Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sebastian Meiser

Einsatz von Erlang in der Lehre - Eine Untersuchung für das Fach "Verteilte Systeme"

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sebastian Meiser

Einsatz von Erlang in der Lehre - Eine Untersuchung für das Fach "Verteilte Systeme"

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Christoph Klauck
Zweitgutachter: Prof. Dr.-Ing. Martin Hübner

Abgegeben am 29. September 2010

Thema der Bachelorarbeit
Einsatz von Erlang in der Lehre - Eine Untersuchung für das Fach "Verteilte Systeme"

Stichworte
Verteilte Systeme, Erlang, Lehre, Praktikum

Kurzzusammenfassung
In dieser Arbeit wird die Frage untersucht, ob sich die Programmiersprache Erlang für den Einsatz im Praktikum zur Vorlesung "Verteilte Systeme" eignet. Zusätzlich wird untersucht, inwieweit Erlang für das Vermitteln von Konzepten aus der Vorlesung geeignet ist. Es werden interessante Projekte rund um Erlang kurz vorgestellt und eine Betrachtung des n"otigen Aufwands beim Einsatz von Erlang durchgef"uhrt.

Title of the paper
Application of Erlang in Education - An analysis pertaining to the lecture "Verteilte Systeme"

Keywords
Distributed systems, Erlang, Practical course, Lecture

Abstract
This thesis examines whether the programming language "Erlang" is suitable for the use in a practical course in the lecture on "Verteilte Systeme". Furthermore it is discussed to which extend Erlang is useful to convey the concepts of the lecture. As part of the Documentation, projects concerning Erlang will be introduced and the requiered effort considered.

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit der Frage, ob sich die Programmiersprache Erlang für den Einsatz im Praktikum zur Vorlesung „Verteilte Systeme“ eignet. Als Einstieg in das Thema bietet dieses Kapitel einen kurzen Überblick über Erlang und die Zielsetzung dieser Arbeit sowie einen Blick auf die Gliederung.

1.1 Motivation

Erlang wurde Mitte der 1980er Jahre im Forschungslabor von Ericsson entwickelt. Die Sprache entstand aus der Aufgabe, für die nächste Generation der Telekommunikations-Produkte von Ericsson eine geeignete Programmiersprache zu finden. Nachdem das Team um Joe Armstrong, Robert Virding und Mike Williamson die zu der Zeit verfügbaren Sprachen getestet hatte, kamen sie zu dem Ergebnis, dass keine der Sprachen alle nötigen Eigenschaften aufwies. Also kreierten sie eine eigene Sprache: Erlang. Erlang wurde konstruiert, um verteilte, fehlertolerante Soft-real-time Systeme für den Telekommunikationsbereich zu entwickeln. In Erlang sind Einflüsse aus einer Reihe von Programmiersprachen eingegangen, u.a. ML, Miranda, ADA, Modula, Chill und PROLOG. Die erste Virtual Machine für Erlang wurde durch Joe Armstrong in PROLOG geschrieben. Erlang wurde durch Ericsson 1998 als Open Source freigegeben (Cesarini und Thompson (2009)).

Aus den Anforderungen, nach denen Erlang entwickelt wurde, ergeben sich auch die grundlegenden Charakteristiken, die Erlang interessant für die Vorlesung „Verteilte Systeme“ machen. Nebenläufige und verteilte Programmierung sind direkter Bestandteil der Sprache. Über asynchrones Message Passing ist eine Kommunikation zwischen Prozessen sehr einfach zu realisieren, auch über Computergrenzen hinweg in einem Netzwerk. Erlang wurde entworfen, um Systeme mit einer sehr niedrigen Downtime implementieren zu können, die Möglichkeit der Entwicklung von robusten Systemen ist also, wie die Konstrukte zur nebenläufigen und verteilten Programmierung, direkter Bestandteil der Sprache.

Generell findet Erlang in den letzten Jahren eine immer größere Verbreitung und wird bereits in vielen kommerziellen und Open Source Projekten eingesetzt. Zu den kommerziellen Produkten gehört neben Telekommunikationslösungen von Ericsson (wie das Erlang-

Flaggschiff-Projekt AXD 301[1]) z.B. das Backend-System des Facebook-Chat[2] [3] [4] oder das Backend-System von Klarna (früher Kreditor)[5] [6], einem Dienstleister für Zahlungslösungen im Internethandel. Bekannte Open Source Projekte sind Ejabberd[7] (ein XMPP Instant Messaging Server), CouchDB[8] (eine „Schema-less“ Dokument-orientierte Datenbank) und Riak[9] (verteilter Key-Value-Store), RabbitMQ[10] (eine AMQP Messaging Protokoll Implementierung) sowie mehrere Web-development Frameworks[11] [12] [13], Webserver(-Frameworks)[14] [15] und ein CMS[16]. Zu vielen dieser Open Source Projekte gehören Firmen, die Dienstleistungen rund um dieses Produkt anbieten, z.B. CouchOne[17], Basho[18] oder ProcessOne[19].

Auch im Hinblick auf die zukünftige Entwicklung von Prozessoren ist eine Sprache, in der nebenläufige Programmierung einen festen Platz hat, sehr interessant. Forschungsprojekte z.B. bei Intel [20] lassen auf künftige Generationen von Prozessoren schließen, bei denen nicht nur vier oder acht Kerne wie momentan, sondern 30, 40 oder sogar hunderte Prozessorkerne auf einem Chip vorhanden sind, wobei jeder einzelne Kern eher weniger Rechenleistung hat als heutige Kerne. Massiv nebenläufige Programme könnten einen Geschwindigkeitsvorteil erlangen und Erlangs Konzept der leichtgewichtigen Prozesse unterstützt diesen Ansatz direkt.

[1] http://www.erlang.se/publications/ericsson_review_axd301_1998012.pdf
[2] http://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf
[3] http://www.facebook.com/note.php?note_id=51412338919
[4] http://www.facebook.com/note.php?note_id=14218138919
[5] http://www.klarna.se/de/gewerbe/was-ist-klarna
[6] http://www.erlang-factory.com/upload/presentations/68/Kreditor.pdf
[7] http://www.ejabberd.im/
[8] http://couchdb.apache.org/
[9] http://www.basho.com/Riak.html
[10] http://www.rabbitmq.com/
[11] http://www.erlang-web.org/
[12] http://nitrogenproject.com/
[13] http://erlyweb.org/
[14] http://yaws.hyber.org/
[15] http://github.com/mochi/mochiweb
[16] http://zotonic.com/
[17] http://www.couch.io/
[18] http://www.basho.com/
[19] http://www.process-one.net
[20] http://techresearch.intel.com/articles/Tera-Scale/1421.htm

1.2 Zielsetzung

Um die Eignung von Erlang für den Praktikumsbetrieb zu untersuchen, gibt es mehrere Fragen, die durch diese Arbeit eine Antwort finden sollen. Dieser Abschnitt bietet eine Übersicht über die Aspekte, die in dieser Arbeit untersucht werden und was am Ende als Ergebnis festzuhalten ist.

Es gilt zunächst zu untersuchen, ob sich Erlang generell für die Lösung der Praktikumsaufgaben eignet, ob die Sprachmittel also ausreichen, die Aufgaben angemessen zu bearbeiten. Als weiterer wichtiger Punkt ist der Aufwand zu untersuchen, den eine Einarbeitung in Erlang von den Studierenden erfordern würde. In den Bereich des Aufwands fällt auch die Frage, inwieweit Erlang von der Lösung der eigentlichen Aufgabe ablenkt, wie schwer es also z.B. ist, das Laufzeitsystem aufzusetzen.

Hinausgehend über diese grundlegenden Fragen ist der Aspekt, wie gut sich Erlang eignet, Konzepte aus der Vorlesung im Praktikum vertiefen zu können. Wie lange braucht es also, bis man von der grundlegenden Einarbeitung zur Bearbeitung von Problemstellungen aus der Vorlesung kommt. An dieser Stelle soll auch ein Vergleich bzw. eine Gegenüberstellung mit dem bisher eingesetzten System stattfinden.

Am Ende der Arbeit soll eine Bewertung stehen, aus der ersichtlich wird, ob sich Erlang für den Einsatz in der Lehre eignet, hier im speziellen für die Vorlesung „Verteilte Systeme“. Je nach Ausgang der Untersuchung steht am Ende eine Auflistung der Punkte, die für oder gegen den Einsatz sprechen.

1.3 Aufbau der Arbeit

Im Kapitel Analyse werden die Lernziele dargestellt, die durch das Praktikum zur Vorlesung erreicht werden sollen sowie die Vorkenntnisse, die die Studierenden im fünften Semester haben. Das Kapitel Grundlagen gibt einen Einstieg in die Sprache Erlang, um den Rest der Arbeit verständlich zu halten. Im Abschnitt Konzeption wird die Herangehensweise an die Untersuchung beschrieben, welche Aufgaben gelöst werden und was für Erweiterungen es eventuell zu untersuchen gibt. Der Realisierungsteil beschreibt die Lösungen und den dafür benötigten Aufwand, sodass am Schluss, basierend auf diesen Ergebnissen, ein Fazit gezogen werden kann. Zusätzlich wird im Schlussteil noch auf interessante Technologien, Produkte und Bibliotheken aus dem Umfeld von Erlang hingewiesen, die eventuell als Grundlage für weitergehende Aufgaben für die Studierenden dienen können.

2 Analyse

Da diese Arbeit sich auf die Voraussetzungen, die im Rahmen der Vorlesung „Verteilte Systeme“ an der HAW Hamburg gelten, bezieht, werden in diesem Kapitel die Rahmenbedingungen der Vorlesung erläutert. Weiterhin werden die vom Professor gewünschten Lernziele, die durch das Praktikum zur Vorlesung erreicht werden sollen, dargestellt.

2.1 Rahmenbedingungen

Die Angaben zu den Rahmenbedingungen beziehen sich auf Studierende, die im Bachelor-Studiengang Angewandte Informatik an der HAW Hamburg nach der Prüfungsordnung aus dem Jahr 2008[21] (im folgenden PO2008) studieren.

Die Vorlesung „Verteilte Systeme“ findet nach Lehrplan im fünften Semester des Studiengangs „Angewandte Informatik“ statt. Der Umfang beträgt drei Semesterwochenstunden (SWS) für die Vorlesung sowie eine SWS für das Praktikum. Das Praktikum findet an vier Terminen mit einem Umfang von jeweils drei Stunden statt. Als Vorbereitungszeit sind für jeden Praktikumstermin je nach Komplexität der Aufgabe zwischen fünf und zehn Stunden einzuplanen. Als Vorbereitung auf jede Aufgabe können eineinhalb Stunden in der Vorlesung eingeplant werden. Eine separat angebotene Einführung vor dem ersten Praktikumstermin im Umfang von einmalig drei Stunden durch den Professor ist je nach Bedarf möglich.

Das Labor, in dem das Praktikum stattfindet, bietet ein Netzwerk aus Rechnern, auf denen jeweils ein Erlang-System installiert ist. Als Entwicklungsumgebungen stehen Eclipse, Netbeans sowie diverse Texteditoren zur Verfügung.

Wenn die Studierenden im fünften Semester angekommen sind, haben sie nach Lehrplan bereits einige Voraussetzungen, die eine Einarbeitung in Erlang erleichtern können. In den Vorlesungen und Praktika Programmieren I und II lernen sie Ruby und Java kennen, haben also grundlegende Erfahrung im Umgang mit Programmiersprachen und sequentiellen Programmiersprachenkonstrukten wie Schleifen und bedingte Verzweigungen. Es werden in Programmieren I teilweise schon Konzepte aus der funktionalen Programmierung erläutert. Im Praktikum zur Vorlesung Logik und Berechenbarkeit wird PROLOG eingesetzt, das

[21] http://www.informatik.haw-hamburg.de/fileadmin/Homepages/ProfPareigis/PO2008AI.pdf

einen großen Einfluss auf die Entwicklung von Erlang hatte. Rekursion ist in allen drei oben genannten Vorlesungen Thema und eine rekursive Herangehensweise sollte so bekannt sein. Listen und deren Verarbeitung über Rekursion ist ebenfalls aus der Arbeit mit PROLOG bekannt.

Da ein wichtiges Ziel dieser Arbeit eine Abschätzung des Aufwands zum Erlernen von Erlang ist, sollen an dieser Stelle noch die Vorkenntnisse des Autors dargestellt werden. Der Autor hat noch nach der Prüfungsordnung von 2004 studiert. Statt Ruby und Java in Programmieren I und II wurden Smalltalk und Java eingesetzt. PROLOG ist nicht aus der Vorlesung Logik und Berechenbarkeit, sondern aus der Vorlesung Intelligente Systeme aus dem sechsten Semester bekannt. Das Erlernen von Erlang fand erst im Rahmen dieser Arbeit statt, die Voraussetzungen hierbei waren also ähnlich wie bei den Studierenden, die nach der PO2008 studieren und ggf. Erlang im Rahmen der Vorlesung Verteilte Systeme kennen lernen. Ein Aspekt, der sich verkürzend auf die benötigte Zeit zur Lösung der Aufgaben in Erlang durch den Autor auswirken könnte, ist das bereits absolvierte Praktikum zur Vorlesung Verteilte Systeme. Die Struktur der Aufgaben ist demnach bekannt und eine Lösung in der Umgebung Java/C++/CORBA liegt bereits als Orientierung vor. Im Rahmen der Darstellung der Lösung und der dafür benötigten Zeit wird versucht, diesen Umstand in die Bewertung mit einfließen zu lassen.

