Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht – SoSe 2009

Dominik Charousset

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

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 (Haller und Odersky, 2009, S. 1). Programmierung nebenläufiger Systeme bedeutet in Sprachen wie C++, 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: den *shared memory*.

Die Verantwortung für den Zugriff auf den *shared memory* obliegt dem Entwickler. Eine fehlerhafte Implementierung kann zu *Race Conditions*, *Deadlocks* und/oder *Lifelocks* führen. Das korrekte Verhalten des Systems ist zudem schwierig bis unmöglich zu verifizieren (Hansen, 1973). Selbst in der Praxis eingesetzte Pattern zur nebenläufigen Programmierung können zu Fehlverhalten führen (Meyers und Alexandrescu, 2004).

Das macht die Entwicklung nebenläufiger Software aufwändig und es ist viel Spezialwissen erforderlich um Komponenten *thread safe* zu schreiben, ohne durch die Synchronisierung zu viel Performance einzubüßen. Zum Synchronisieren von Programmteilen gibt es prinzipiell zwei Strategien: Fine-Grained Locking und Coarse-Grained Locking. Fine-Grained Locking ist sehr schwer in großen Anwendungen deadlockfrei zu implementieren und bedeutet insbesondere bei Änderungen/Erweiterungen der Software hohen Aufwand, während Coarse-Grained Locking große Probleme bei Skalierbarkeit und Performance hat.

Ein zunehmend populäres Programmiermodell aus dem Bereich der verteilten Programmierung ist das *Aktormodell*, das sowohl sicher (im Sinne von ausfallsicher, seiteneffekt- und deadlockfrei) als auch skalierbar ist und mit dem sich diese Projektarbeit beschäftigt.

2 Das Aktormodell

Aktoren sind nebenläufige Softwarekomponenten, die keine gemeinsame Sicht einen Speicherbereich haben. Sie kommunizieren durch asynchronen Nachrichtenaustausch miteinander und 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 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.

Programme, die nach dem Aktormodell programmiert wurden, sind in hohem Maße skalierbar. 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. Da Aktoren keine gemeinsamen Speicherbereiche haben und primär auf ihrem Stack arbeiten, können die Hardware-Ressourcen optimal genutzt werden.

3 Zielsetzung

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. Da Aktoren über Nachrichten kommunizieren, lassen sich Anwendungen prinzipiell sehr einfach über ein Netzwerk verteilen. Insbesondere lassen sich Anwendungen, die ursprünglich nur lokal verteilt waren sehr einfach nachträglich über ein Netzwerk verteilen (natürlich abhängig von der Implementierung, aber insb. in Erlang ist dies ohne Aufwand möglich).

Die funktionale Sprache Erlang ist derzeit die Referenzimplementierung für das Aktormodell. Allerdings lässt sich Erlang nur bedingt mit vorhandenen Bibliotheken und/oder bestehender Software nutzen, da Erlang nur auf der eigens entwickelten virtuellen Maschine läuft.

Wie in der Ausarbeitung zu Anwendungen 2 diskutiert, existieren alle bestehenden Lösungen entweder für sehr wenig verbreitete Plattformen (Erlang), erfordern das Lernen einer völlig neuen Sprache (Scala und Erlang) oder sind nur schwer mit bereits vorhandenem Code zu mischen, bzw. haben andere Limitierungen (Kilim, Retlang).

Insbesondere im C++ Umfeld gibt es bislang noch keine zufriedenstellende Möglichkeit, verteilte Anwendungen mithilfe des Aktormodells zu realisieren und bestehende Bibliotheken und/oder bestehenden Quellcode weiter zu nutzen.

Ziel des Projektes ist eine plattformneutrale Bibliothek zur Programmierung in C++ nach dem Aktormodell zu schaffen, die die Verteilung einer Anwendung lokal und über Netzwerk erlaubt und mindestens auf den drei derzeit am weitesten verbreiteten Betriebssystemen verfügbar ist: Linux, Mac OS X und Microsoft Windows. Aktueller Arbeitsname der Bibliothek ist “acedia”.

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 Entwicklung von acedia können die Implementierungen aus Erlang und Scala angesehen werden.

4 Architektur

Die Bibliothek lässt sich in die folgenden Komponenten aufteilen, die jedoch nicht zueinander unabhängig sind (z.B. benötigt sowohl der Nachrichtenaustausch als auch die Netzwerk-Abstratktion das Typsystem).

