Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht WS 09/10

Dominik Charousset

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

Die stetig zunehmende Verbreitung von Mehrkernsystemen macht Nebenläufigkeit zu einem wichtigen Kriterium für die Performance von Software (Haller und Odersky, 2009).

Obwohl Mehrkernsysteme im Alltag Einzug gehalten haben, ist parallel laufende Software keine Selbstverständlichkeit. Dies liegt unter anderem darin begründet, dass die Informatik auch heute noch von imperativen, C-artigen Sprachen wie Java, C# und C++ dominiert wird. Nebenläufigkeit wird in diesen Sprachen i.d.R. durch Multithreading erreicht.

Zugriffe in den Shared Memory müssen dabei mit *Locks* synchronisiert werden, was entweder eine sehr hohen Komplexität (*Fine-Grained Locking*) zur Folge hat oder zu Warteschlangenbildung führt, wenn ein Lock mehrere Objekte schützt (*Coarse-Grained Locking*). Dabei sind *Race Conditions*, *Deadlocks* und/oder *Lifelocks* de-facto unmöglich durch Testen auszuschließen (Hansen, 1973). Zudem ist viel Expertenwissen erforderlich um potentiell gefährliche Codeteile überhaupt identifizieren zu können, wie das Beispiel des *Double-Checked Locking Pattern* (Meyers und Alexandrescu, 2004) zeigt.

Das die Programmierung verteilter/nebenläufiger Software nicht so komplex und fehleranfällig sein muss, zeigen Beispiele aus der funktionalen Welt. Ein Programmiermodell, mit dem sich Nebenläufigkeit auf einer hohen Abstraktionsschicht ausdrücken lässt ist das *Aktormodell* (erstmals beschrieben von Hewitt u. a. (1973)).

Ziel dieses Masterprojektes ist die Entwicklung der Bibliothek *acedia*, die das Aktormodell für C++ implementiert, um die Programmierung nebenläufiger und verteilter Anwendungen zu erleichtern.

2 Das Aktormodell

Aktoren sind:

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

Gemäß dem Aktormodell können Aktoren zu ihrer Laufzeit weitere Aktoren erschaffen und haben – im Gegensatz zu Threads – keine gemeinsame Sicht auf einen Speicherbereich. Das Modell entspricht also einer *shared nothing* Architektur (Stonebraker, 1986).

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 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 bekannten anderen Aktoren kommunizieren kann. Dazu kann dem Aktor bei seiner Erschaffung eine Liste anderer Aktoren mitgegeben werden oder es muss eine Art von zentralem Namensdienst geben um bestimmte (besonders wichtige oder zentrale) Aktoren finden zu können.

Um auf das Ausfallen einzelner Aktoren reagieren zu können, bieten Implementierungen wie in Erlang oder in Scala die Möglichkeit, dass beim Ausfall eines Aktors mit ihm verbundene Aktoren ebenfalls beendet werden oder mit einer Nachricht über den Ausfall informiert werden. Durch diesen Mechanismus lassen sich ausfallsichere Anwendungen implementieren, die auf den Ausfall einzelner Komponenten (Aktoren) reagieren und diese ggf. neustarten können.

Dieses Verhalten wurde in acedia übernommen und auch ein zu Erlang vergleichbarer Namensdienst implementiert.

3 Vorarbeiten

Vorbereitend und begleitend zum ersten Teil des Masterprojekts erfolgte eine Literaturrecherche, sowie eine *Related Work* Recherche, in der Projekte, die in einem ähnlichen Kontext entstanden sind, mit acedia verglichen wurden.

Zusammenfassend lässt sich sagen, dass das Aktormodell die Grundlage für – in der Theorie – beliebig skalierbare Anwendungen sein kann und Programmierung im Aktormodell deutlich einfacher, verständlicher und sicherer ist als Programmierung mit Threads. Ein Alleinstellungsmerkmal des Aktormodells ist, dass es sich sowohl für lokal (innerhalb eines Prozesses auf mehrere Threads) verteilte Anwendungen, wie auch für im Netzwerk verteilte Anwendungen eignet.