2.2 Lernziele

Das Praktikum dient der Vertiefung von in der Vorlesung vermittelten Konzepten aus dem Bereich der verteilten Systeme. Die Einarbeitung in ein System zur programmiertechnischen Umsetzung dient also nur der Schaffung von Grundlagen, um eine Bearbeitung der Aufgaben zu ermöglichen und ist nicht die eigentliche Zielsetzung des Praktikums. Es gilt also zu untersuchen, in wieweit Erlang den Studierenden „im Weg steht“ bei einer Bearbeitung der Aufgaben. In diesem Zusammenhang gilt es Antworten auf folgende Fragestellungen zu finden:

- Wie hoch ist der Einarbeitungsaufwand in die Programmiersprache Erlang?
- Wie kompliziert ist es, eine Entwicklungsumgebung für die Arbeit mit Erlang aufzusetzen?
- Ist es den Studierenden möglich, zuhause an einer Lösung zu arbeiten?
- Wie schwer ist es, eine Kommunikation im Netzwerk aufzubauen?

Zu den Konzepten, die in der Vorlesung vorgestellt und im Praktikum vertieft werden sollen, gehören z.B. RPC Remote Procedure Call und Message Passing als Möglichkeiten der Interprozesskommunikation. Inwieweit werden diese in Erlang direkt abgebildet bzw. wie schwer ist eine manuelle Umsetzung? Gibt es Bibliotheksfunktionen für die Umsetzung?

Interessant im Zusammenhang mit der Vorlesung ist auch die Interoperabilität von Erlang. Wie schwer ist es also, mit anderen Programmiersprachen zu kommunizieren? Welche Möglichkeiten gibt es?

Das Thema Fehlerbehandlung sollte sich ebenfalls im Praktikum widerspiegeln. Welche Fehler können in der Interprozesskommunikation auftreten? Welche Möglichkeiten bietet Erlang, um mit Fehlern umzugehen? Gibt es Unterstützung durch Bibliotheken?

2.3 Aktuelle Umgebung

Momentan wird im Praktikum auf eine Kombination aus Java und C++ gesetzt. Die einzelnen Komponenten kommunizieren mit Hilfe von CORBA. Als CORBA-Implementierungen werden Java IDL für die Java Seite und Mico für die C++ Seite eingesetzt. Diese sind im Labor der HAW installiert, eine Bearbeitung der Aufgaben außerhalb des Labors setzt allerdings die Installation von Mico voraus. Aus eigener Erfahrung kann der Autor sagen, dass die Installation und das „zum Laufen bringen“ von Mico nicht immer trivial ist und ein Großteil der für den C++ Teil der ersten Aufgabe benötigten Zeit auf die Lösung von Problemen mit C++ und/oder Mico entfällt und nicht auf die zu implementierende Funktionalität.

3 Grundlagen

Um einen Überblick über den Sprachumfang von Erlang und damit auch erste Hinweise auf die benötigte Einarbeitungszeit zu gewinnen, bietet dieser Abschnitt eine grobe Einarbeitung in die Sprache Erlang. Gleichzeitig wird damit auch einem Leser ohne Vorwissen in Erlang das Verständnis der im Kapitel Realisierung vorgestellten Lösungen ermöglicht. Es werden nur Kernkonstrukte vorgestellt, die für eine grundlegende Lösung der Aufgaben notwendig sind. Weitergehende Konstrukte und Konzepte, z.B. aus der OTP-Bibliothek oder die Integration mit anderen Programmiersprachen, werden bei Bedarf im Rahmen der Vorstellung von möglichen Erweiterungen der Aufgaben erläutert (siehe Kapitel 4). Dieses Kapitel basiert auf den einführenden Büchern Armstrong (2007b) und Cesarini und Thompson (2009). Eine gute Einführung im Internet findet sich z.B. bei „Learn You Some Erlang for Great Good!“[22].

3.1 Grundlegende Sprachmittel

3.1.1 Arithmetik

In Erlang gibt es zwei Typen, um Zahlen zu repräsentieren, Fließkommazahlen (im weiteren Verlauf Floats) und Ganzzahlen (Integers). Ein Integer kann dabei beliebig groß werden (bzw. solange noch Speicher verfügbar ist). Die Genauigkeit von Floats entspricht der 64-bit Repräsentation des IEEE 754-1985 Standards. Die Reihenfolge der Auswertung der Rechenoperatoren entspricht der aus der Mathematik bekannten und kann durch Klammersetzung beeinflusst werden. Es stehen folgende Operatoren zur Verfügung:

- * : Multiplikation
- / : Division von Floats
- `div` : Integer Division
- `rem` : „Rest“ bei der Integer Division
- + : Addition

[22] http://learnyousomeerlang.com/

- - : Subtraktion

Zu beachten ist dabei, dass bei der Division mit / das Ergebnis immer ein Float ist. Werden also zwei Integer mit / dividiert, wird das Ergebnis dennoch ein Float sein.

3.1.2 Typen

Neben den für die Arithmetik verwendeten Typen zur Repräsentation von Zahlen gibt es noch eine Reihe anderer Typen, die in diesem Abschnitt kurz vorgestellt werden, namentlich: Atoms, Booleans, Tuples, Listen, Strings.

Atoms sind konstante Werte, die für sich selbst stehen. Sie werden ähnlich eingesetzt wie Konstanten oder Aufzählungstypen in Sprachen wie Java oder C oder Symbole in Ruby. Auf Atoms können nur Vergleichsoperationen durchgeführt werden. Ein Atom startet mit einem kleinen Buchstaben oder wird durch einfache Anführungszeichen umschlossen. Beispiele für Atoms:

- `konstante`
- `beta45_omega`
- `info@test`
- `'Alles innerhalb der Anführungszeichen  344 f£@ dsad'`

Eigentlich gibt es keinen eigenen Typ, um **boolesche Werte** darzustellen. Stattdessen werden die Atoms `true` und `false` eingesetzt, um das Ergebnis von Tests darzustellen, beispielsweise wird `1==2` zu `false` ausgewertet. Die logischen Operatoren umfassen:

- `and`
- `andalso` Wertet das zweite Argument nur aus, wenn das erste nicht schon `false` ergab
- `or`
- `orelse` Wertet das zweite Argument nur aus, wenn das erste nicht schon `true` ergab
- `xor`
- `not`

Die beiden Operatoren `andalso` und `orelse` sind sogenannte „Short-circuit boolean expressions“ und bieten eine Optimierungsmöglichkeit. Während die normalen Operatoren `and` und `or` in jedem Fall beide Argumente auswerten, folglich auch potentiell rechenintensive Berechnungen durchführen, werten die beiden Erstgenannten ihren zweiten Operanden nur aus, wenn dies zur Bestimmung des Wahrheitswertes der Aussage nach Auswertung des ersten Operanden noch notwendig ist.

Bei den Vergleichsoperatoren muss man besonders auf den Gleichheitsoperator achten:

- == Gleichheit
- /= Ungleichheit
- =:= Exakte Gleichheit
- =/= Exakte Ungleichheit
- =< Kleiner gleich
- < Kleiner als
- > Größer als
- >= Größer gleich

Der Unterschied zwischen == und =:= (bzw. /= und =/=) besteht im Vergleich von Zahlen. Wenn man Floats mit Integern vergleicht, führt der Operator == bzw. /= eine Umwandlung des Integer zu einem Float durch, sodass `1 == 1.0 true` zurückgibt, während `1 =:= 1.0 false` ergäbe. Man muss sich hier also die Semantik klar machen und den passenden Operator auswählen.

Tuples werden eingesetzt, um eine Sammlung einer festgelegten Anzahl an Werten zu repräsentieren. Die einzelnen Werte müssen nicht vom selben Typ sein. Tuples werden von {} umschlossen, die einzelnen Werte durch Kommata getrennt. Nach Konvention wird, falls der erste Wert des Tuples ein Atom ist, dieses als Tag bezeichnet und damit die Bedeutung des Tuples gekennzeichnet, z.B. `{person, 'Sebastian', 'Meiser'}`.

Listen in Erlang sehen denen in PROLOG sehr ähnlich, was bei der Geschichte von Erlang nicht verwundert. Sie werden durch [] umschlossen, ihre einzelnen Bestandteile durch Kommata getrennt. Auch Listen können Elemente unterschiedlicher Typen enthalten. Der Operator | trennt wie in PROLOG Kopf und Restliste, mit ++ und -- werden Listen zusammengeführt bzw. voneinander subtrahiert. Erlang hat keinen eigenen Typen für **Strings**, diese werden in Erlang intern als Listen von Integern repräsentiert, wobei die Integer die Ascii-Werte der einzelnen Zeichen sind. So ist der String `"Hello World"` eigentlich die Liste `[72,101,108,108,111,32,87,111,114,108,100]`. Dies macht die Arbeit mit Strings teilweise aufwendiger als aus Sprachen wie z.B. Ruby gewohnt.

3.1.3 Variablen und Pattern Matching

Variablen in Erlang funktionieren anders als aus Sprachen wie Ruby und Java bekannt ist. Erlang hat sogenannte „Single Assignment Variables“, einer Variable kann also nur einmal ein Wert zugewiesen werden. Variablennamen starten wie in PROLOG mit einem Großbuchstaben. Um einer Variablen einen Wert zuzuweisen, wird der Operator = verwendet.

Dies ist der Pattern Matching Operator, der im Fall einer ungebundenen Variablen (also einer Variablen, der noch kein Wert zugewiesen wurde) wie ein Zuweisungsoperator wirkt.

Pattern Matching kommt ebenfalls bei der Verarbeitung von Tuples und Listen zum Einsatz um z.B. einzelne Werte aus diesen zu extrahieren. Nach dem folgenden Ausdruck stehen in den Variablen X und Y die einzelnen Koordinaten des Punktes, in diesem Fall wird X also den Wert 4 und Y den Wert 5 haben: `{point, X, Y} = {point, 4, 5}`.

3.1.4 Programmstrukturierung

Module und **Funktione**n dienen der Strukturierung der Programme. Ein **Modul** enthält eine Reihe an Funktionen, im Normalfall funktionell zusammengehörende Funktionen. So enthält das Modul `lists` beispielsweise nur Funktionen für die Verarbeitung von Listen. Ein Modul wird in einer Datei mit der Endung `.erl` gespeichert und am Anfang stehen die Anweisungen `-module(modulname)` und `-export([Liste mit öffentlich zugänglichen Funktionen])`. Über die `export` Anweisung kann also gesteuert werden, welche Funktionen von außerhalb des Moduls erreichbar sind.

Funktionen bestehen aus Kopf und Rumpf, getrennt durch `->`. Im Kopf steht der Funktionsname und ein Pattern, das bestimmt, ob der Funktionsrumpf aufgerufen wird. Im Rumpf steht dann eine Sequenz von Anweisungen. Eine Funktion kann mehrere Rümpfe haben und über Pattern Matching im Kopf der Funktion wird ähnlich wie in PROLOG bestimmt, welcher Rumpf aufgerufen wird. Eine Funktion mit zwei Rümpfen könnte z.B. so aussehen:

```
total([{kosten, Wert}, Restliste]) ->
                          Wert + total(Restliste);
total([]) -> 0.
```

Funs sind anonyme Funktionen. Sie können an Variablen gebunden werden und dann wie normale Funktionen benutzt werden wie im folgenden Beispiel:

```
Double = fun(X) -> 2*X end.
Double(2).
```

Funs können genau wie Funktionen mehrere Funktionsrümpfe haben, die Syntax ist dieselbe.

Da Erlang eine funktionale Sprache ist, können Funktionen und Funs als Argumente an Funktionen übergeben werden oder auch als Rückgabewert zurückgegeben werden. In diesem Zusammenhang spricht man dann von „higher-order functions“. Eine verbreitete Funktion ist z.B. `lists:map(F,L)`, welche die Funktion oder Fun F auf jedes Element der Liste L anwendet und die so erzeugte Liste zurückgibt.

An dieser Stelle ein kurzer Einschub zu den verschiedenen Trennzeichen in Erlang, die am Anfang für Verwirrung sorgen können. Mit Kommata werden Argumente in Funktionen sowie Elemente in Listen, Tuples und in Mustern getrennt. Der Punkt, von einem Whitespace gefolgt, trennt komplette Funktionen und Ausdrücke voneinander. Das Semikolon trennt einzelne Klauseln, z.B. unterschiedliche Rümpfe innerhalb einer Funktion oder in den später erwähnten Blöcken `if, case, try..catch` und `receive`.

3.1.5 Rekursion

Rekursive Funktionen finden in Erlang sehr häufige Anwendung. An der oben dargestellten Funktion sieht man bereits, dass in Erlang ähnlich wie in PROLOG Listen meistens über Rekursion bearbeitet werden. Bei der nebenläufigen und verteilten Programmierung spielen rekursive Funktionen ebenfalls eine große Rolle. Das Prinzip der Rekursion ist den Studierenden bereits aus den Vorlesungen PR1 und PR2 sowie Logik bekannt und wird hier nicht näher erläutert.

3.1.6 Programmfluss

Neben der Verzweigung über Pattern Matching in Funktionsköpfen gibt es noch aus anderen Programmiersprachen bekannte Konstrukte wie `if` und `case` Anweisungen. In diesem Zusammenhang spielen so genannte Guards eine wichtige Rolle, die neben dem Einsatz in den im letzten Satz genannten Konstrukten auch in Funktionsköpfen zum Einsatz kommen können. Guards erweitern die Möglichkeiten des Pattern Matching um weitere Tests. In Funktionsköpfen werden Guardsequenzen durch das Schlüsselwort `when` eingeleitet, einzelne Teile einer Guard werden durch ein Semikolon getrennt. Guards müssen zu `true` oder `false` evaluiert werden können und dürfen nur einen eingeschränkten Satz an Konstrukten enthalten (siehe S. 56 Armstrong (2007b)).

Ein `case` Block hat den folgenden Aufbau:

```
case Ausdruck of
  Pattern1 [when Guard1] -> Sequenz_von_Ausdrücken_1;
  Pattern2 [when Guard2] -> Sequenz_von_Ausdrücken_2;
  ...
  PatternN [when GuardN] -> Sequenz_von_Ausdrücken_N
end.
```

Ein `if` Block hat den folgenden Aufbau:

```
if
  Guard1 -> Sequenz_von_Ausdrücken_1;
  Guard2 -> Sequenz_von_Ausdrücken_2;
  ...
  GuardN -> Sequenz_von_Ausdrücken_N
end.
```

Hier ist zu beachten, dass in einem `if` Ausdruck mindestens eine Guard zu `true` evaluieren muss. Häufig wird deshalb als letzte Guard das Atom `true` gesetzt, um sicherzustellen, dass diese Sequenz ausgeführt wird, falls alle Guards vorher zu `false` evaluieren. Meistens findet man `case` Blöcke in Erlang Programmen, der `if` Block findet auf Grund des genannten Verhaltens weniger Anwendung.