4.1 Aktorsystem

Aktoren leben in einem eigenen Kontext und kommunizieren ausschließlich über Nachrichten. Jeder Aktor hat eine eigene Mailbox, in der eingehende Nachrichten nach dem FIFO-Prinzip zwischengespeichert werden.

Um einen eigenen Aktor zu definieren, leitet der Benutzer der Bibliothek die abstrakte Klasse `Actor` ab und überschreibt die virtuelle Methode `act()`. Ähnlich wie ein Thread ist die Lebenszeit des Aktors vorbei, sobald `act()` vollständig abgearbeitet wurde. Zum Empfangen von Nachrichten stehen die Methoden `receive` und `tryReceive` (Empfangen mit Timeout) zur Verfügung.

Die Aktoren sind zueinander isoliert und können keine Methoden eines anderen Aktors aufrufen. Dies wird dadurch erreicht, dass keine direkte Referenz auf einen Aktor beim Nachrichtenversand benutzt wird, sondern Objekte der Klasse `ActorRef`, die mit der *Process-ID* (PID) in Erlang vergleichbar ist.

Aktoren werden mit der Funktion `spawn` gestartet, deren Rückgabewert ein `ActorRef`-Objekt ist, dass zur Kommunikation mit dem gestarteten Aktor benutzt wird. Das eigentliche `Actor`-Objekt wird nicht nach außen gegeben.

4.2 Typsystem

Empfangene Nachrichten werden zur Laufzeit gegen ein vorgegebenes Pattern geprüft. Zwar bietet C++ mit seinen Templates eine Möglichkeit zur generischen Programmierung, jedoch ist die Möglichkeit Typen zur Laufzeit zu prüfen nicht gegeben (der rudimentäre `typeof`-Operator ist ungenügend für diese Aufgabenstellung). Damit die Datentypen auch zur Laufzeit bekannt sind und abgefragt werden können, wird ein eigenes Typsystem benötigt. Des Weiteren wird das Typsystem zur Serialisierung/Deserialisierung benötigt.

Das Typsystem ist mithilfe zweier Klassen implementiert: `MetaClass` und `Any`.

Für jeden registrierten Typ gibt es genau ein `MetaClass`-Objekt, welches den Namen des Typs speichert und Objekte des Typs deserialisieren kann.

Einem Objekt der Klasse `Any` lässt sich jedes Objekt / jeder Wert zuweisen, dessen Datentyp dem Typsystem bekannt ist. Dabei wird der Wert selbst und die zugehörige `MetaClass`-Instanz des Typs gespeichert.

Zur Laufzeit lässt sich jede Instanz von `Any` nach ihrem aktuellen Typ und Wert fragen.

4.3 Netzwerk-Abstraktion

Aktoren können nicht nur mit Aktoren sprechen, die im gleichen Prozess oder auf der gleichen Maschine laufen. Nachrichtenaustausch ist Netzwerktransparent.

Aktoren auf anderen Rechnern im Netzwerk verhalten sich beim Nachrichtenversand so, wie es lokale Aktoren tun und sie sind über das gleiche Interface (`ActorRef`) ansprechbar. Repräsentiert das (`ActorRef`)-Objekt einen Aktor im Netzwerk, so verbirgt sich dahinter ein Proxy, der alle empfangenen Nachrichten an einen *Middle Man* (MM) weiterleitet, der die Netzwerkkommunikation kapselt und die Nachricht serialisiert/deserialisiert (Abb. 1, *Actor 1* sendet eine Nachricht an *Actor 2*, der über das Netzwerk erreichbar ist).

Jede Verbindung wird durch einen *Middle Man* überwacht, der für den Programmierer nicht sichtbar ist. Wird die Netzwerkverbindung unterbrochen oder ein Fehler tritt auf informiert der MM alle Proxys, die er erzeugt hat.

