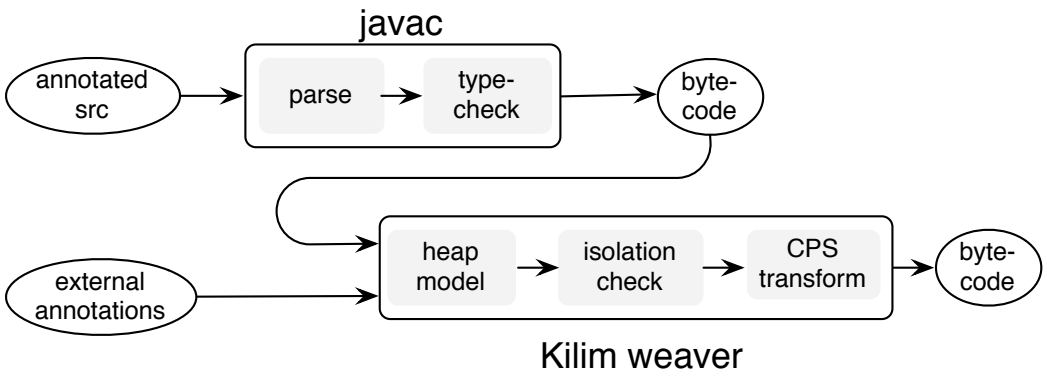


**Abbildung 1: javac** output post-processed by Kilim **weaver** [Srinivasan und Mycroft (2008)]

### 2.4.2 Retlang

Retlang (aktuelle Version: 0.4.2) ist eine C# Bibliothek zur Entwicklung verteilter Anwendungen, die sich am Aktormodell orientiert. Die sog. Fiber können als Aktoren aufgefasst werden.

Im Gegensatz zu Erlang oder Scala gibt es kein Pattern Matching. Nachrichten werden über Channel ausgetauscht, die jeweils genau einen Nachrichtentyp transportieren können. Fiber können sich an einem Channel mit einem Callback anmelden. Bei einer eingehenden Nachricht auf dem Channel wird der Fiber informiert und kann seinen Callback ausführen. Wie bei der eventbasierten Implementierung von Scala, wird der Callback in seinem eigenen Kontext ausgeführt.

[6] Annotationen sind in Zusatzinformationen im Quellcode, die der Entwickler z.B. zu Feldern oder Methoden schreiben kann und die vom Compiler (bzw. Postcompiler in diesem Fall) interpretiert oder zur Laufzeit der Anwendung ausgewertet werden können; siehe (Gosling u. a., 2005, Kap. 9.7)

[7] **C**ontinuation-**P**assing **S**tyle: Eine Methode, bei der der Kontrollfluss von Funktionen gesteuert werden kann, um sie z.B. unterbrechen und später an gleicher Stelle fortführen zu können; siehe: Appel und Jim (1989)

### 2.4.3 Theron

Theron (aktuelle Version 1.02) ist eine C++ Bibliothek, die nach dem Aktormodell implementiert wurde. Um einen Aktor zu definieren, leitet man die Klasse `Theron::Actor` ab. Jeder Aktor hat einen Ein- und einen Ausgabeport zum Nachrichtenversand. Sollen zwei Aktoren miteinander kommunizieren, müssen ihre Ports entsprechend verbunden werden. Als Nachrichten können nur Subklassen von `Theron::Message` verwendet werden.

Die Implementierung benutzt einen internen Threadpool, auf den die Aktoren verteilt werden, wenn sie eine Nachricht bearbeiten.

## 2.5 Transactional Memory

Transaktionaler Speicher (TM) kann viele Probleme einer *shared memory* Architektur lösen und entfernt die Notwendigkeit, Datenstrukturen per Hand mit Locks zu synchronisieren. Bei TM werden Schreiboperationen, ähnlich zu Datenbanksystemen, in Transaktionen organisiert, die erst nach einem erfolgreichen Commit wirksam werden. So kann atomar ein Wert ausgelesen, modifiziert und zurückgeschrieben werden. Beim Zurückschreiben des modifizierten Wertes schlägt das Commit fehl, wenn der Wert zwischenzeitlich geändert wurde (Herlihy und Moss, 1993).

Transaktionaler Speicher lässt sich in Software (STM), Hardware (HTM) oder als hybrides System (HyTM) implementieren und ist noch ein aktuelles Forschungsthema.

> “STM performs comparably to well-tuned lock code, but has a significant overhead at low processor counts.” (Saha u. a., 2006)

Um den Overhead aktueller STM Lösungen zu verringern und Skalierbarkeit und Performance weiter zu steigern, arbeiten Hersteller wie Intel und Sun derzeit an Hardwareunterstützung für TM. Erste praktische Ergebnisse eines solchen Systems sind zu finden bei Dice u. a. (2009).

Heute verfügbare STM-Systeme sind Intels “C++ STM Compiler” (ein C++ Compiler, der die Sprache um TM erweitert) und die funktionale Sprache Clojure (Hickey, 2008) (die wie Scala auf der JVM läuft), bei der TM ein fester Bestandteil der Sprache ist.