3.2 Fortgeschrittene Konstrukte

Dieser Abschnitt gibt einen Überblick über die Sprachkonstrukte von Erlang, mit denen fortgeschrittene Anwendungen entwickelt werden können. Es werden jeweils nur die grundlegenden Elemente in den Bereichen nebenläufige und verteilte Programmierung sowie Fehlerbehandlung vorgestellt. Tiefer gehende Details werden bei Bedarf im Rahmen des Kapitels 5 erläutert.

3.2.1 Nebenläufige Programmierung

Nebenläufige Programmierung in Erlang basiert auf den drei Basis-Operationen **spawn**, **send** und **receive**.

Mit `Pid = spawn(Fun)` wird ein neuer Prozess erzeugt, der parallel zum erzeugenden Prozess läuft und Fun evaluiert, wobei spawn den Process identifier (Pid) des neuen Prozesses zurückgibt. Über den in der Variablen Pid gespeicherten Process identifier können nun Nachrichten an den Prozess gesendet werden. Hierfür wird der send-Operator ! eingesetzt. Mit `Pid ! Message` wird Message an den Prozess mit dem Identifier Pid gesendet. Das Verschicken von Nachrichten erfolgt asynchron, der abschickende Prozess läuft also direkt mit den nachfolgenden Anweisungen weiter. Um Nachrichten zu empfangen wird ein `receive` Block definiert:

```
receive
  Pattern1 [when Guard1] -> Sequenz_von_Ausdrücken_1;
  Pattern2 [when Guard2] -> Sequenz_von_Ausdrücken_2;
  ...
  PatternN [when GuardN] -> Sequenz_von_Ausdrücken_N
```

```
after Time ->
  Sequenz_von_Ausdrücken
end
```

Nachrichten werden beim Empfang in einer Queue gespeichert und es wird versucht, die Nachricht gegen die Muster im `receive` Block zu „matchen“. Als Nachrichten werden häufig Tuple in der Form `tag, Data` gesendet, um ein einfaches Pattern Matching anhand des Bezeichners `tag` zu ermöglichen. Der `after` Block ist optional und ermöglicht die Angabe eines Timeout Wertes. Die Anweisungssequenz im `after` Block wird ausgeführt, wenn in einem durch den Timeout Wert angegebenen Zeitintervall keine Nachricht beim Prozess eintrifft.

Neben dem direkten Gebrauch der Pids besteht auch die Möglichkeit, einen Prozess unter einem Namen zu registrieren und in der Folge Nachrichten an diesen Namen anstatt der Pid zu schicken. Eine Registrierung erfolgt über die Funktion `register(Name, Pid).`

Erlang Prozesse werden häufig als leichtgewichtig bezeichnet. Sie gehören nicht zum zugrundeliegenden Betriebssystem, sondern werden in der Erlang Virtual Machine verwaltet und bringen weniger Overhead mit sich als native Prozesse des Betriebssystems. Auch das Scheduling der Prozesse wird durch die VM geregelt.

3.2.2 Verteilte Programmierung

Für die verteilte Programmierung mit Erlang gibt es mehrere Möglichkeiten, wobei ich mich in dieser Arbeit auf „Distributed Erlang“ beschränken werde. Es gibt daneben z.B. die Möglichkeit über TCP zu kommunizieren, auch ein CORBA ORB gehört zum Lieferumfang der Standarddistribution. Distributed Erlang bietet die Möglichkeit, die im vorangegangenen Abschnitt vorgestellten Techniken des Message Passing über Computergrenzen hinweg einzusetzen oder auf entfernten Systemen Prozesse starten zu lassen.

Ein verteiltes Erlang-System besteht aus mehreren miteinander kommunizierenden Nodes (Node ist die Bezeichnung für ein Erlang Laufzeit System). Distributed Erlang eignet sich nur für den Einsatz in einer sicheren Umgebung in der jedem Teilnehmer vertraut wird, da jeder Node auf jedem anderen verbundenen Node alle Operationen ausführen darf. Der einzige Schutz ist der „magic cookie“. Damit zwei Erlang Nodes miteinander kommunizieren können, müssen sie denselben „magic cookie“ haben. Dieser wird entweder über die Datei `.erlang.cookie` im Heimatverzeichnis des Anwenders, über einen Kommandozeilenparameter oder über die Funktion `set_cookie(node(), Cookie)` gesetzt. Ein Node braucht außerdem einen Namen, der ihn identifiziert, gesetzt über den Kommandozeilenparameter `-sname` (Short name, nur erster Teil des Hostnamen wird verwendet) oder `-name` (Long name, kompletter Hostname wird verwendet). Ein Node hat dann z.B. den Namen `nodename@localhost` oder `nodename@erlang.mydomain.de`.

Konsistent mit der Funktion `spawn` des vorangegangenen Kapitels wird über die Funktion `spawn(NodeName, Fun)` auf dem Node `NodeName` ein neuer Prozess erzeugt, der `Fun` evaluiert. Auch hier ist der Rückgabewert eine Pid. Sobald der Name eines anderen Nodes benutzt wird (wie hier bei der Funktion `spawn`, oder auch über die Funktion `net_adm:ping(NodeName)`), wird ein Verbindungsversuch zu diesem Node gestartet. Sollte dieser erfolgreich sein, sind die beiden Nodes fortan verbunden. Dieser Vorgang ist transitiv, sollte sich also Node1 mit Node2 verbinden, und Node2 ist bereits mit Node3 verbunden, wird sich nun auch Node1 mit Node3 verbinden.

Wenn man nun nur mit den Pids arbeitet, ändert sich beim Message Passing und dem im nächsten Abschnitt vorgestellten Verlinken von Prozessen nichts gegenüber der Variante innerhalb eines Erlang Systems, die Programmierung ist hier also transparent. Der Name von registrierten Prozessen ist jedoch lokal, sodass beim Nachrichtenversand in einem verteilten System zusätzlich der Nodename angegeben werden muss: `{registrierter_name@nodename} ! Message`.

3.2.3 Fehlerbehandlung

Eine Möglichkeit der Fehlerbehandlung sind Exceptions, die sehr ähnlich wie in Java und Ruby aussehen und funktionieren, weswegen hier nur einmal der Aufbau eines `try..catch` Blocks dargestellt wird, wodurch die Ähnlichkeit zu Java und Ruby bereits deutlich werden sollte, auch wenn der Name der Schlüsselworte teilweise abweicht:

```
try FunktionOderAusdruck of
  Pattern1 [when Guard1] -> Sequenz_von_Ausdrücken_1;
  Pattern2 [when Guard2] -> Sequenz_von_Ausdrücken_2;
  ...
  PatternN [when GuardN] -> Sequenz_von_Ausdrücken_N
catch
  Exception1 [when Guard1] -> Sequenz_von_Ausdrücken_1;
  Exception2 [when Guard2] -> Sequenz_von_Ausdrücken_2;
  ...
  ExceptionN [when GuardN] -> Sequenz_von_Ausdrücken_N;
after
  Sequenz_von_Ausdrücken
end
```

Für die Fehlerbehandlung in nebenläufigen/verteilten Programmen bietet Erlang die Konzepte gelinkte Prozesse und Monitore.

Über die Funktion `link(Pid)` werden zwei Prozesse miteinander verbunden. Es reicht, wenn ein Prozess diese Funktion aufruft, im Folgenden ist diese Verbindung beidseitig.

Wenn nun einer der Prozesse stirbt, erhält der andere Prozess ein sogenanntes „exit signal“. Ohne Vorkehrungen würde nun der Prozess, der das exit signal erhält, ebenfalls sterben. Man hat allerdings die Möglichkeit, über den Funktionsaufruf `process_flag(trap_exit, true)` einen Prozess zu einem Systemprozess zu machen und diesem die Möglichkeit zu geben, das entsprechende Signal abzufangen ohne zu sterben. Der Prozess würde dann, falls ein gelinkter Prozess stirbt, eine Nachricht der Form `{'EXIT', Pid, Grund}` empfangen.

Ein Monitor ist eine unidirektionale Überwachung. Ein Prozess kann über die Funktion `monitor(process, Pid)` einen Monitor auf einen Prozess mit dem Identifier Pid einrichten und empfängt die Nachricht `{'DOWN', Ref, process, Pid2, Grund}`, sobald der Prozess Pid stirbt. Der Wert `Ref` wird von der Funktion `monitor` zurückgegeben und ist eine eindeutige Referenz für diesen Monitor.

3.3 Bewertung

Auf den ersten Blick ist die Syntax von Erlang in manchen Fällen sicher gewöhnungsbedürftig (siehe z.B. auch einen Blogeintrag von einem der Autoren von CouchDB[23]), wenn man bisher hauptsächlich mit Sprachen wie Java und Ruby gearbeitet hat. Durch den geringen Sprachumfang und bei entsprechenden Hinweisen zu Stolperfallen (wie z.B. die verschiedenen Trennzeichen) sollte eine schnelle Einarbeitung in die Sprache aber aufgrund der vorhandenen Erfahrung mit wichtigen Konzepten wie Listen und Rekursion aus PROLOG möglich sein und gerade die Arbeit mit Prozessen und die Interprozesskommunikation ist durch die wenigen, direkt zur Sprache gehörenden Befehle schnell erlernt. Eine nähere Betrachtung zum zeitlichen Aufwand erfolgt im Rahmen der Bewertung zur Lösung der ersten Aufgabe (siehe Abschnitt 5.2.7).

[23] http://damienkatz.net/2008/03/what_sucks_abou.html

4 Konzeption

Dieses Kapitel beschreibt die Herangehensweise an die Untersuchung der Eignung der Programmiersprache Erlang für den Einsatz im Praktikum zur Vorlesung „Verteilte Systeme“. Es werden die Aufgabenstellungen, die bearbeitet werden, sowie mögliche Ergänzungen vorgestellt. Die Ergänzungen orientieren sich an den Lernzielen, die im Kapitel 2.2 dargestellt wurden. Basierend auf der ursprünglichen Aufgabenstellung werden die Anpassungen beschrieben, die nötig sind, um eine Bearbeitung mit Erlang möglich zu machen.

4.1 Aufgaben

Es werden die ersten beiden Aufgaben des bisherigen Praktikums als Grundlage für eine Einarbeitung in Erlang genutzt. Gleichzeitig findet damit eine Untersuchung des Aufwands statt, den eine Lösung in Erlang erfordert. Es folgt eine kurze Vorstellung der Grundideen der Aufgaben.

4.1.1 Aufgabe 1

In der ersten Aufgabe soll die einfache Client/Server-Anwendung „Message of the Day“ implementiert werden. Die Clients senden Nachrichten an einen Server, der diese verwaltet. Jede Nachricht bekommt eine eindeutige Nummer. Clients können vom Server von ihnen noch nicht gelesene Nachrichten abrufen, der Server muss also für jeden angemeldeten Client verwalten, welche Nachrichten dieser schon gelesen hat. Meldet sich ein Client eine gewisse Zeit nicht, wird er vom Server vergessen.

Diese Aufgabe eignet sich auf Grund der einfachen Struktur des Clients gut, um Implementierungen in verschiedenen Programmiersprachen zu testen. Zudem bietet sich die Aufgabe an, um unterschiedliche Implementierungsansätze in Erlang gegenüberzustellen.

4.1.2 Aufgabe 2

Die zweite Aufgabe beschäftigt sich mit der Implementierung eines verteilten Algorithmus sowie der Koordination und Verwaltung, die für den Ablauf des Algorithmus notwendig sind. Das Gesamtsystem dient der verteilten Berechnung eines größten gemeinsamen Teilers (ggT). Es besteht aus den Arbeitsprozessen, die die Berechnung durchführen sowie einem Koordinator, der den Ablauf steuert und die Ergebnisse visualisiert.

Die genauen Aufgabenstellungen in ihrer ursprünglichen Form zu beiden Aufgaben befinden sich im Anhang.

In einem ersten Arbeitsschritt werden die Aufgaben mit möglichst wenig Sprachmitteln von Erlang umgesetzt, um aufbauend auf den grundlegenden Erfahrungen, was Zeitaufwand und Komplexität angeht, Ergänzungen testen und bewerten zu können.

4.2 Anpassungen Aufgabe 1

Die erste Version der Implementierung der ersten Aufgabe wird auf reinem Message Passing basieren. Die in der ursprünglichen Aufgabe vorkommenden Funktionsaufrufe werden in Nachrichten umgewandelt und die bisherige synchrone Arbeitsweise durch die asynchrone „fire and forget“ Semantik des in Erlang implementierten Message Passing ersetzt. Ziel ist es, so ein Gefühl für die Grundlagen des nebenläufigen/verteilten Programmierens mit den Sprachmitteln von Erlang zu erhalten. Zunächst werden sowohl Client als auch Server in Erlang implementiert. Es wird beim Server auf eine grafische Oberfläche verzichtet. Es gibt zwar mit wxErlang in der Standarddistribution eine Anbindung an das GUI-Framework wxWidgets, allerdings würde dies von den Studierenden einen erheblichen Einarbeitungsaufwand erfordern, der mit dem eigentlichen Thema „Verteilte Systeme“ nichts mehr zu tun hat. Die GUI wird deswegen in Java entwickelt, näheres dazu im folgenden Abschnitt 4.2.1.