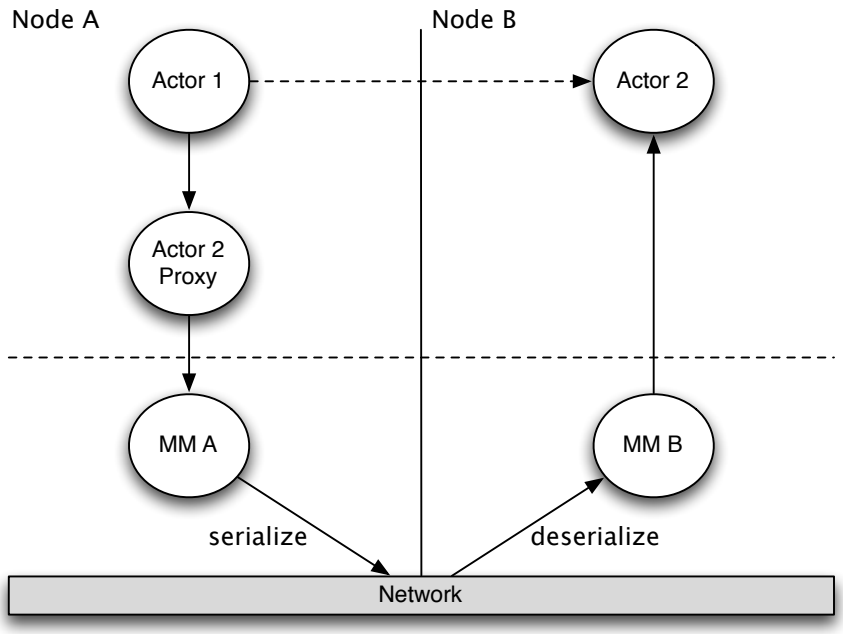

Abbildung 1: Nachrichtenversand über das Netzwerk

Beim Verbindungsaufbau muss ein Knoten als Server fungieren, damit andere sich mit ihm verbinden können. Hierfür muss ein Aktor mit `publish` an einem Port registriert werden, der eingehende Verbindungen verarbeitet (Abb. 2).

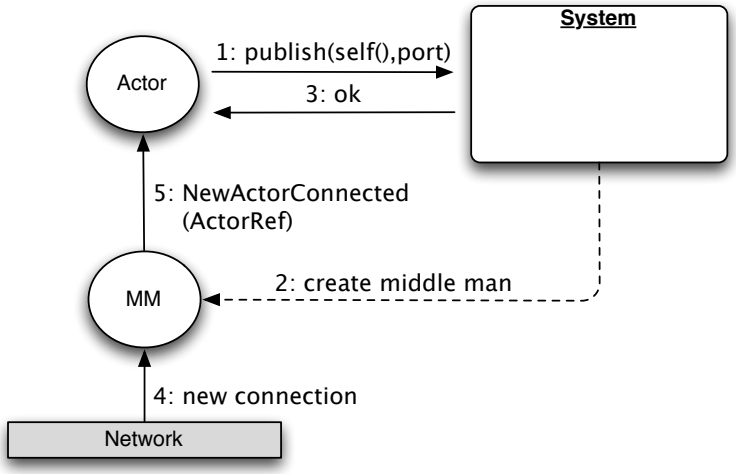

Abbildung 2: Registrierung eines Aktors mit `publish`

Der MM ist mit dem registrierten Aktor per Link verbunden. Stirbt der Aktor, gibt der MM den Port frei, damit ggf. ein anderer Aktor erneut registriert werden kann. Vom MM erschaffene Proxys bleiben solange gültig, bis sie ungenutzt sind oder die Verbindung unterbrochen wird.

Um zu gewährleisten, dass nur kompatible Knoten miteinander verbunden sind, wird die Versionsnummer der Bibliothek beim Verbindungsaufbau abgeglichen (Abb. 3).

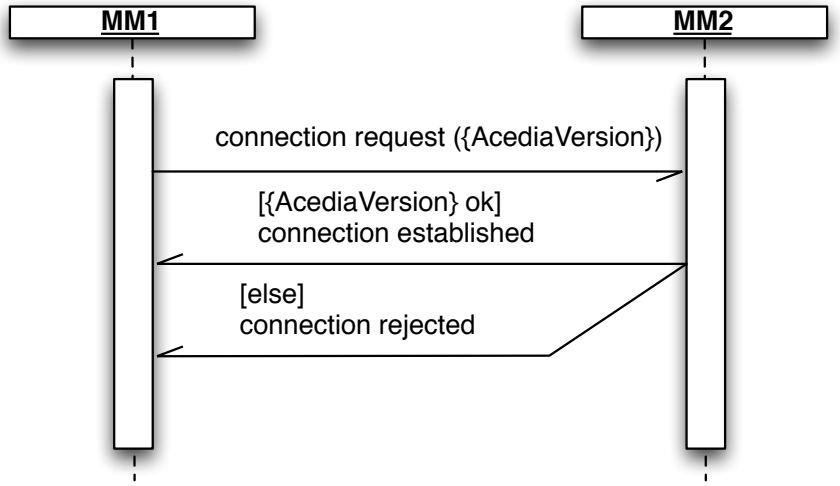

Abbildung 3: MM1 versucht eine Verbindung mit MM2 aufzubauen

Um eine Referenz zu einem mit `publish` im Netzwerk freigegebenen Aktor zu bekommen steht die Funktion `remoteActor` zur Verfügung, wie das folgende Codebeispiel zeigt:

```
Either<ActorRef, ConnectionError> res = remoteActor(host, port);
if (res.isLeft()) { // res is an ActorRef
  ActorRef ref = res.left();
  send(ref, "Hello remote actor!");
} else { // res is a ConnectionError
  ...
}
```