# 3 Diskussion

Das Aktormodell bietet eine Architektur für verteilte Anwendungen, die durch Isolation der parallelen Komponenten Seiteneffekte ausschließt und die Komplexität für den Entwickler reduziert.

Der Zustand der Anwendung ergibt sich aus dem aktuellen Zustand und Verhalten aller Aktoren und aller aktuellen Nachrichten, was es schwer macht einen globalen Zustand zu bestimmen. Konkrete Probleme können zudem auftreten, wenn ein Aktor mehr Nachrichten bekommt, als er verarbeiten kann oder ungewollte Nachrichten in der Mailbox verbleiben und nie gelöscht werden. Wie jeder anderer Buffer auch, kann eine Mailbox überlaufen. Dies kann entweder zu Nachrichtenverlust führen, wenn es eine maximale Kapazität gibt oder im schlimmsten Fall den gesamten verfügbaren Speicher des Systems aufbrauchen.

Aktoren sind eine Möglichkeit, verteilte Anwendungen zu schreiben, die sowohl lokal als auch über ein Netzwerk verteilt sein können. Diese können in vielen Fällen konventionelle Multithreading-Architekturen ersetzen, sind aber nicht in allen Anwendungsfällen verteilter Anwendungen das richtige Werkzeug.

Für parallelisierte Rechenoperationen existieren mathematische Modelle (üblicherweise in funktionalen Sprachen implementiert), um nebenläufige Prozesse zu beschreiben. So z.B. das Join-Calculus (Fournet und Gonthier, 1996) aus der Familie der $\pi$-Calculi (Milner, 1999). Für stark verteilte Operationen auf Clustern existieren Projekte, wie z.B. Googles MapReduce (Dean und Ghemawat, 2008).

## 3.1 Bewertung der Arbeiten

Erlang kann als Referenzimplementierung des Aktormodells bezeichnet werden. Die Sprache kennt keinen *shared memory* und ist speziell für verteilte, nebenläufige Anwendungen entwickelt worden. Dieser Implementierung am nächsten kommt Scala, wo ebenfalls Pattern Matching ein Bestandteil der Sprache ist und bei der Realisierung verteilter Anwendungen mit Aktoren Erlang auch in Struktur und Syntax sehr nahe kommt.

Die vorgestellten Projekte haben im Vergleich zu Erlang (und Scala) einen höheren Implementierungsaufwand für den Entwickler und sind nicht darauf ausgelegt auch über ein Netzwerk Nachrichten zu versenden. Bei Retlang und insbesondere bei Kilim liegt der Fokus auf Geschwindigkeit.

Die einzige C++ Bibliothek (Theron) ist nicht für die Realisierung großer verteilter Anwendun-

gen geeignet, da u.A. jegliche Möglichkeit fehlt auf den Ausfall einzelner Aktoren zu reagieren und das nicht pre-emptive Scheduling der Aktoren keine Möglichkeit hat Starvation zu erkennen oder zu verhindern. Eine weitere Möglichkeit das Aktormodell in C++ zu implementieren wird in Kafura u. a. (1992) beschrieben. Das Projekt ist aber nicht öffentlich zugänglich oder wurde eingestellt.

Transactional Memory ist eine interessante Technologie, um seiteneffektfreien Zugriff auf *shared memory* zu garantieren. Um vom Entwickler wirklich nutzbar zu sein und wartbaren Code zu erzeugen, muss TM Bestandteil der Sprache, wie z.B. bei Clojure, sein. Die Probleme heutiger multithreaded Anwendungen löst TM jedoch nur zum Teil, da z.B. die Synchronisierung der Threads immer noch vom Entwickler vorgenommen werden muss.

## 3.2 Einordnung der eigenen Arbeit

Insbesondere im C++ Umfeld gibt es bislang noch keine zufriedenstellende Möglichkeit, verteilte Anwendungen mithilfe des Aktormodells zu realisieren.

Mit den Möglichkeiten, die die nächste Sprachversion von C++ (Becker, 2009) (unter Entwicklung derzeit als C++0x) bietet und insbesondere die Erweiterung der Standardbibliothek (entwickelt unter dem Projektnamen “boost”) eröffnen sich viele neue Möglichkeiten für Entwickler, zu denen acedia ein Beitrag sein soll. Als Referenz für die weitere Entwicklung von acedia können die Implementierungen aus Erlang und Scala angesehen werden.

Des Weiteren beschäftigt sich die Arbeit mit den Möglichkeiten, die sich durch das Aktormodell in der Praxis bieten und wie es die Entwicklung verteilter Anwendungen in Zukunft beeinflussen und erleichtern könnte.

## 3.3 Ausblick

Die Entwicklung verteilter Anwendungen ist unerlässlich auf parallel laufender Hardware.

Das Aktormodell bietet eine theoretisch beliebig skalierbare Architektur für verteilte Anwendungen, die zudem einfacher zu implementieren und zu verstehen ist, als Implementierungen mit Threads.