4.2.1 Integration anderer Programmiersprachen

Die GUI des Servers wird in Java umgesetzt, da die Grundlagen der GUI-Entwicklung mit Java den Studierenden bereits bekannt sind. So wird eine zusätzlich nötige Einarbeitung in das Thema GUI mit Erlang vermieden, um sich auf den Kernbereich „Verteilte Systeme“ zu konzentrieren. Die GUI wird auf einem anderen Rechner als der Server laufen können und es wird auch möglich sein, mehrere GUIs beim Server anzumelden. Für die Kommunikation wird die in der Standarddistribution von Erlang enthaltene Java-Bibliothek `JInterface` eingesetzt, die einen Distributed Erlang Node in Java simuliert. Auch hier wird die Kommunikation zunächst auf reinem Message Passing basieren. Als Einarbeitung in die Bibliothek

findet im ersten Schritt eine Implementierung des Clients mit Java statt. Aufbauend auf den gesammelten Erfahrungen wird dann die komplexere Aufgabe der GUI-Implementierung umgesetzt. Hier bietet sich dann auch ein Vergleich mit der alten Aufgabenstellung an, bei der die GUI ebenfalls in Java implementiert wurde.

Für die Integration mit C gibt es in der Standarddistribution die `erl_interface` Bibliothek. Um die Arbeit mit dieser Bibliothek zu beurteilen, wird eine Client-Implementierung mit C stattfinden. Eine höhere Abstraktionsebene bei der Arbeit mit Erlang-Nodes versucht die noch sehr junge Bibliothek `tinch++` zu bieten, die, wie der Name bereits vermuten lässt, für C++ geschrieben wurde. Diese Bibliothek ist allerdings zum Zeitpunkt der Erstellung dieser Arbeit erst in der Version 0.1 erschienen und nicht Teil der Standarddistribution. Aufgrund der höheren Abstraktion und damit einer eventuell leichteren Programmierung wird dennoch anhand einer Client-Implementation ein Test dieser Bibliothek mit einbezogen. Ähnlich wie schon bei Java bietet sich auch hier ein direkter Vergleich mit der ursprünglichen Aufgabe an, da in der bisherigen Aufgabenstellung C++ für die Client-Implementation eingesetzt wurde.

Sowohl bei der C als auch der C++ Implementation wird ein wichtiger Aspekt bei der Bewertung des Aufwandes eine Analyse zur Schwierigkeit der Einrichtung einer Kompilationsumgebung sein, da die Schwierigkeit hierbei einer der Kritikpunkte an der bisherigen Aufgabenstellung ist.

Für Ruby existiert mit `erlectricity` ebenfalls eine Bibliothek, um mit Erlang zu kommunizieren. Diese basiert allerdings auf der Kommunikation über Erlang-Ports und nicht über Distributed Erlang, sodass sie sich nicht für den Einsatz innerhalb des Szenarios der hier vorgestellten Implementation der ersten Aufgabe eignet.

4.2.2 Mögliche Ergänzungen der Aufgabenstellung

Neben der transparenten Integration anderer Programmiersprachen als Client, wie im letzten Abschnitt erwähnt, sollen an dieser Stelle noch weitere Ergänzungen vorgestellt werden, deren Aufwand und Komplexität anhand einer Beispiel-Implementierung getestet werden.

Nachdem die erste Implementierung auf Message Passing basiert, soll in einem zweiten Schritt eine Gegenüberstellung mit einer Implementierung basierend auf RPC erfolgen. Dabei werden folgende Fragen untersucht:

- Wie stark ändert sich dadurch die Implementierung auf Client- und Serverseite?
- War es einfach, diese Änderungen durchzuführen?
- Ist es zeitlich möglich, die Studierenden beide Implementationen durchführen zu lassen, um einen Vergleich Message Passing $\Leftrightarrow$ RPC zu ermöglichen

- Wie schwer ist die Anpassung der Implementationen in anderen Programmiersprachen?

Eine weitere mögliche Ergänzung betrifft die Fehlerbehandlung. Wie lassen sich die Stärken von Erlang in diesem Bereich verdeutlichen? Hier wird das Verlinken von Prozessen vorgestellt, um die technischen Grundlagen des in der OTP-Bibliothek implementierten Supervisor-Verhaltens (Behavior) kennen- und verstehen zu lernen. Dies soll eine Basis für einen eventuellen Einsatz des Supervisor-Verhaltens in den Ergänzungen zur zweiten Aufgabe bieten.

Zudem soll eine Implementierung Gebrauch von einem der am häufigsten eingesetzten Behaviors machen, dem `gen_server`. Auch hier steht eine Bewertung des Aufwands beim Einsatz von `gen_server` im Vordergrund.

In diesem Zusammenhang stellen sich dann bspw. folgende Fragen:

- Welche zusätzlichen Konzepte und Techniken müssen gelernt werden?
- Welche Vorteile ergeben sich aus dem Einsatz von `gen_server`?
- Inwieweit beeinflusst dies die Implementierung in anderen Programmiersprachen?

4.3 Anpassungen Aufgabe 2

Um eine möglichst einfache Implementierung als Grundlage für Anpassungen zu haben, wird auch Aufgabe 2 zunächst nur in Erlang und nur über Message Passing umgesetzt. Aufbauend darauf erfolgen Erweiterungen, die sich auf fehlertolerante, verteilte Systeme konzentrieren. Zudem sollen Erfahrungen, die aus den Implementierungen der ersten Aufgabe gewonnen wurden, in die Lösung dieser Aufgabe einfließen und beim Vorstellen der Lösung im folgenden Kapitel erläutert werden.

4.3.1 Integration anderer Programmiersprachen

Wie schon in Aufgabe 1 wird auch hier die GUI für die Visualisierung in Java implementiert. Je nachdem, wie die Implementierung mit anderen Programmiersprachen in Aufgabe 1 verläuft, werden Worker oder Starter ggf. in anderen Sprachen implementiert.

4.3.2 Mögliche Ergänzungen der Aufgabenstellung

Die zu testenden Ergänzungen bei dieser Aufgabe konzentrieren sich auf fortgeschrittene Techniken beim Einsatz von Erlang, insbesondere den Einsatz der Techniken, die die Grundlage für Supervisor Bäume sind bzw. für fehlertolerante Systeme.

In der Aufgabenstellung wird bereits angedeutet, dass die beteiligten Systemteile je nach Zustand unterschiedliche Aufgaben erfüllen. Hier bietet sich also eine Umsetzung über eine Finite State Machine (FSM) an, die durch die OTP-Bibliothek über das `gen_fsm` Behavior unterstützt wird.

Nachdem in den Ergänzungen zu Aufgabe 1 die Grundlagen der Fehlerbehandlung in verteilten Erlang-Systemen eingeführt wurden, soll diese Aufgabe genutzt werden, um die Möglichkeit des Einsatzes eines Supervisor Tree zu testen. Der Koordinator oder die Starter werden gleichzeitig als Supervisor fungieren, die eventuell abgestürzte Arbeitsprozesse neu starten. Es wird außerdem ein Blick auf die Möglichkeiten des Fehler-Logging geworfen und untersucht, ob die Aufgabe sich zusätzlich eignet, eine Ereignisbehandlung mit `gen_event` durchzuführen. Zusätzlich wird die Lösung als OTP-Application verpackt und die in der Erlang Standarddistribution enthaltenen Monitoring-Tools eingesetzt.

5 Realisierung

In diesem Kapitel werden die Implementierungen der im letzten Kapitel vorgestellten Aufgaben dargestellt. Es wird auf den zeitlichen Umfang eingegangen, den die Implementierungen in den unterschiedlichen Umgebungen benötigt haben. Im Verlauf der Vorstellung der Implementierungen in anderen Programmiersprachen als Erlang wird auf die verwendeten Bibliotheken eingegangen, insbesondere auf die Schwierigkeit der Einrichtung der Entwicklungsumgebung und dem zusätzlich benötigten Aufwand bei der Anwendung. Doch zunächst eine kurze Darstellung der Entwicklungsumgebung, die für die Programmierung mit Erlang eingesetzt wurde.

5.1 Entwicklungsumgebung

Entwickelt wurde hauptsächlich unter Microsoft Windows 7, wobei im Labor der HAW die Tests auf einem SuSE-Linux durchgeführt wurden. Getestet wurde zudem zuhause in einem Netzwerk aus einem Laptop mit Windows XP, dem Entwicklungsrechner unter Windows 7 sowie einer virtuellen Maschine mit Ubuntu Linux 10.04. Der Erlang-Code konnte hier eins zu eins übernommen werden, es waren keine system-spezifischen Anpassungen notwendig. Ein Punkt, auf den es allerdings zu achten gilt, wenn man zuhause entwickelt und an der HAW testet, ist die unterschiedliche Netzwerk-Konfiguration, insbesondere was die Namensauflösung betrifft. Hier gilt es darauf zu achten, ob man einen Erlang-Node mit kurzem Namen oder vollständigem Namen startet. Erlang wurde in der Version 5.7.5/OTP R13B04 eingesetzt.

Als Entwicklungsumgebung diente eine Standard-Eclipse-Distribution mit dem Erlang-Plugin Erlide[24]. Die Installation des Plugins erfolgte unkompliziert über den in Eclipse vorhandenen Installationsmechanismus. Diese Umgebung ist ebenfalls auf den Laborrechnern an der HAW vorhanden. Für die C/C++ Entwicklung wurde die CDT-Umgebung für Eclipse eingesetzt sowie unter Windows MinGW installiert. Wer gerne mit Emacs arbeitet, für den gibt es in der Standarddistribution von Erlang einen Erlang-Mode[25]. Weitergehende Funktionalität bietet das Projekt Distel[26] [27].

[24] http://marketplace.eclipse.org/content/erlide-erlang-eclipse-ide
[25] http://www.erlang.org/doc/man/erlang.el.html
[26] http://code.google.com/p/distel/
[27] http://bc.tech.coop/blog/070528.html

5.2 Aufgabe 1

Die Implementierung der Lösung zu Aufgabe 1 diente vor allem der Gewinnung von Entwicklungserfahrung mit Erlang. Sie bot außerdem die Möglichkeit, mit verschiedenen Implementierungen in Erlang und anderen Programmiersprachen zu experimentieren. Die Ergebnisse werden in den folgenden Abschnitten vorgestellt. Es werden in den meisten Fällen nur kleine Ausschnitte des Quelltextes zur Erläuterung eingesetzt. Der vollständige Quelltext befindet sich auf der beiliegenden CD.

5.2.1 Implementierung mit purem Erlang und Message Passing

Als Basis für weitere Experimente diente eine Implementierung, die für die Kommunikation nur auf Message Passing setzt. In einem ersten Schritt wurde der Message-Server implementiert.

5.2.1.1 Nachrichtenserver

Die Grundstruktur des Servers besteht aus einer Funktion `loop`, in der in einem `receive` Block die unterschiedlichen Nachrichten verarbeitet werden und die Funktion über einen endrekursiven Aufruf wieder gestartet wird. Die in der ursprünglichen Aufgabe vorhandenen Methoden spiegeln sich hier in den Nachrichten wieder, die der Server versteht:

```
loop(...) ->
  ...
  receive
    {From,{getmsgid,RechnerID}}->
      ...
      loop(...);
    {From,{dropmessage,SenderID, Zeit, Nachricht,
                   MessageID}}->
      ...
      loop(...);
    {From,{getmessages,RechnerID}}->
      ...
      loop(...);

    ...
  end,
  ...
end.
```

Als Argumente an die Funktion `loop` werden eine Reihe von Variablen übergeben, die den Zustand des Servers speichern. Der Zustand des Servers besteht aus den folgenden Variablen:

- `Clients` ist ein Dictionary(eine Key-Value Struktur, vgl. bspw. HashMap in Java), in dem die registrierten Clients mit ihren Werten (letzte abgerufene Nachricht und zugehöriger Timer) gespeichert sind. Als Key dient die ID der Clients.
- `DeliveryQueue` Die für den Abruf durch die Clients vorgehaltenen Nachrichten, ebenfalls ein Dictionary. Als Key kommt die Nummer der Nachricht zum Einsatz.
- `HoldbackQueue` Bei Lücken im Nachrichtenstrom dient diese Struktur als Puffer für eingehende Nachrichten. Es wird eine einfache Liste eingesetzt.
- `Wartezeit` Die Zeit, die der Server wartet bevor er sich selbst beendet.
- `WarteTimer` Die PID des Timers zur Wartezeit wird mitgeführt, um diesen anhalten und neustarten zu können.
- `ClientTimeout` Zeit, bis ein Client vergessen wird.
- `MaxNachrichtenAnzahl` Die maximal in der DeliveryQueue vorgehaltene Anzahl an Nachrichten.
- `MsgID` Die momentane durch Clients abzufragende MsgID, wird jeweils inkrementiert.
- `Nameserver` Die PID des Nameservers.
- `HoldTimer` Die PID des Timers für die HoldbackQueue.
- `GuiList` Die Liste mit den PIDs des beim Server registrierten GUIs

Diese Struktur mit einer endrekursiven Server-Schleife, die als Argument den Serverzustand mitführt, ist ein Standardmuster bei der Implementierung von Serverprozessen in Erlang. Es ist keine Synchronisation erforderlich, da die Nachrichten der Clients nacheinander abgearbeitet werden. Dadurch entfallen viele mögliche Fehler, die z.B. durch Locking entstehen können.

Dank der Möglichkeit, aus der Erlang-Shell heraus Nachrichten an Prozesse schicken zu können, konnte die Serverfunktionalität bereits während der Implementierung und ohne vorhandenen Client immer wieder getestet werden. So konnten die drei im obigen Code-Abschnitt abgebildeten Nachrichtentypen nacheinander implementiert und getestet werden. Fehler wurden schnell gefunden und es stand somit zum Start der Client-Implementierung bereits ein gut funktionierender Server zur Verfügung, was die Fehlersuche auf der Client-Seite vereinfachte, da klar war, dass der Fehler hier zu suchen ist.