4.4 Scheduler

Jedem Aktor einen nativen Thread zuzuweisen ist ineffizient bei reaktiven Aktoren, die nur auf eingehende Nachrichten reagieren und darüber hinaus keinen eigenen Kontrollfluss haben. Die Verwaltung der nativen Threads obliegt zudem dem Betriebssystem (Kernel), d.h. die Zeit zum starten/beenden von Aktoren, sowie die Performance bei Kontextwechseln ist an die

Performance der Kernelthreads gebunden. Die Anzahl der gleichzeitig möglichen Aktoren ist plattformabhängig limitiert.

Um sowohl die verfügbaren Hardware-Ressourcen möglichst optimal auszunutzen und nicht an die Limitierungen von Kernelthreads gebunden zu sein, wird ein eigener Scheduler angestrebt, der für das Ausführen der Aktoren zuständig ist.

Der Scheduler startet so viele native Threads, wie für die jeweilige Plattform optimal ist und verteilt auf diesen die Laufzeit der Aktoren. Ein Aktor wird als *ready* markiert, wenn sich unbearbeitete Nachrichten in seiner Mailbox befinden. Jeder Aktor bearbeitet eine Nachricht und gibt danach die Priorität ab, um ggf. andere Aktoren ausführen zu lassen. Das abgeben der Priorität geschieht automatisch, wenn der Aktor im Scheduler läuft und mit `receive` (oder `tryReceive`) auf die nächste Nachricht wartet.

Um sowohl Aktoren mit eigenen Kontrollfluss als auch reaktive Aktoren optimal zu unterstützten, wird es dem Benutzer überlassen, ob er einem Aktor einen eigenen, nativen Thread zuweisen möchte oder der Scheduler benutzt werden soll (was der Default ist). Zudem kann der Scheduler Aktoren in eigene Threads auslagern, wenn diese zu lange blockieren (z.B. weil sie einen eigenen Kontrollfluss haben).

5 Aktueller Stand nach dem SoSe 09

Die Grundfunktionalität von acedia ist vorhanden und nutzbar.

Um dies zu evaluieren und erste Erfahrungen mit der neuen Bibliothek zu sammeln, wurde das virtuelle Bogenschießen, das im Rahmen von AW 1 im Wintersemester 08/09 entstand, auf acedia portiert. Die Software nutzte vorher Multithreading mit Locking und die Kommunikation zwischen den Komponenten funktionierte nach dem Signal/Slot Konzept (Gregor, 2001–2004).

Jede eigenständige Komponente ist nun durch einen Aktor implementiert und die Anwendung läuft nach der Portierung ohne Locks oder andere Mechanismen zur Synchronisierung. Bei der Kommunikation werden keine Signals/Slots mehr verwendet, sondern die Aktoren nutzen die in acedia vorgesehenen asynchronen Nachrichten. Bei der Bedienung der Software sind keine – insbesondere keine negativen – Unterschiede zur vorigen Version feststellbar.

Der aktuelle Stand implementiert jedoch noch nicht die vollständige Architektur. Der Fokus lag primär darauf, die Kernkomponenten zu implementieren, um die Bibliothek früh testen zu können. Das Typsystem und das damit verbundene Pattern Matching sollte für den Nutzer möglichst einfach und intuitiv sein. Eine Erlang-ähnliche Formulierung der Pattern ist so in C++ nicht möglich. Am nächsten würde der Erlang-Syntax ein `case`-Konstrukt (wie in Scala) kom-

men. Da aber für `case`-Ausdrücke nur konstante Integer-Werte laut C++ Sprachspezifikation erlaubt sind, war dies leider nicht möglich.

Das folgende Beispiel zeigt ein receive mit Pattern Matching in Erlang und zwei Möglichkeiten es in der aktuellen acedia-Version zu implementieren.