Ob sich das Aktormodell als Softwarearchitektur für verteilte Anwendungen auch durchsetzen wird, ist derzeit nicht abzusehen, auch wenn viele Entwicklungen in diese Richtung gehen.

## Literatur

[Agha 1990] AGHA, Gul: Concurrent object-oriented programming. In: *Commun. ACM* 33 (1990), Nr. 9, S. 125–141. – ISSN 0001-0782

[Appel und Jim 1989] APPEL, A. W. ; JIM, T.: Continuation-passing, closure-passing style. In: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1989, S. 293–302. – ISBN 0-89791-294-2

[Armstrong 2003] ARMSTRONG, Joe: *Making Reliable Distributed Systems in the Presence of Software Errors*. 2003. – URL `http://erlang.org/download/armstrong_thesis_2003.pdf`

[Armstrong 2007a] ARMSTRONG, Joe: A history of Erlang. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. New York, NY, USA : ACM, 2007, S. 6–1–6–26. – ISBN 978-1-59593-766-X

[Armstrong 2007b] ARMSTRONG, Joe: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. – ISBN 193435600X

[Becker 2009] BECKER, Pete: *Working Draft, Standard for Programming Language C++*. 2009

[Corrêa 2009] CORRÊA, Fábio: Actors in a new "highly parallel"world. In: *WUP '09: Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*. New York, NY, USA : ACM, 2009, S. 21–24. – ISBN 978-1-60558-565-9

[Dean und Ghemawat 2008] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: simplified data processing on large clusters. In: *Commun. ACM* 51 (2008), Nr. 1, S. 107–113. – ISSN 0001-0782

[Dice u. a. 2009] DICE, Dave ; LEV, Yossi ; MOIR, Mark ; NUSSBAUM, Daniel: Early experience with a commercial hardware transactional memory implementation. In: *SIGPLAN Not.* 44 (2009), Nr. 3, S. 157–168. – ISSN 0362-1340

[Earle u. a. 2005] EARLE, Clara B. ; FREDLUND, Lars-Åke ; DERRICK, John: Verifying fault-tolerant Erlang programs. In: *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*. New York, NY, USA : ACM, 2005, S. 26–34. – ISBN 1-59593-066-3

[Fournet und Gonthier 1996] FOURNET, Cédric ; GONTHIER, Georges: The reflexive CHAM and the join-calculus. In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1996, S. 372–385. – ISBN 0-89791-769-3

[Gosling u. a. 2005] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification, Third Edition*. 3. Amsterdam : Addison-Wesley Longman, 2005. – 688 S. – ISBN 0321246780

[Haller und Odersky 2006] HALLER, Philipp ; ODERSKY, Martin: Event-Based Programming without Inversion of Control. In: *Joint Modular Languages Conference*, 2006 (Lecture Notes in Computer Science), S. 4–22

[Haller und Odersky 2009] HALLER, Philipp ; ODERSKY, Martin: Scala Actors: Unifying thread-based and event-based programming. In: *Theor. Comput. Sci.* 410 (2009), Nr. 2-3, S. 202–220. – ISSN 0304-3975

[Hansen 1973] HANSEN, Per B.: *Operating system principles*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1973. – ISBN 0-13-637843-9

[Herlihy und Moss 1993] HERLIHY, Maurice ; MOSS, J. Eliot B.: Transactional memory: architectural support for lock-free data structures. In: *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*. New York, NY, USA : ACM, 1993, S. 289–300. – ISBN 0-8186-3810-9

[Hickey 2008] HICKEY, Rich: The Clojure programming language. In: *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*. New York, NY, USA : ACM, 2008, S. 1–1. – ISBN 978-1-60558-270-2

[Kafura u. a. 1992] KAFURA, Dennis G. ; MUKHERJI, Manibrata ; LAVENDER, Gregory R.: ACT++ 2.0: A Class Library for Concurrent Programming in C++ Using Actors. Blacksburg, VA, USA : Virginia Polytechnic Institute && State University, 1992. – Forschungsbericht

[Meyers und Alexandrescu 2004] MEYERS, Scott ; ALEXANDRESCU, Andrei: C++ and the Perils of Double-Checked Locking. (2004)

[Milner 1999] MILNER, Robin: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999. – ISBN 0521658691

[Odersky 2009] ODERSKY, Martin: *The Scala Language Specification Version 2.7*. 2009

[Saha u. a. 2006] SAHA, Bratin ; ADL-TABATABAI, Ali-Reza ; JACOBSON, Quinn: Architectural Support for Software Transactional Memory. In: *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2006, S. 185–196. – ISBN 0-7695-2732-9

[Srinivasan und Mycroft 2008] SRINIVASAN, Sriram ; MYCROFT, Alan: Kilim: Isolation-Typed Actors for Java. In: *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*. Berlin, Heidelberg : Springer-Verlag, 2008, S. 104–128. – ISBN 978-3-540-70591-8

[Stonebraker 1986] STONEBRAKER, Michael: The case for shared nothing. In: *Database Engineering* 9 (1986)