Soweit zur Grundstruktur, es folgen noch ein paar Beschreibungen von Implementationsdetails. Für die Konfiguration wurde eine Konfigurationsdatei genutzt, die Erlang-Tupel enthält, da sich diese sehr einfach mittels `file:consult("server.cfg")` in eine Liste einlesen und weiterverarbeiten lassen. Die Datei sieht z.B. so aus:

```
{nameserver, 'nameserver@tatooine'}.
{wartezeit, 400000}.
{clienttimeout, 12000}.
{maxnachrichtenanzahl, 125}.
```

Für den Zugriff auf einen bestimmten Konfigurationswert dient die rekursive Funktion `get_config_value(Key,Liste)`[28]:

```
get_config_value(_Key, []) ->
  {error, not_found};
get_config_value(Key, [{Key, Value} | _Config]) ->
  {ok, Value};
get_config_value(Key, [{_Other, _Value} | Config]) ->
  get_config_value(Key, Config).
```

Um die verschiedenen Timer, die für diese Aufgabe benötigt werden, zu realisieren, kommen die im Buch von Joe Armstrong im Rahmen der Vorstellung des Erlang-Message-Passing abgebildeten Funktionen `sleep(T)`, `start_timer(Time, Fun)`, `cancel_timer(Pid)` und `timer(Time, Fun)` zum Einsatz (vgl S. 142-145 Armstrong (2007b)). Sollten die Timer ablaufen, schicken sie entsprechende Nachrichten an den Server:

- `process_hold` Die Wartezeit für die HoldbackQueue ist abgelaufen. Es werden entsprechende Fehlernachrichten sowie die Nachrichten aus der HoldbackQueue in die DeliveryQueue eingefügt.
- `deleteclient` Die Wartezeit eines Client ist abgelaufen, er wird aus dem Dictionary der bekannten Clients entfernt.

Daneben kommen noch eine Reihe an Hilfsfunktionen zum Einsatz, die der Verständlichkeit des Quelltextes dienen und hier nicht näher erläutert werden. Die weiteren Nachrichtentypen, die in der Server-Schleife verarbeitet werden, werden in den folgenden Abschnitten an den Stellen erläutert, an denen sie zum Einsatz kommen, um den nötigen Kontext herzustellen.

[28] http://spawnlink.com/articles/managing-application-configuration/

5.2.1.2 Nameserver

Neben dem Nachrichtenserver kommt noch ein einfacher Nameserver zum Einsatz, über den die Clients die PID des Nachrichtenservers finden können. Er basiert auf dem simplen Nameserver `kvs` aus Armstrong (2007b) (S. 170), benutzt aber ein Dictionary zum Speichern der Werte und versteht neben `store` und `lookup` auch `unregister` Nachrichten, um einen Nachrichtenserver beim Runterfahren beim Nameserver abmelden zu können.

5.2.1.3 Client

Für die Konfiguration und die Timer kamen die selben Techniken wie beim Server zum Einsatz, weswegen hier nicht nocheinmal darauf eingegangen wird. Der Client besteht, wie bereits der Nachrichtenserver, im Wesentlichen aus einer endrekursiven Funktion. In dieser wird der Nachrichtenabstand über einen Aufruf der `sleep` Funktion gewährleistet, eine Nachrichtennummer (MsgID) beim Server erfragt und im anschließenden `receive` Block auf die Antwort bzw. das Signal zum Beenden gewartet. Beim Erhalt einer Nachrichtennummer wird eine entsprechende Nachricht an den Server gesendet. Im Anschluss wird ggf. der Nachrichtenabstand zufällig vergrößert/verkleinert oder die dem Client unbekannten Nachrichten beim Server abgefragt. Da die Hauptfunktion relativ kompakt ist, wird sie hier einmal in voller Länge gezeigt (mit kleinen Kürzungen auf Grund zu langer Zeilen):

```
loop(Nachricht, NachrichtenAbstand, RechnerID, ...) ->
 sleep(NachrichtenAbstand*1000),
 MessageServer ! {self(),{getmsgid,RechnerID}},
  receive
   {MessageServer,{getmsgid,RechnerID,MsgID}} ->
    Zeit = time(),
    MessageServer ! {self(),{dropmessage,RechnerID,
       Zeit,...}},
    io:format("~p:␣Nachricht␣Nr.␣~p␣gesendet.~n",
       [Zeit,MsgID]);
   kill ->
    exit(0)
  end,
  AnzahlTextzeilenNeu = AnzahlTextzeilen +1,
  case AnzahlTextzeilenNeu rem 5 of
   0 -> NachrichtenAbstandNeu =
          create_abstand(NachrichtenAbstand);
   _ -> NachrichtenAbstandNeu = NachrichtenAbstand
  end,
  case AnzahlTextzeilenNeu rem 7 of
```

```
    0 -> spawn(fun() -> hole_nachrichten(MessageServer,
                                         RechnerID) end);
    _ -> ok
  end,
 loop(Nachricht, NachrichtenAbstandNeu, RechnerID, ...).
```

Die beiden Hilfsfunktionen `create_abstand` und `hole_nachrichten` dienen der Übersichtlichkeit und generieren alle fünf gesendeten Nachrichten einen neuen Nachrichtenabstand bzw. holen alle sieben gesendeten Nachrichten die unbekannten Nachrichten beim Server ab. Zum Abholen der Nachrichten wird ein eigener Prozess erzeugt, sodass dies nebenläufig zur Hauptschleife geschieht.

5.2.2 Client-Implementierungen in Java und C/C++

Nach der erfolgreichen Implementierung in Erlang folgte die Implementierung der Client-Funktionalität in den anderen Programmiersprachen. Die Struktur wurde von der Lösung in Erlang übernommen, es musste also nur eine Anpassung an die gegebenen Sprachmittel stattfinden.

5.2.2.1 Java

Die Entwicklung der Java-Lösung erfolgte mithilfe der `jinterface` Bibliothek, die im Paket `com.ericsson.otp.erlang.*` diverse Hilfsklassen bereitstellt, um von Java aus mit Erlang Nodes zu kommunizieren. Es gibt Klassen für die Abbildung der Erlang Datentypen (`OtpErlangAtom, OtpErlangTuple, OtpErlangPid` etc.), für die Repräsentation eines Erlang Node (`OtpNode`) sowie eines Prozesses samt Mailbox für eingehende Nachrichten (`OtpMbox`).

Die größte Schwierigkeit bei der Umsetzung des Erlang Quellcodes nach Java besteht in der Konstruktion bzw. dem Entpacken der Erlang Tupel, die als Nachrichten verschickt werden. Als Beispiel an dieser Stelle die Hilfsfunktion zum Erstellen des Erlang Tupel `{self(),{getmsgid,RechnerID}}`:

```
OtpErlangTuple createGetMsgID(OtpErlangPid pid,
        String rechnerID) {
  OtpErlangObject[] msg = new OtpErlangObject[2];
  msg[0] = pid;
  OtpErlangObject[] temp = new OtpErlangObject[2];
  temp[0] = new OtpErlangAtom("getmsgid");
  temp[1] = new OtpErlangString(rechnerID);
  OtpErlangTuple temp_tuple = new OtpErlangTuple(temp);
```

```
  msg[1] = temp_tuple;
  OtpErlangTuple tuple = new OtpErlangTuple(msg);

  return tuple;
}
```

Wie man sieht, ist dies im Vergleich zum Erlang-Quellcode deutlich umfangreicher und es verursachte bei der Bearbeitung dann auch den meisten zeitlichen Aufwand.

5.2.2.2 C/C++

Für die Entwicklung mit C müssen die Pfade zu den Include- und Library-Verzeichnissen der `erl_interface` Bibliothek im Erlang-Installationsverzeichnis in den Projekteigenschaften eingetragen werden. Die Kompilierung hat allerdings nur unter Linux funktioniert, unter Windows mit MinGW als Compiler-Umgebung gab es nur Fehlermeldungen. Zudem ist beim Eintragen der Bibliotheken in den Linker-Eigenschaften auf die richtige Reihenfolge zu achten, sonst kommt es zu Fehlern. Man gibt als erstes die `pthread`-, dann die `erl_interface`- und als letztes die `ei`-Bibliothek an.

Die Implementierung in C ist deutlich aufwändiger als in Java, auch wenn das bauen von Erlang Tupeln dank unterstützender Funktionen einfacher ist. Das Verbinden mit Nodes ist schwieriger, man muss die manuelle Speicherverwaltung beachten und die Verarbeitung der Kommunikation und die Initialisierung ist komplexer. Bei der Implementierung mit C wurde direkt mit RPC gearbeitet um die Komplexität etwas niedriger zu halten. Von der Struktur her gleichen sich die Implementierungen allerdings, so dass an dieser Stelle auf das abdrucken des Quellcodes verzichtet wird.

Der Installationsaufwand für die Bibliothek `tinch++` ist deutlich umfangreicher als für die reine C-Implementierung. Es müssen zusätzlich CMake, die boost Bibliothek sowie für die Dokumentation Doxygen installiert bzw. im Fall der boost-Bibliothek kompiliert werden. Anschließend kann dann die `tinch++` Bibliothek kompiliert werden. Auf dem Entwicklungsrechner kam es dabei leider zu Fehlermeldungen beim Linken der Testprogramme und die Bibliothek konnte nicht eingesetzt werden. Getestet wurden die Versionen 0.1 und 0.2. Dies ist schade, da sich anhand der Code-Beispiele erkennen lässt, dass die Programmierung dichter an der in Erlang ist, insbesondere die Konstruktion der Nachrichten-Tupel erfolgt auf intuitivere Weise. Die Bibliothek unterliegt allerdings aktiver Entwicklung. Kurz vor Abgabe dieser Arbeit lag Version 0.3 vor, und es lohnt sich eventuell, die zukünftige Entwickung im Auge zu behalten, falls eine Kommunikation über Programmiersprachengrenzen hinaus über das Message Passing von Erlang erwünscht ist.

5.2.3 GUI-Implementierung mit Java

Der Code für die Benutzeroberfläche stammt aus der originalen Lösung des Autors zur ersten Aufgabe des Praktikums und wurde um den Code zur Kommunikation mit dem Erlang Nachrichtenserver erweitert. Die GUI-Funktionalität unterscheidet sich leicht von der in der ursprünglichen Aufgabe. Die GUI kann unabhängig vom Nachrichtenserver und auch auf anderen Rechnern gestartet werden. Zudem können sich mehrere GUIs beim Nachrichtenserver registrieren. Für diese Funktionalität wurde der Nachrichtenserver um die Bearbeitung der folgenden Nachrichten erweitert:

- `{From,register_gui}` Mit dieser Nachricht registriert sich eine GUI beim Server. Es wird ein Link zwischen GUI und Nachrichtenserver erzeugt, sodass der Nachrichtenserver benachrichtigt wird, wenn die GUI geschlossen wurde. Zudem wird die GUI in die Liste der registrierten GUIs eingefügt.
- `{'EXIT',From,Reason}` Diese Nachricht wird empfangen, wenn ein gelinkter Prozess stirbt. Da in dieser Aufgabe nur die GUIs mit dem Nachrichtenserver gelinked sind, kann hier der entsprechende Code zum entfernen der GUI aus der Liste der registrierten GUIs aufgerufen werden.

Zudem werden an den passenden Stellen die Ereignisse, die in der GUI angezeigt werden sollen, über Nachrichten an alle registrierten GUIs gesendet. Die Implementierung des Nachrichtenempfangs bzw. -sendens auf Java-Seite war dank Copy&Paste aus dem bereits erstellten Java-Client nicht mehr im gleichen Maße aufwändig wie noch bei der Client-Programmierung.

5.2.4 RPC-Implementierung

Den Nachrichtenserver auch über Remote Procedure Calls erreichbar zu machen ist sehr einfach möglich und erfordert nur eine Ergänzung der bisherigen Implementierung, kein komplettes Umschreiben. Man ergänzt den Nachrichtenserver um eine funktionale Schnittstelle für die zur Verfügung gestellte Funktionalität. Die Hauptschleife mit der Nachrichtenverarbeitung bleibt bestehen. Als Beispiel so einer Funktion der funktionalen Schnittstelle dient `getmsgid(RechnerID)` zur Erfragung einer Nachrichtennummer:

```
getmsgid(RechnerID) ->
  {MessageServer,{getmsgid,RechnerID,MsgID}} =
                      rpc({getmsgid,RechnerID}),
        MsgID.

rpc(Q) ->
  server!{self(),Q},
  receive
```

```
    Reply ->
      Reply
  end.
```

Die Funktion `rpc` übernimmt das Senden und Empfangen der Nachrichten an den Server. Die Nachrichten werden dann vom Server wie bisher verarbeitet. Die anderen Funktionen des Servers werden analog zu `getmsgid(RechnerID)` mit Hilfe der Funktion `rpc` implementiert.

Auf Client Seite greifen die Änderungen stärker in die Struktur des Programms ein, da keine Nachrichten mehr an den Server gesendet werden, sondern die Funktionen des Interfaces über RPC angesprochen werden. Über den Bibliotheksaufruf `rpc:call(node(MessageServer),server,getmsgid,[RechnerID])` wird ein RPC an den Server durchgeführt, in diesem Fall an die oben dargestellte Funktion `getmsgid(RechnerID)`. Das erste Argument der Funktion erwartet den Namen des Nodes, auf dem man den RPC aufrufen möchte. Der Funktionsaufruf `node(MessageServer)` gibt den Namen des Nodes der in der Variablen MessageServer gespeicherten PID zurück. Die beiden nächsten Argumente sind Modul- und Funktionsname der aufzurufenden Funktion, die Liste im letzten Argument sind die Optionen, die an die Funktion übergeben werden.

Mehr ist nicht nötig, um vom purem Message Passing zu einem ansprechenden, über RPC erreichbaren, funktionalen Server-Interface zu kommen. Da die Hauptschleife des Servers ohne Änderung bestehen bleibt, können zudem die Clients und die GUI, die nur über Message Passing mit dem Server kommunizieren, weiterhin eingesetzt werden.

5.2.5 Einsatz von gen_server

Noch einen Schritt weiter geht dann die Implementierung über das Behavior `gen_server`. Dies ist eine der am häufigsten eingesetzten Techniken bei der Programmierung mit Erlang. Die Server-Funktionalität wird über Implementierungen von einer Reihe an vorgegebenen Funktionen erzeugt. Hier ist ein größerer Umbau des bisherigen Servers erforderlich, da die gesamte Struktur angepasst werden muss. Der Client, der über RPC kommuniziert hat, kann dagegen unverändert übernommen werden, da das Interface des Servers gleich bleibt.