Erlang:

```
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 something~n")
  after 50 -> io:format("Received nothing~n")
end
```

acedia / C++:

```
ACEDIA_DECLARE_CASE_CLASS(Hello)
...
Message msg;
if (tryReceive(msg, 50)) {
  if (msg.match<int>) cout << "Received an integer" << endl;
  else if (msg.match<Hello, std::string>)
    cout << "Received: hello " << msg.valueAt<std::string>(1) << endl;
  else cout << "Received something" << endl;
}
else cout << "Received nothing" << endl;
```

Für das Atom "hello"(Atome sind konstante Singleton-Werte in Erlang, die klein geschrieben werden) wurde eine sog. *Case Class* definiert, die den gleichen Zweck erfüllt.

Das prüfen von Hand mit einem if-Konstrukt ist nur eine Möglichkeit eingehende Nachrichten zu prüfen. Angenommen der Aktor kennt die Funktionen `receivedInt`, `receivedString` und `receivedSomething`:

```
InvokingMessageProcessor imp;
imp.add(on<int>() >> receivedInt)
   .add(on<Hello,std::string>() >> receivedString)
   .add(others() >> receivedSomething);
if (!tryReceiveAndInvoke(imp, 50))
  cout << "Received nothing" << endl;
```

Nimmt die Funktion Argumente entgegen, werden diese Argumente automatisch mit dem Inhalt der Nachricht belegt. Dabei ist das letzte Element der Nachricht gleich dem letzten Element

der Funktion/Methode. Hat die Funktion weniger Argumente als das Pattern, werden die vorderen Elemente der Nachricht verworfen. Die Funktion `receivedString` könnte also so aussehen:

```
void receivedString(const std::string &str) {
  cout << "Received: " << str << endl;
}
```

Da die Klasse `Hello` nur zum strukturieren von Nachrichten dient und `Hello`-Objekte keine Informationen enthalten, kann es bei der Funktion weggelassen werden, da das Argument ohnehin überflüssig wäre.

Die Prüfung, ob eine Funktions-/Methodensignatur zum angegebenen Pattern passt, geschieht bei der Kompilierung. Dadurch werden Fehler frühzeitig gefunden und bei der Ausführung des Programms werden nicht mehr Typ-Überprüfungen vorgenommen als unbedingt nötig ist, was der Performance zugute kommt.

Die Mailbox eines Aktors nimmt `Message`-Objekte entgegen, die ihre Elemente in Instanzen von `Any` speichern. Damit stehen zur Laufzeit alle Typinformationen zur Verfügung. Beide oben gezeigten Codebeispiele sind in der aktuellen Version gültig und lauffähig.

Derzeit implementiert ist das Aktorsystem (Kap. 4.1) und das das Typsystem (Kap. 4.2). Um dem Typsystem einen eigenen Typ bekannt zu machen, muss er mit dem Makro `ACEDIA_ANNOUNCE` registriert werden. Zur Serialisierung von Objekten wird *boost*-Serialization benutzt (Ramey, 2004). Leider ist es in C++ nicht (wie z.B. in Java) möglich, beliebige Typen serialisieren zu lassen, weshalb der Entwickler den Quellcode zur Serialisierung und Deserialisierung nach wie vor selbst schreiben muss, was eine Limitierung der verwendeten boost-Bilbiothek ist.

Nur Teilweise implementiert (derzeit nicht lauffähig) und in Arbeit ist die Netzwerk-Abstraktion (Kap. 4.3). Der Scheduler (Kap. 4.4) ist derzeit nicht implementiert (siehe Kap. 6, weshalb aktuell jeder Aktor in einem eigenen Thread läuft.

6 Aufgetretene Probleme

Um den Scheduler zu implementieren werden sog. *Lightweight Threads* (oder auch *Userspace Threads* / *Fiber*) benötigt, da jeder Aktor einen eigenen Stack benötigt und um den Kontextwechsel zwischen den Aktoren von Hand im Scheduler regeln zu können. Unter Windows gibt es von Microsoft die *Fiber*-Bibliothek, unter Linux und Mac OS X gibt es die Unix-Schnittstelle für *Userspace Threads* und die Bibliothek *GNU pth* (Engelschall, 2006).

Bei ersten Laborversuchen mit der Unix-Schnittstelle lief der Code zwar unter Linux wie gewünscht, jedoch brach das identische Programm unter Mac OS X mit einem Speicherzugriffsfehler ab, obwohl die API unter beiden Systemen identisch ist.