Erlang gilt als die Referenzimplementierung des Aktormodells (Corrêa, 2009, S. 23) und ist auch Vorbild für acedia. Die wichtigsten vergleichbaren Arbeiten sind: Die Aktorbibliothek für Scala, Kilim (ein Framework für Java) und Retlang (eine Bibliothek für C# die sich grob am Aktormodell orientiert).

Im C++ Umfeld gibt es bislang nur Theron, das sich zwar am Aktormodell orientiert, viele wichtige Aspekte aber nicht implementiert. In Kafura u. a. (1992) wird zudem eine Möglichkeit beschrieben das Aktormodell in C++ zu implementieren, allerdings ist das Projekt entweder nicht öffentlich zugänglich oder wurde eingestellt.

Im ersten Teil des Masterprojekts wurden die Grundlagen für acedia geschaffen, wobei insbesondere die Realisierung des Pattern Matching[1] mit Templates im Vordergrund stand. Ein wichtiger Teil dieser Arbeit umfasste auch das zur Verfügung stellen und Auswerten von Typinformationen zur Laufzeit (in Java unter dem Begriff *Reflections* bekannt) und kann daher als Machbarkeitsstudie und Infrastrukturaufbau gesehen werden.

[1] In diesem Fall nicht das allgemeine Pattern Matching für beliebige Datenstrukturen, wie es aus funktionalen Sprachen bekannt ist, sondern die Prüfung eingehender Nachrichten auf Art und Inhalt gemeint.

4 Architektur

Im Folgenden wird die grundlegende Architektur der Bibliothek beschrieben. Alle hier vorgestellten Funktionen und Klassen liegen im Namespace `acedia`.

4.1 Speichermanagement

Für alle bereitgestellten Datenstrukturen, Nachrichten und Aktoren wurde eine Gargabe Collection mit *Reference Counting* (Detlefs (1990)) implementiert. Um Thread-Safety zu erreichen sind zudem alle Datenstrukturen (wie z.B. Strings und Tuple) entweder immutable (erlauben also nur lesenden Zugriff) oder nutzen *Copy on Write* (Javier und Guttman, 1995), wodurch z.B. Strings ohne sie kopieren zu müssen zwischen verschiedenen Aktoren ausgetauscht werden können, die *Call by Value*-Semantik jedoch erhalten bleibt.

Dadurch wird dem Nutzer von acedia das Speichermanagement über mehrere Threads (bzw. Aktoren) hinweg abgenommen. Die Bibliotheks-API nutzt zudem ausschließlich *Call by Value*-Semantiken, so dass Shared Memory Zugriffe durch den Nutzer verhindert werden.

4.2 Aktoren

Die Basisklasse aller benutzerdefinierten Aktoren ist die abstrakte Klasse `Actor`, deren rein virtuelle Methode `act()` mit dem Verhalten des Aktors überschrieben werden muss.

Gestartet werden Aktoren mit der Templatefunktion `spawn`, die ein Objekt der angegeben Klasse (die `Actor` als Basisklasse haben muss) erzeugt, dieses an eine `ActorRef` bindet und ins Scheduling (siehe Kap. 4.4) aufnimmt. `ActorRef` ist ein *Smart Pointer* (Alexandrescu, 2001, S. 157) für Aktoren, der keinen direkten Zugriff auf das Objekt zulässt. Alle Funktionen der Bibliothek arbeiten ausschließlich mit `ActorRef`-Argumenten, wodurch direkte Methodenaufrufe der Aktoren untereinander (und damit letztlich *Shared Memory* Zugriffe auf Aktoren) unterbunden werden. Aktoren werden automatisch gelöscht, sobald sie ihre Ausführung beendet haben und keine Referenzen mehr auf sie existieren.

Damit Aktoren, die zusammen arbeiten sollen sich finden können gibt es nach dem Vorbild von Erlang einen lokalen Namensdienst. Die Funktion `registerName` bindet einen Namen (String) an ein `ActorRef`-Objekt, das mit `whereisName` abgefragt werden kann. `registeredNames` gibt alle im System registrierten Namen zurück.