In der Implementierung mit gen_server kamen bereits einige der gemachten Erfahrungen (siehe 5.2.6) aus den bisherigen Implementierungen zum Einsatz, wie beispielsweise ein `record`, um den Zustand des Servers zu speichern und der Einsatz des `global` Moduls für die Namensauflösung. Ansonsten bestand die Implementierung hauptsächlich im Kopieren der entsprechenden Code-Teile der ursprünglichen Version an die vom gen_server Behavior vorgesehenen Stellen. Diese Implementierung bricht mit der Abwärtskompatibilität zu den

Implementierungen mit reinem Message Passing. Theoretisch ließe sich diese herstellen, indem man entsprechende `handle_info` Funktionen implementiert.

Die Java GUI wurde erweitert, um über eine RPC Aufruf an die Funktion `global:whereis_name()` die Pid des Servers rauszufinden.

5.2.6 Erfahrungen aus der ersten Aufgabe

An dieser Stelle soll noch eine Aufzählung an gemachten Erfahrungen folgen. Diese bestehen zum Teil aus „Stolpersteinen“, auf die hingewiesen werden soll und zum anderen aus möglichen Vereinfachungen, die erst aus dem Zuwachs an Erlang-Wissen im Laufe der Erstellung dieser Arbeit offensichtlich wurden:

- **Sichtbarkeit der Nodes untereinander**: Nodes in einem verteilten Erlang-System müssen erst eine Verbindung zueinander herstellen, bevor eine Kommunikation stattfinden kann. Hierzu eignet sich z.B. der Befehl `net_adm:ping(Nodename)`, wobei `Nodename` hier der Nodename des entfernten Nodes ist. Dieser Befehl ist auch in der Erlang-Shell nützlich, um etwaige Netzewerk- oder Namensauflösungsprobleme zu erkennen.
- **Server-Zustand**: In der Implementierung der ersten Aufgabe wird der Zustand des Servers in einer Reihe an Variablen gespeichert. Dies hat lange Funktionsaufrufe zur Folge, die den Quellcode schwerer zu lesen machen und fehleranfällig sind. Hier bietet sich eine Kapselung des Zustandes in einem Record an. Dies wird in der Lösung zur Aufgabe 2 Anwendung finden und dort noch einmal näher erläutert.
- **Nameserver**: Der Einsatz eines eigenen Nameserver ist überflüssig, da über den Funktionsaufruf `global:register_name(name, Pid)` die Pid unter dem angegebenen Namen global bei allen verbundenen Nodes bekannt gemacht werden kann.
- **Timer und Sleep**: Hier gibt es ebenfalls Bibliotheksfunktionen, die man nutzen kann.

Allerdings haben die eigentlich überflüssigen Implementierungen beim Einstieg in Erlang sehr geholfen, sodass eine Implementierung der entsprechenden Funktionalität in einem dem Praktikum vorausgehenden Vorbereitungstermin durchaus Sinn machen könnte. Zusammen mit den Funktionen zur Verarbeitung der Konfigurationsdateien kommen sehr viele zum Lösen der Aufgabe benötigte Techniken zum Einsatz.

5.2.7 Aufwandsschätzungen und Bewertung

Der Aufwand zum Aufsetzen der Entwicklungsumgebung ist deutlich geringer als bei der bisher eingesetzen CORBA-Umgebung, wenn man sich auf Erlang und Java im Zusammenspiel beschränkt. Für Windows gibt es ausführbare Installationsdateien, unter Linux kann man die zu seiner Distribution passende Paketverwaltung nutzen um Erlang zu installieren. In der Standardinstallation findet man dann auch die zur Arbeit mit der `jinterface` Bibliothek notwendige .jar-Datei. Anschließend kann man Eclipse installieren und über den Plugin-Mechanismus das Erlang Plugin Erlide hinzufügen. Jetzt muss man einmal in den Einstellungen den Pfad zur Erlang-Installation eintragen und dann kann man bereits mit dem ersten Projekt beginnen. Je nach Geschwindigkeit der Internet-Verbindung hat man in 10-30 Minuten eine funktionierende Entwicklungsumgebung.

5.2.7.1 Grundlegende Implementierung

Die erste Aufgabe wurde mit so wenig Einarbeitung in Erlang wie möglich umgesetzt, angefangen mit dem Nachrichtenserver und dem Nameserver, um realistische Bedingungen zur Bewertung des Zeitaufwandes zu erhalten. Innerhalb von vier Stunden wurden die Kapitel zur sequentiellen, nebenläufigen und verteilten Programmierung aus dem Buch von Joe Armstrong (Armstrong (2007b)) durchgearbeitet bzw. teilweise nur für einen groben Überblick überflogen. Die Kapitel zur Fehlerbehandlung und die fortgeschrittenen Themen weiter hinten im Buch wurden zunächst nicht beachtet.

Die Programmierung der grundlegenden Funktionalität des Servers gelang innerhalb von eineinhalb Stunden. Die Arbeit mit Prozessen und Nachrichten war hierbei der leichteste Part. Am meisten Schwierigkeiten machte die Umstellung auf die Eigenarten der funktionalen Programmierung mit Single Assignment Variablen, wenn man z.B. Java als Programmiersprache gewohnt ist, hier insbesondere der Umgang mit den komplexen Datentypen `dict` und `list` mitsamt ihrer Bibliotheksfunktionen. Andererseits helfen die Single Assignment Variablen bei der Fehlersuche, da man sehr schnell den Punkt im Quellcode findet, wo der Fehler sein muss. Ebenfalls hilfreich ist die Möglichkeit, von der interaktiven Erlang-Konsole Nachrichten an Prozesse schicken zu können. So konnten Teile des Servers bereits getestet werden, ohne dass ein Client existierte.

Die Implementierung des Client war bereits wesentlich einfacher, da man nach der anfänglichen Umgewöhnungsphase schnell ein Gefühl für die Programmierung in Erlang bekommt und man bereits mit wenigen Sprachmitteln komplexe Aufgaben lösen kann. Hilfreich ist es, die Online Dokumentation der Standardbibliothek nachschlagebereit zu haben, so lernt man schnell weitere nützliche Funktionen kennen. Nach einer Stunde war die erste Version fertig.

5.2.7.2 Fehlersuche und GUI

Anschließend begann die Fehlersuche beim Zusammenspiel von Client und Server, wobei zunächst beide auf einem Node gestartet wurden, um Probleme mit dem Netzwerk ausschließen zu können. Die Fehlermeldungen von Erlang zusammen mit dem oben bereits erwähnten Vorteil der Single Assignment Variablen gestalteten die Fehlersuche angenehm und effizient. Beim Starten auf unterschiedlichen Nodes im Netzwerk traten keine neuen Fehler auf, man muss lediglich darauf achten, wie das Netzwerk konfiguriert ist, also ob man kurze oder lange Nodenamen verwendet. Außerdem sollte man darauf achten, dass auf allen kommunizierenden Nodes derselbe Cookie-Wert gesetzt ist. Ob die Kommunikation der Nodes untereinander funktioniert, kann man wiederum sehr gut über die interaktive Erlang-Konsole feststellen. An dieser Stelle ein Hinweis zur Konsole in Erlide. Diese ist nicht so komfortabel zu bedienen wie die Original Erlang Shell, wie man sie erhält, wenn man das Programm `erl` von einer Konsole aus startet. Zudem kam es zwischendurch zu Instabilitäten, weswegen es ggf. komfortabler ist, in Eclipse zu entwickeln, das Starten der Programme dann aber über eine externe Konsole zu erledigen.

Die Umsetzung der GUI in Java war der nächste Schritt. Die Kommunikation aus Java heraus mit Erlang-Nodes erfordert eine Menge Code zum Zusammensetzen der Nachrichten-Tupel, wobei hier auch viel Copy&Paste zum Einsatz kam, sodass die Programmierung zwar aufwendig aber nicht sehr schwierig war. Als Grundlage diente das Beispiel zu Java aus „Erlang Programming“ (S. 337ff Cesarini und Thompson (2009)).

5.2.7.3 Bilanz

Nach sechs Stunden war war die Programmierung der Grundfunktionalität des Servers, der GUI und des Clients abgeschlossen. Hierauf aufbauend begann dann die Einarbeitung in Themen wie z.B. Fehlerbehandlung, RPC und der Einsatz der OTP-Bibliothek. Wenn man sich auf Erlang und Java beschränkt, liegt der zeitliche Aufwand mit vier Stunden Einarbeitung und sechs Stunden vorbereitende Implementierung gut im vorgegebenen Rahmen (siehe 2.1). Wenn man als Vorbereitung auf das Praktikum einen vorgelagerten Termin für die Einarbeitung in Erlang anbieten möchte, kann man aus den Erfahrungen mit der Implementierung der ersten Aufgabe ablesen, dass der Schwerpunkt weniger auf dem Einsatz von Prozessen und Message Passing liegen sollte, sondern auf dem Näherbringen der funktionalen Aspekte der sequentiellen Programmierung in Erlang.

5.2.7.4 Ergänzungen

Die Einbindung von anderen Programmiersprachen beschränkt sich bei den Implementierungen auf die Kommunikation über Distributed Erlang, also dem Simulieren eines Erlang

Nodes in der anderen Sprache, für eine Besprechung weiterer Möglichkeiten der Kommunikation siehe 6.2. Für die Umsetzung der GUI ist Java gut geeignet, da die Studierenden bereits Erfahrung mit der GUI-Programmierung in Java haben, eine Client-Implementierung in einer anderen Sprache als Erlang bietet in diesem Szenario allerdings kaum Mehrwert: Das bauen der Nachrichtentupel entspricht dem in der GUI und es kommen keine weiteren neuen Konzepte hinzu. In C ist es einfach nur noch aufwändiger. Wenn man tiefer in Erlang einsteigen möchte, kann so eine Low-Level-Programmierung interessant sein, im Rahmen der Vorlesung „Verteilte Systeme“ gibt es allerdings keine nennenswerten Argumente dafür. Zudem kommen in C und C++ die Probleme bei der Einrichtung der Entwicklungsumgebung dazu.

Der Einsatz des Behavior `gen_server` ist zwar bei der produktiven Entwicklung mit Erlang wichtig, im Rahmen eines Praktikums versteckt der Einsatz allerdings interessante Teile der Implementierung wie z.B. die rekursive Hauptfunktion. Im Quellcode[29] des Behavior findet man diese Bestandteile wieder. Interessanter aus Sicht der Lehre sind die Gründe, die zur Entwicklung der unterschiedlichen Behaviors geführt haben. In diesem Zusammenhang ist das Kapitel „OTP Introduction“ in Joe Armstrongs Buch (Armstrong (2007b)) interessant, hier werden die Hintergründe von `gen_server` erläutert. Außerdem kann man im Quelltext[30] zum `rpc` Modul erkennen, dass auch dieses auf einem `gen_server` basiert. Um Fehler während eines RPC zu erkennen, wird der Aufruf der `gen_server` Funktionen in einem Prozess durchgeführt, auf den der aufrufende Prozess einen Monitor eingerichtet hat.

5.3 Aufgabe 2

Nachdem die erste Aufgabe vor allem zum Einarbeiten in Erlang und dem Experimentieren diente, soll die Implementierung der zweiten Aufgabe sich auf die gewonnenen Erfahrungen stützen. Die Erweiterungen der Aufgabe 2 konzentrieren sich auf die Fehlerbehandlung und den Einsatz der in Erlang enthaltenen Konstrukte, um ein fehlertolerantes System aufzusetzen.

5.3.1 Implementierung mit purem Erlang

Auch bei dieser Aufgabe dient eine Implementierung in purem Erlang als Ausgangspunkt für weitere Tests von Implementierungs-Techniken.

Als erstes wurde, aufbauend auf den Erfahrungen aus der ersten Aufgabe, für den Server-Zustand ein Record definiert, das die einzelnen Bestandteile kapselt:

[29] http://github.com/erlang/otp/blob/dev/lib/stdlib/src/gen_server.erl
[30] http://github.com/erlang/otp/blob/dev/lib/kernel/src/rpc.erl

```
-record(state, {anzahl_ggt_prozesse, verzoegerungszeit,
                timeout, ggt, zustand=wait,
                gui=not_registered, ggt_prozesse=[]}).
```

Hier ist `state` der Bezeichner des Record, die nachfolgenden Atoms sind die Namen der Bestandteile. Die Angabe eines Wertes nach einem Gleichheitszeichen gibt einen Standardwert an, der benutzt wird, falls eine Komponente beim Erzeugen eines Record dieses Typs keinen Initialwert erhält. Durch den Einsatz des Records werden die Funktionsdefinitionen übersichtlicher, allerdings ist die Syntax zum Gebrauch des Records im ersten Moment erklärungsbedürftig und wenig intuitiv. Zum Zugriff auf die Bestandteile eines Records kommt Pattern Matching zum Einsatz. Nachdem das Prinzip einmal klar ist, geht die Arbeit mit dem State-Record allerdings leicht von der Hand und macht den Quelltext deutlich übersichtlicher.