In den jeweiligen Dokumentationen ist kein Hinweis darauf zu finden, dass sie sich unterschiedlich verhalten und weder in der Dokumentation von Apple noch in der von Linux findet sich ein Hinweis auf den Fehler. Mit GNU pth traten ähnliche Probleme auf und der Quellcode erzeugte auf den beiden System ein unterschiedliches Verhalten, was darauf hindeutet, dass pth intern die gleiche Schnittstelle nutzt.

Die Fiber-Bibliothek von Microsoft muss noch evaluiert werden und da es keine Bibliothek gibt, die die gewünschte Funktionalität auf allen Systemen zur Verfügung stellt, muss diese wohl zuerst entwickelt werden, bevor der Scheduler darauf aufbauend implementiert werden kann.

7 Ausblick

Das Skalierungsverhalten von acedia ist in der aktuell vorliegenden Version noch von der Implementierung der Kernelthreads auf der jeweiligen Plattform abhängig.

Bei vielen kurzlebigen und/oder reaktiven Aktoren ist die Performance von acedia noch verbesserungswürdig, wie Vergleichstests mit Scala und Erlang zeigen. Die Performance in diesem Bereich wird maßgeblich durch die zukünftige Implementierung des Schedulers bestimmt werden.

Vorrangig wird jedoch die Netzwerkunterstützung in acedia implementiert, da sich für Netzwerkverteilte Anwendungen nach dem Aktormodell viele interessante Anwendungsfälle ergeben, die als Beispielanwendungen für acedia interessant sind.

Ein solcher Anwendungsfall aus dem Bereich *Ambient Intelligence* ist ein verteilter Audioplayer in einer intelligenten Wohnung, bei der Sensoren erfassen, ob eine Person im Raum ist oder nicht. Ein zentraler Musikserver (der über mehrere Terminals aus verschiedenen Räumen gesteuert werden kann) spielt Musik nur in den Räumen ab, in denen sich auch eine Person befindet.

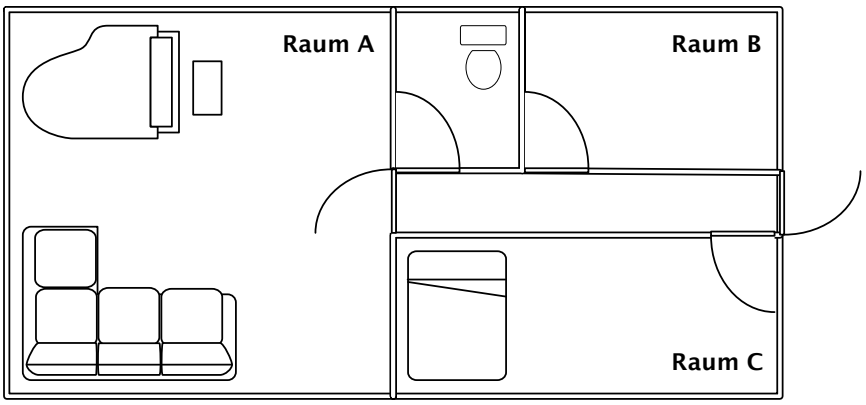

Abbildung 4: Szenario verteilter Audioplayer

Verlässt eine Person z.B. Raum C und betritt Raum A (Abb. 4), folgt ihr die Musik, ohne dass der Benutzer mit dem System interagieren muss.

Dieses Szenario ist bereits an der HAW auf Basis von iROS (eine Blackboard-Middleware) aufgebaut. Ein direkter Vergleich im Hinblick auf Implementierungsaufwand, Skalierbarkeit und Erweiterbarkeit zwischen einer Aktor- und einer Blackboard-basierten Lösung wäre sehr interessant und würde praxisnahe Erfahrungswerte für spätere Arbeiten mit sich bringen.

Literatur

[Becker 2009] BECKER, Pete: Working Draft, Standard for Programming Language C++, The C++ Standards Committee, 2009

[Engelschall 2006] ENGELSCHALL, Ralf S.: *GNU pth*. 2006. – `http://www.gnu.org/software/pth/`

[Gregor 2001–2004] GREGOR, Douglas: *Boost.Signals*. 2001-2004. – `http://www.boost.org/doc/libs/1_40_0/doc/html/signals.html`

[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

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

[Ramey 2004] RAMEY, Robert: *boost Serialization*. 2004. – `http://www.boost.org/doc/libs/1_40_0/libs/serialization/doc/index.html`

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