4.3 Netzwerkabstraktion

Um über das Netzwerk kommunizieren zu können muss eine Seite die Rolle des Servers übernehmen, zu dem sich dann beliebig viele Clients verbinden können.

4.3.1 Serverseite

Auf der Serverseite muss zunächst ein Aktor mit `publish` an einen Port gebunden werden (Abb. 1). Das System erzeugt einen speziellen Aktor (*Middle Man*, kurz MM), der die Kommunikation über diesen Port kapselt und bei eingehenden Nachrichten Proxys für die über das Netzwerk erreichbaren Aktoren erzeugt. Der MM übernimmt auch das Serialisieren und Deserialisieren von Nachrichten mithilfe der `boost` Bibliothek.

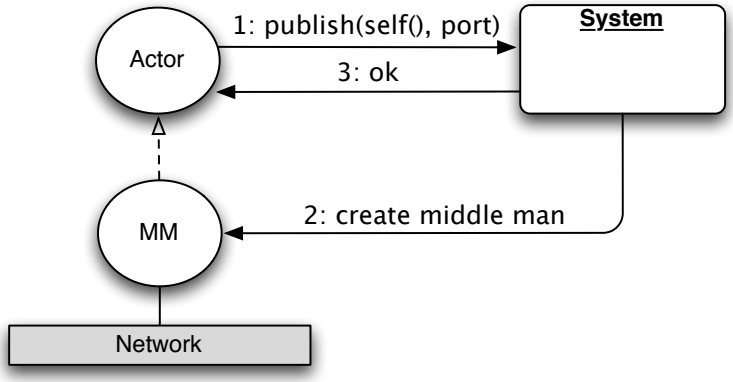

Abbildung 1: Freigabe eines Aktors im Netzwerk

Minimales Codebeispiel:

```
using namespace acedia;
class ExampleActor : public Actor { ... };
int main() {
  ActorRef exampleActor = spawn<ExampleActor>();
  PublishingResult res = publish(exampleActor, 1234);
  if (res.successful()) { ... }
  else { /* error handling */ }
  return 0;
}
```

4.3.2 Clientseite

Auf der Clientseite erhält man nach einer Erfolgreichen Verbindung eine Referenz (Proxy) für den freigegebenen Aktor mit `remoteActor`. Auch auf der Clientseite wird ein *Middle Man* gestartet, der für die Kommunikation über das Netzwerk verantwortlich ist.

Minimales Codebeispiel:

```
using namespace acedia;
int main() {
  pair<ActorRef, ConnectionError> res =
      remoteActor("localhost", 1234);
  if (res.first.isValid()) { ... }
  else { /* error handling */ }
  return 0;
}
```

4.3.3 Nachrichtenversand über das Netzwerk

Nachdem sich zwei Knoten verbunden haben wurden auf beiden Seiten Middle Man erzeugt, die die eigentliche Netzwerkkommunikation übernehmen. Für die Aktoren selbst ist es völlig transparent, ob sie eine Nachricht an einen Aktor im selben Prozess oder an einen Aktor über das Netzwerk senden.

Diese Abstraktion wird mit Proxies erreicht, die von den MM für Aktoren auf anderen Knoten angelegt werden. Aktoren und Proxies nutzen das selbe Interface (`ActorRef`, siehe Kap. 4.2), so dass ein Aktor mit einem Proxy auf die gleiche Weise kommuniziert. Empfängt ein Proxy eine Nachricht, so leitet er sie zu seinem MM weiter (Abb. 2).