Da der Koordinator verschiedene Zustände durchläuft und jeweils andere Nachrichten empfangen werden können, wird als Struktur eine FSM gewählt, wobei die einzelnen Zustände durch jeweils eine eigene Funktion mit `receive` Block realisiert werden. Ein Zustandsübergang erfolgt dann einfach durch einen entsprechenden Funktionsaufruf. Damit ergibt sich folgende Grundstruktur des Koordinators:

```
wait(#state{gui=GUI} = State) ->
  receive
    {init,GUI} ->
      init(State#state{zustand=init}),
      wait(State#state{zustand=wait});
    {register_gui, From} ->
      link(From),
      io:format("~p:␣Gui␣wurde␣registriert:␣~p.~n",
                                    [time(),From]),
      send_values(State),
      wait(State#state{gui=From});
    ...
    Msg ->
      io:format("Nicht␣erwartete␣Nachricht")
  end,
  ok.

init(#state{gui=GUI} = State) ->
  receive
    {getsteeringval, From} ->
      ...;
                {hello, {From,Name}} ->
      ...;
```

```
    {ready, GUI} ->
      ready(State#state{zustand=ready});
    {'EXIT',GUI,Reason} ->
      wait(State#state{gui=not_registered})
  end,
  ok.

ready(#state{gui=GUI} = State) ->
  ....

shutdown() ->
  ...
```

Neben diesen Zuständen, in denen entsprechend der Aufgabe die verschiedenen Nachrichtentypen behandelt werden, kommen noch ein paar Hilfsfunktionen zum Einsatz:

- Eine Funktion, um die aktuellen Konfigurationswerte an die Gui zu senden, die z.B. bei der Registrierung einer GUI am Koordinator zum Einsatz kommt.
- Eine Funktion, die eine Zufallszahl aus einem durch die Parameter bestimmten Intervall zurückliefert.
- Für den Aufbau des Ringes aus Arbeitsprozessen, das Senden der Initialwerte und das Starten der Berechnung durch Zusendung von Berechnungswerten an drei zufällige Prozesse gibt es ebenfalls eigene Hilfsfunktionen.
- Für das Mischen von Listen kommt eine Funktion `shuffle(List)` zum Einsatz, die auf der Erlang Community-Seite Trapexit[31] vorgestellt wurde, da es in der Standardbibliothek hierfür leider keine Funktion gibt.

Der Starter ist in der ursprünglichen Version ohne Supervisor sehr einfach gehalten und Bedarf keiner weiteren Erklärung:

```
start(Nummer) ->
 Gruppe = "1",
 Team= "03",
 Chef = global:whereis_name(chef),
 Chef ! {getsteeringval, self()},
 receive
  {values,{Anzahl,Verz,Timeout}}->
  List = lists:seq(1, Anzahl),
  lists:foreach(fun(Elem)->
   spawn(fun() ->
```

[31] http://www.trapexit.org/RandomShuffle

```
    worker:start(Gruppe++Team,Nummer,Elem,Verz,Timeout)
     end)
      end,List)
end.
```

Für die Implementierung der Arbeitsprozesse kommt das bekannte Muster der `receive`-Schleife zum Einsatz, in der die verschiedenen Nachrichtentypen abgearbeitet werden. Im Zustands-record werden die Nachbarn im Ring, eine Timer-Referenz und die verschiedenen Konfigurationswerte gespeichert. Über die Timer-Referenz lässt sich der Timer bei Erhalt einer neuen Zahl abbrechen und ein neuer Timer kann gestartet und gespeichert werden, um die Timeout-Funktionalität zu realisieren. Ansonsten kommen keine neuen Techniken zum Einsatz, sodass auf eine weitere Erläuterung an dieser Stelle verzichtet wird.

5.3.2 Ergänzungen

Wie schon bei der ersten Aufgabe basieren die hier vorgestellten Ergänzungen auf einer funktionierenden Implementierung mit purem Erlang.

5.3.2.1 Einsatz von gen_fsm

Das OTP-Framework bietet auch für den Einsatz einer FSM mit `gen_fsm` einen entsprechenden Abstraktionsmechanismus. Der Aufbau ist hier sehr ähnlich zu dem des `gen_servers`. Es müssten wiederum die Codeteile an die durch das Behavior vorgegebenen Stellen kopiert werden. Da im Verlauf der ersten Aufgabe und dem Einsatz von `gen_server` bereits deutlich wurde, dass sich der Lerneffekt in Grenzen hält, wenn man aus der bestehenden Lösung die Codeteile umkopiert, wurde bei dieser Aufgabe darauf verzichtet.

5.3.2.2 Implementierung eines Supervision Tree

Ursprünglich war es angedacht, den Koordinator zusätzlich als Supervisor auftreten zu lassen. Nach weiteren Recherchen im Verlauf der Erstellung dieser Bachelorarbeit wurde allerdings deutlich, dass Supervisor Trees pro Erlang Node aufgebaut werden, also nicht über Nodegrenzen hinweg. Prozesse auf anderen Nodes können sich zwar über Links oder Monitore über Fehler informieren lassen, das automatische neustarten wird jedoch nur auf demselben Node ausgeführt, auf dem auch der Fehler geschah. Deswegen wurde die Planung angepasst und die Starter dienen nun als Supervisor, die eventuell abstürzende Arbeitsprozesse auf ihrem Node neu starten. Hierbei kommt wiederum nicht das vorgegebene Behavior der OTP Bibliothek zum Einsatz, sondern es wird anhand der Sprachmittel von Erlang eine

eigene Supervisor Lösung implementiert, um den Aufwand und Lerneffekt einschätzen zu können.

Den Starter zum Supervisor seiner Arbeitsprozesse zu machen ist, mit den von Erlang zur Verfügung gestellten grundlegenden Sprachmitteln zur Arbeit mit Prozessen, sehr einfach möglich. Zunächst muss dieser zum Systemprozess gemacht werden (`process_flag(trap_exit,true)`). Anschließend nutzt man statt `spawn()` die verwandte Funktion `spawn_link()`. Durch diese wird gleich beim Erzeugen des Prozesses ein Link zwischen den beiden Prozessen erstellt. Nun kann man die `'EXIT'` Nachrichten der Arbeitsprozesse empfangen und entsprechend reagieren, also z.B. den Koordinator über das Abstürzen eines Prozesses informieren und einen neuen Arbeitsprozess starten. Dies ist auch die prinzipielle Arbeitsweise des Supervisor-Behavior aus der OTP-Bibliothek. Wenn man sich den Quelltext[32] des Supervisor-Behavior ansieht, findet man die entsprechenden Schritte wieder. Zusätzlich ist natürlich noch deutlich mehr Funktionalität vorhanden, um z.B. den generischen Einsatz zu ermöglichen.

5.3.3 Integration anderer Programmiersprachen

Da die Lösungen zu dieser Aufgabe sich vor allem auf verschiedene Techniken der Erlang Umgebung konzentriert, erfolgt nur die Umsetzung der GUI in Java. Als Grundlage dient die GUI aus der ersten Aufgabe mit Anpassungen der Nachrichtenformate an die Gegebenheiten dieser Aufgabe sowie weiterer Eingabefelder für die steuernden Werte. Hier ergibt sich nichts Neues im Vergleich mit der ersten Aufgabe. Nur der Umfang ist aufgrund der weiterreichenden Funktionalität größer.

5.3.4 Aufwandsschätzungen und Bewertung

Durch die Entwicklungserfahrung, die im Verlauf der Programmierung der verschiedenen Versionen der ersten Aufgabe aufgebaut werden konnte, gelang die Umsetzung der zweiten Aufgabe trotz deutlich komplexerer Aufgabenstellung in vergleichbarer Zeit.

Die Umsetzung eines auf die Aufgabe zugeschnittenen Supervisor gelang dank der in Erlang enthaltenen Konstrukte zur Fehlererkennung sehr einfach. Interessant wäre eventuell eine Umsetzung eines generischen Supervisors ähnlich dem der OTP-Bibliothek in einer späteren Aufgabe.

[32] http://github.com/erlang/otp/blob/dev/lib/stdlib/src/supervisor.erl

6 Schluss

In dieser Arbeit wurde die Eignung von Erlang für den Einsatz in der Vorlesung „Verteilte Systeme“ sowie dem dazugehörigen Praktikum untersucht. Dieses Kapitel bietet eine Zusammenfassung der gemachten Erfahrungen, einen Überblick über die Implikationen für die in der Vorlesung vermittelten Konzepte bei einem eventuellen Einsatz von Erlang sowie die abschließende Bewertung. Zudem erfolgt ein Ausblick auf Techniken, Systeme und Arbeiten rund um Erlang, die für die Implementierung der ersten beiden Aufgaben nicht von Interesse waren, allerdings interessante Aspekte aus dem Bereich „Verteilte Systeme“ behandeln. Hieraus ergeben sich eventuell auch Anreize für komplexere Aufgabenstellungen zu den Praktikumsterminen drei und vier.

6.1 Konzepte

Bisher stand die praktische Anwendbarkeit von Erlang für die Implementierung der Praktikumsaufgaben in dieser Arbeit im Vordergrund. Es wurden hauptsächlich Betrachtungen zu Einarbeitungsaufwand, Implementierungsaufwand, Syntaxbesonderheiten und ähnlichen Themen durchgeführt. In diesem Abschnitt soll nun eine abstraktere Betrachtung der Eignung von Erlang für die Vermittlung von Konzepten aus der Vorlesung erfolgen. In diesem Abschnitt stehen Fragen wie z.B. die folgenden im Vordergrund:

- Welche Konzepte aus der Vorlesung werden direkt in Erlang abgebildet?
- Wie werden diese ggf. abgebildet?
- Welche Eigenschaften haben die Komponenten in Erlang?
- Welche zusätzlichen Konzepte treten durch Erlang in den Vordergrund?
- Lassen sich weitere Inhalte einfach in Erlang darstellen/erläutern?
- Welche momentanen Inhalte der Vorlesung finden sich nicht wieder?

6.1.1 Message Passing als Grundlage

Erlang als Middleware setzt als grundlegende Kommunikationsform auf asynchrones Message Passing zwischen Prozessen. Damit wird Message Passing als Möglichkeit der Interprozesskommunikation direkt in der Sprache unterstützt und kann durch die Operatoren `!` (send) und `receive` einfach dargestellt werden. Das „Marshalling“ bzw. „Unmarshalling“ der Nachrichten in das für den Transport verwendete externe Format wird durch Erlang automatisch durchgeführt. Ohne weitere Programmierung folgt das Message Passing in Erlang einer „Fire and Forget“-Semantik, das Senden einer Nachricht schlägt niemals fehl, selbst wenn der Prozess, an den die Nachricht gesendet wird, gar nicht existiert (vgl. S. 92 Cesarini und Thompson (2009)). Möchte man über Fehler informiert werden, kann man z.B. auf Links und Monitore zurückgreifen, um über Fehler im Empfängerprozess informiert zu werden. Nachrichten müssen auch nicht direkt empfangen werden, sondern werden zunächst im Nachrichteneingang des empfangenden Prozesses hinterlegt. Nur wenn dieser die receive-Operation aufruft, können Nachrichten aus dem Nachrichteneingang entfernt werden. Für die Fehlerbehandlung auf Empfängerseite kann neben Links und Monitoren noch ein Timeout eingesetzt werden, dessen Anwendung direkt im receive-Statement unterstützt wird.

Die Reihenfolge der Nachrichten wird nur pro sendendem Prozess berücksichtigt. Wenn Prozess P1 also die Nachrichten M1 und M2 in dieser Reihenfolge an den Prozess P2 schickt, werden diese auch in dieser Reihenfolge im Nachrichteneingang von P2 enthalten sein, sofern beide Nachrichten ankommen. Es könnte also auch vorkommen, dass nur M1 oder auch keine Nachricht ankommt. Diese Annahme hinsichtlich der Nachrichtenreihenfolge kann aber ohne weitere Vorkehrungen nur für die Kommunikation innerhalb eines Node garantiert werden. In verteilten Systemen, wenn die Nodes sich auf unterschiedlichen, über ein Netzwerk verbundenen, Rechnern befinden, kann es bei einem kurzfristigen Ausfall des Netzwerks passieren, dass im obigen Beispiel nur M2 ankommt (vgl. Svensson und Fredlund (2007a)). Auf diese Fehlermöglichkeit in verteilten Systemen wird normalerweise nicht eingegangen, wenn das Message Passing in Erlang, speziell Distributed Erlang, vorgestellt wird. Es findet sich bspw. in keinem der beiden einführenden Bücher (Armstrong (2007b) und Cesarini und Thompson (2009)). Die Autoren Hans Svensson und Lars-Ake Fredlund schlagen deswegen eine Erweiterung der normalerweise zu findenden Garantie hinsichtlich der Reihenfolge von Nachrichten („The basic guarantee for message passing, in the intra-node case, is that messages sent from one process to another are delivered in order, if they are delivered at all, without message corruption or duplication.“, Svensson und Fredlund (2007b)) in die folgende vor: „If two messages m1 and m2 are sent, in order, from a process Q to a process P, and Q is linked (or monitors) to P, and no exit message from P is received at Q, then it is guaranteed that the messages have been delivered, in order, at P.“, Svensson und Fredlund (2007b).

6.1.2 Design Philosophie

In Erlang basiert die Strukturierung der Programme häufig auf Prozessen als Strukturierungseinheit. Joe Armstrong nennt diese Design-Philosophie „Concurrency Oriented Programming“ (vgl. Armstrong (2007a)). Häufig wird die Programmierung mit Erlang als „Actor-style Concurrency“ bezeichnet, und tatsächlich kann Erlang als eine Implementierung des ACTOR-Modells angesehen werden. In diesem Modell ist die einzige Art der Kommunikation die von voneinander isolierten Aktoren über Nachrichten. Aktoren reagieren auf Nachrichten, können Nachrichten an andere Aktoren versenden und ihr eigenes Verhalten gemäß den empfangenen Nachrichten anpassen (vgl. Agha (1985)). Die isolierten Prozesse in Erlang, die nur über Message Passing kommunizieren, entsprechen diesem Modell. Dass das Actor-Modell in den letzten Jahren mehr Aufmerksamkeit auf sich zieht, lässt sich z.B. durch ein Zitat aus der Einleitung der Arbeit von Agha ablesen: „It is generally believed that the next generation of computers will involve massively-parallel architectures. This thesis studies one of the proposed paradigms for exploiting parallelism, namely the actor odel of concurrent computation.“(S. iii Agha (1985)). Da diese Parallelisierung erst 20 Jahre später durch die großen Chiphersteller umgesetzt wurde, bekommt nun auch das Actor-Modell wieder mehr Aufmerksamkeit.