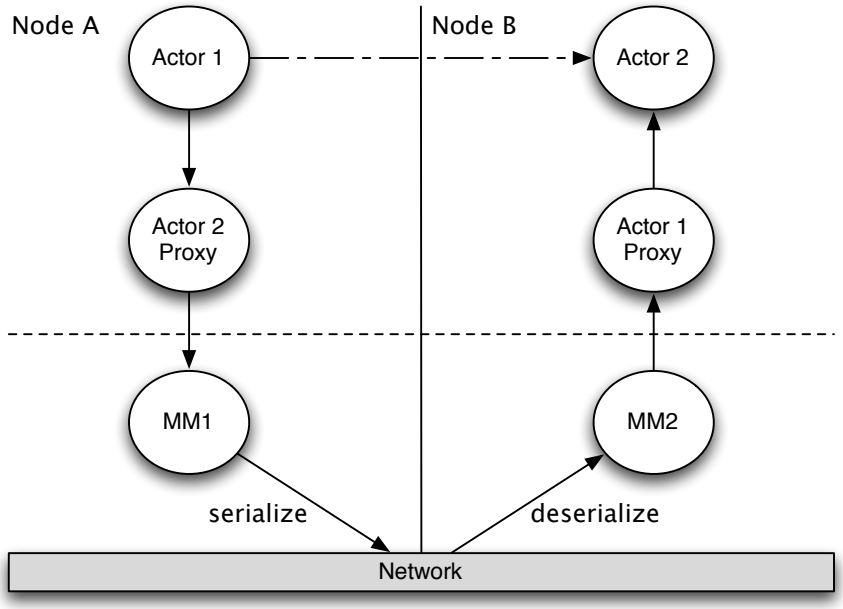

Abbildung 2: Netzwerkabstraktion

In dem verwendeten Beispiel aus Abb. 2 sieht `Actor 2` als Absender der Nachricht `Actor 1 Proxy`, so dass eine Antwort auf die empfangene Nachricht den umgekehrten Weg zu `Actor 1` geht. `MM1` und `MM2` sind zu keinem Zeitpunkt für lokale Aktoren (und damit für den Programmierer) sichtbar.

4.4 Scheduling

Aktoren wurden als *Lightweight Threads* oder auch *User Level Threads* mit dem zusätzlichen Zustand `detached` (Abb. 3) modelliert. Dazu wurde unter Mac OS X und Linux die *User Thread Context* API (Apple, 2002) und unter Windows die *Fibers* API (Microsoft, 2010) genutzt. Ein einmal im Zustand `detached` befindlicher Aktor kann nicht mehr in den normalen Zyklus (bestehend aus `ready`, `running` und `blocked`) aufgenommen werden. Dieser Zustand beschreibt Aktoren, die sich nicht für das kooperative Scheduling eignen.

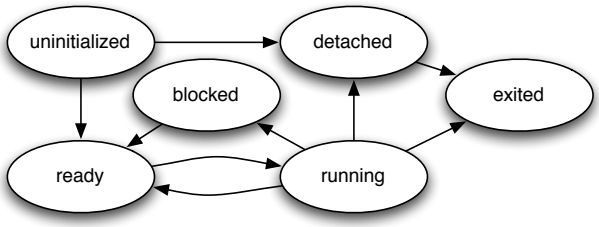

Abbildung 3: Lebenszyklus eines Aktors

Aktoren werden nicht eins zu eins auf systemeigene (native) Threads abgebildet. Der nicht-preemptive (FIFO-) Scheduler (Abb. 4) verteilt lauffähige Aktoren auf einen von ihm verwalteten Threadpool. Die Zuordnung zwischen Threads und Aktoren ist also eine N:M Beziehung, wobei N eine feste Größe (bestimmt zur Laufzeit durch die gefundene Hardware) und M die Anzahl aller aktuell lauffähigen Aktoren ist.

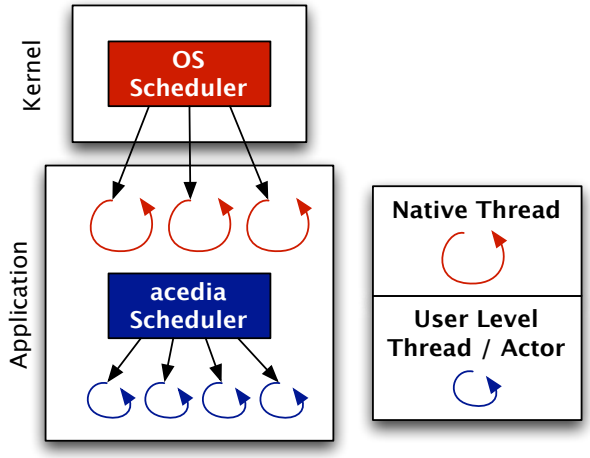

Abbildung 4: N:M Beziehung zw. Aktoren und Nativen Threads

Aktoren geben immer dann dem Scheduler die Möglichkeit einen anderen Aktor auszuführen, wenn sie die nächste Nachricht empfangen wollen. Dadurch entsteht ein implizites, kooperatives Scheduling.

Die Komplexität beim Erzeugen von Aktoren beschränkt sich auf das Allokieren eines Stacks und das Ausführen des Konstruktors, bzw. das Freigeben des benötigten Speichers nach Ablauf der Lebensdauer. Da weder beim Starten noch beim Löschen von Aktoren das Betriebssystem involviert ist, lassen sich mehr Aktoren starten, als auf der jeweiligen Plattform Threads möglich sind. Zudem fördert diese Architektur das Auslagern von kleinen Aufgaben (bei denen der Overhead einer Threaderzeugung in keiner Relation zur Laufzeit stünde) in kurzlebige Aktoren, wovon auf paralleler Hardware die Laufzeit insgesamt profitiert.

Kooperatives Scheduling birgt immer die Gefahr von Starvation. Dem Vorbild der Scala-Implementierung folgend werden daher Aktoren, die über einen langen Zeitraum ihren zugewiesenen Thread (aus dem Threadpool des Schedulers) blockieren *detached*, also in einen eigenen, nativen Thread ausgelagert und vom FIFO-Scheduling ausgeschlossen.

5 Aktueller Stand nach dem WiSe 09/10

Alle in Kap. 4 vorgestellten Komponenten wurden implementiert und acedia liegt aktuell in der ersten Beta Version vor. Die Beta läuft unter Linux, Mac OS X und Windows und ist als Open Source bei Source Forge unter `http://sourceforge.net/projects/acedia/` erreichbar.

6 Mögliches weiteres Vorgehen

In einer anschließenden Masterarbeit wäre der erste Schritt die aktuell im Betastadium befindliche Bibliothek systematischen Tests (Unit Tests, Code Reviews, ...) zu unterziehen und auf Thread-Safety und Deadlockfreiheit zu prüfen.

Des weiteren wären Testszenarien auf Systemen mit 2-8 (oder mehr) Cores für das Skalierungsverhalten des Schedulers interessant, da die Gesamtperformance von acedia-basierten Anwendungen maßgeblich vom Scheduler abhängig ist. Darüber hinaus würden sich Performancevergleiche mit anderen Implementierungen des Aktormodells (wie z.B. Erlang und die Aktorbibliothek in Scala) anbieten.

Am Ende der Masterarbeit würde ein stabiler Release stehen, der im Produktivbetrieb einsetzbar ist und die Entwicklung verteilter Anwendungen (mithilfe des in funktionalen Sprachen bereits bewährten Aktormodells) in C++ erleichtern würde.

Literatur

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

[Alexandrescu 2001] ALEXANDRESCU, Andrei: *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0-201-70431-5

[Apple 2002] APPLE: *user thread context*. 2002. – URL `http://developer.apple.com/Mac/library/documentation/Darwin/Reference/ManPages/man3/ucontext.3.html`

[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

[Detlefs 1990] DETLEFS, David L.: Concurrent garbage collection for C++. In: *IN ISMM '04: PROCEEDINGS OF THE 4TH INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT*, MIT PRESS, 1990, S. 37–48

[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

[Hewitt u. a. 1973] HEWITT, Carl ; BISHOP, Peter ; STEIGER, Richard: A Universal Modular ACTOR Formalism for Artificial Intelligence. (1973). – URL `http://citeseerx.ist.psu.edu/viewdoc/summary;jsessionid=2AB90D3F553A566B546FF79DD2AC7812?doi=10.1.1.77.7898`

[Javier und Guttman 1995] JAVIER, Francisco ; GUTTMAN, Joshua D.: *Copy on Write*. 1995. – URL `http://imps.mcmaster.ca/doc/copy-on-write.ps`

[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)

[Microsoft 2010] MICROSOFT: *Fibers*. 2010. – URL `http://msdn.microsoft.com/en-us/library/ms682661(VS.85).aspx`

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