Man kann diese Abstraktion auch als Form der Objekt Orientierung sehen, denn der Zustand der Prozesse ist verborgen, er ist nur über die definierte Schnittstelle der vom Prozess verstandenen Nachrichten manipulierbar. Alan Kay, der als einer der Väter der Objekt Orientierung gilt, sieht in „Messaging“ den wichtigeren Aspekt bei der Objekt Orientierung[33].

6.1.3 Entfernter Aufruf

Das Thema in der Vorlesung, welches die meisten Änderungen durch einen Wechsel der eingesetzten Middleware-Technik erfahren würde, ist das Kapitel Interprozesskommunikation. Hier liegt der Fokus momentan auf der Erläuterung eines verteilten Objektmodells mit der konkreten Ausprägung in CORBA. In Erlang liegt der Fokus, wie im vorangegangenen Abschnitt beschrieben, auf Prozessen, die über Message Passing kommunizieren.

In CORBA oder Java RMI wird versucht, einen entfernten Methodenaufruf wie einen lokalen Methodenaufruf aussehen zu lassen. Es wird also versteckt, dass eine Kommunikation potentiell über ein Netzwerk stattfindet. Lokale und entfernte Aufrufe haben allerdings unterschiedliche Aufruf-Charakteristiken, z.B. was mögliche Fehler, die benötigte Zeit für den Aufruf oder den Speicherzugriff angeht. Dies wird in der Vorlesung in den Abschnitten zu Referenz- und Kopiersemantik sowie Fehlersemantik angesprochen.

[33] http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

Aufgrund dieser Unterschiede existiert die Ansicht, dass man entfernte Aufrufe nicht vor dem Programmierer „verstecken“ sollte, damit dieser sich der möglichen Fehler bewusst ist (vgl. Waldo u. a. (1994) und Vinoski (2008) (Randbemerkung: Steve Vinoski war in den 1990ern aktiv an der Entwicklung von und mit CORBA beteiligt, bevor er sich entschlossen hat, in eine andere Richtung zu gehen. Er hat damit eine ähnliche Entwicklung genommen, wie sie in dieser Arbeit für die Vorlesung vorgeschlagen wird)). Joe Armstrong unterstützt diese Ansicht[34], sodass der RPC-Mechanismus in Erlang etwas anders funktioniert als z.B. in einem verteilten Objektsystem. Er schreibt in diesem Blogpost zum Thema RPC in Erlang: “With send, receive and links the Erlang programmer can easily "roll they own RPC"with custom error handling. There is no "standard RPC stub generator" in Erlang nor would it be wise for there to be such a generator.“. Darauf folgen einfache Beispiele, wie verschiedene Implementierungen eines RPC in Erlang aussehen könnten. Die Beispiele in Joe Armstrongs Blogpost sind sehr kompakt und eignen sich gut zur Darstellung von verschiedenen Möglichkeiten des RPC in Erlang.

Das Modul `rpc`[35] aus der Standardbibliothek bietet Implementierungen für eine Auswahl an Standardoperationen im Bereich RPC. Wenn man sich den Quelltext des Moduls[36] anschaut, sieht man, wie die Implementierung im Grunde auf den Mechanismen des Message Passing und der Prozessprogrammierung in Erlang aufsetzt und für die Fehlererkennung auf Monitore setzt.

Als Ergänzung zum Thema Webservices, welche bisher in der Vorlesung auf Basis der Technologien WSDL, SOAP und UDDI vorgestellt werden, bietet sich der Architekturstil REST REpresentational State Transfer (Fielding (2000)) an. Dieser findet insbesondere im Bereich von Web-Applikationen Anwendung. Für Erlang gibt es mit Webmachine[37] ein einfach anzuwendendes Framework, um eigene Applikationen mit einer REST-Schnittstelle über HTTP auszustatten.

6.1.4 Fehlerbehandlung

Da Erlang für die Programmierung von fehlertoleranten Systemen konzipiert wurde, lohnt sich ein Blick auf die Konzepte, die die Sprache (bzw. die VM) in diesem Bereich anbieten. Neben der Möglichkeit, ähnlich wie in anderen Sprachen mit Exceptions zu arbeiten, gilt der Ansatz, entfernte Prozesse über Links und Monitore beobachten zu können und auf deren Fehler entsprechend zu reagieren, als Stärke von Erlang. Links und Monitore bilden die Grundlage für Supervisor-Hierarchien. Diese Konzepte fanden sich bisher nicht in der Vorlesung, da sich auch die dahinterstehende Philosophie vom in Sprachen wie Java und C++ häufig eingesetzten, sogenannten defensiven Programmierstil, unterscheidet.

[34] http://armstrongonsoftware.blogspot.com/2008/05/road-we-didnt-go-down.html
[35] http://erldocs.com/R13B04/kernel/rpc.html
[36] http://github.com/erlang/otp/blob/dev/lib/kernel/src/rpc.erl
[37] http://webmachine.basho.com/

Beim defensiven Programmieren wird versucht, alle möglichen Fehlerfälle abzufangen und im ausführenden Prozess zu beheben. In Erlang wird der Ansatz „Crash early“ bzw. „Let it crash“ verfolgt. Wenn ein Fehler in einem Arbeitsprozess auftritt, soll sich dieser beenden. Dieses wird dann von einem anderen Prozess, dem sogenannten Supervisor, erkannt und der Prozess neu gestartet (siehe Kapitel 4.3 ab S. 104 in Armstrong (2003)). Zur Frage, wann man Fehler abfangen sollte und wann man den Prozess sich beenden lassen sollte, gibt Joe Armstrong folgende Hinweise:

„- exceptions occur when the run-time system does not know what to do.

- errors occur when the programmer doesn't know what to do.“ (S. 107 Armstrong (2003))

Bei einem Fehler sollte sich der Prozess beenden. Dieser Fall kann z.B. eintreten, wenn die Spezifikation, nach der der Programmierer arbeitet, nicht vorgibt, wie in dem auftretenden Fall vorgegangen werden soll. Oder noch anders ausgedrückt „Program for what you expect rather than what you fear“[38].

6.1.5 Weitere Themen

Zum Thema Namensdienst bietet sich statt der Vorstellung des CORBA-Namensdienstes das Modul `global`[39] an, welches die Möglichkeit bietet, Namen bei allen verbundenen Nodes bekannt zu machen. Das Modul bietet zudem die Möglichkeit einen Lock zu setzen um ggf. eine geteilte Ressource zu schützen.

Das Thema „Zeit, Synchronisation und globale Zustände“ wird in der Vorlesung unabhängig von einer Implementierungstechnik vorgestellt und es ergeben sich keine Änderungen. Als Beispiele der Anwendung könnten allerdings die im Ausblick vorgestellten Systeme Dynamo bzw. Riak dienen.

Auch das Thema „Übereinstimmung und Koordination“ wird unabhängig von einer konkreten Technik behandelt, wobei auch zu diesem Thema Dynamo oder Riak als praktisches Anwendungsbeispiel fungieren könnten.

Beim Thema „Verteilte Transaktionen“ wird als konkrete Ausprägung das System CORBA/OTS vorgestellt. Ein entsprechendes System ist in der Erlang Standarddistribution nicht vorhanden.

Auch im Thema „Replikation“ kommen keine Details einer bestimmten Implementierungssprache vor.

[38] http://www.erlang-factory.com/upload/presentations/241/ErlangFactorySFBay2010-YogishBaliga,MarkZweifel.pdf

[39] http://www.erlang.org/doc/man/global.html

6.2 Ausblick

In diesem Abschnitt werden Technologien vorgestellt, die im Laufe der Bearbeitung der Aufgaben und dem Befassen mit Erlang das Interesse des Autors geweckt haben, aber nicht in die Implementierungen der ersten beiden Aufgaben passten.

6.2.1 Weitere Kommunikationsmöglichkeiten

In den Implementierungen der ersten beiden Aufgaben kam als Kommunikationstechnik das native Message Passing von Erlang zum Einsatz, auch für die Kommunikation mit Prozessen in anderen Programmiersprachen. Darüber hinaus gibt es noch weitere Möglichkeiten, über die Grenzen von Erlang hinaus mit anderen Systemen zu kommunizieren. Dieser Abschnitt bietet einen Überblick und eine Bewertung der einzelnen Technologien im Hinblick auf das Praktikum „Verteilte System“.

Als eine Einsatzmöglichkeit für Erlang wird häufig der Einsatz als „Glue“-System genannt, also als verbindende Komponente. Meistens ist hiermit die Verbindung von zwei Komponenten gemeint, die jeweils ein eigenes proprietäres, binäres Protokoll benutzen. Für diesen Einsatzzweck bietet Erlang ein Konstrukt, das das Parsen dieser Protokolle sehr einfach macht. Die Bit Syntax erweitert den Einsatzbereich von Pattern Matching auf binäre Daten. Sie ermöglicht den Zugriff auf einzelne oder zusammenhängende Bits. Wenn M im folgenden Beispiel die Daten enthält, steht in T der Wert der ersten fünf Bit, in U der Wert der nächsten sieben Bit und in V der Wert der folgenden drei Bit:

```
<<T:5, U:7, V:3>> = M.
```

In vielen anderen Sprachen müsste man mit Bit-Shifting und Bit-Masken arbeiten.

Aus dem kommerziellen Bereich kommen drei Technologien, die alle als Open Source freigegeben wurden. Die folgenden Abschnitte stellen diese kurz vor.

Von Facebook stammt **Thrift**[40], dass für die Programmierung von programmiersprachenübergreifenden Services entwickelt wurde. Thrift bringt eine eigene Sprache zur Beschreibung der Services mit. Aus diesen Beschreibungen werden dann über Generatoren die Skelette in den unterschiedlichen Programmiersprachen erzeugt. Es gibt Anbindungen für C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, und OCaml. Das System funktioniert damit ähnlich wie CORBA und bringt damit ähnliche Nachteile mit sich in Bezug auf den Einsatz im Praktikum.

[40] http://incubator.apache.org/thrift/

Von Google stammt ein sprach- und plattformneutrales binäres Nachrichtenformat für die effiziente Serialisierung von strukturierten Daten, die **Protocol Buffers**[41]. Auch hier kommen Beschreibungen der Nachrichtentypen in einer eigenen Sprache zum Einsatz, die dann in die Zielsprache übersetzt werden. In der Erlang-Welt kommen sie z.B. in dem unten besprochenen Riak zum Einsatz.

Von Joe Armstrong wurde **UBF** (Universal Binary Format) als Alternative zu XML als Transport-Format zwischen Anwendungen in verschiedenen Sprachen entwickelt (vgl. Armstrong (2002)). Die Schnittstelle zu Hibari[42], einer verteilten Datenbank, die in Erlang entwickelt wurde, basiert u.a. auf UBF, wobei UBF hier auch noch mit den beiden oben genannten Protokollen, Thrift und Protocol Buffers, kombiniert werden kann[43]. Die Implementierungen zu den verschiedenen Kombinationen sind Online verfügbar[44].

Bei Github[45] wurden **BERT** und **BERT-RPC**[46] entwickelt. BERT steht dabei für Binary ERlang Term. Der Standard basiert auf dem Format, das von der Erlang Funktion `erlang:term_to_binary` erzeugt wird.

Erlang eignet sich auf Grund der leichtgewichtigen Prozesse auch gut für das Aufrechterhalten vieler langlebiger Verbindungen, was z.B. dem Einsatz im Zusammenhang mit den kommenden HTML5 Websockets entgegenkommt.

Für die Kommunikation mit Systemen in anderen Sprachen steht somit eine große Auswahl an Protokollen bereit, für die es Erlang-Bibliotheken gibt und die man je nach Anwendungsfall gezielt auswählen kann.

Von den oben genannten Protokollen ist BERT(-RPC), als einfaches auf dem externen Erlang-Format basierendes RPC-Protokoll, für einen eventuellen Einsatz im Praktikum oder der Vorlesung am interessantesten, da es auf eine einfache, gut verständliche Struktur setzt und durch den Einsatz des externen Erlang-Formats einfach in Erlang umsetzbar ist. Auch eine Beschränkung auf Teile des Protokolls, z.B. nur `call` und `cast` Operationen, wäre denkbar. Der Quellcode zu Ernie[47], einer Server-Implementierung des Protokolls in Erlang, ist als Open Source verfügbar, leicht verständlich und bei Konzentration auf Teile des BERT-Protokolls auch vom Umfang her ausreichend klein. Eine Implementierung des Protokolls in den letzten Aufgaben des Praktikums wäre demnach machbar und durch die vorhandenen Bibliotheken für Klienten-Anwendungen auch einfach testbar.

[41] http://code.google.com/intl/de-DE/apis/protocolbuffers/
[42] http://www.geminimobile.com/technologies/technologiesDB.html
[43] http://hibari.sourceforge.net/hibari-developer-guide.en.html#client-api-ubf
[44] http://github.com/norton/ubf
[45] http://github.com/blog/531-introducing-bert-and-bert-rpc
[46] http://bert-rpc.org/
[47] http://github.com/mojombo/ernie/tree/master/elib/

6.2.2 Andere Syntax

Da die Syntax von Erlang für Programmierer aus anderen Sprachen wie Java, Ruby oder Python teilweise sehr ungewohnt erscheint, gibt es Ansätze, andere Sprachen für die ErlangVM bereit zu stellen.

Reia[48] orientiert sich an der Syntax von Ruby und Python. Der Quellcode wird in das Erlang Abstract Format übersetzt, eine Struktur zum repräsentieren von geparstem Erlang Code. Reia unterstützt die Standardfunktionalität von Erlang, wie z.B.:

- Die Standard-Datentypen
- Pattern matching
- Prozesse
- Das linken von Prozessen

Darüber hinaus bietet Reia z.B. ein einfacheres Arbeiten mit Strings und eine einfache Möglichkeit, Funktionen, die in Erlang geschrieben wurden, aufzurufen. Umgekehrt können Reia Module auch aus Erlang heraus aufgerufen werden.

Eine weitere Sprache mit einem ähnlichen Ansatz wie Reia ist **Efene**[49]. Die Syntax lehnt sich mehr an C, Java und Javascript an. Der Quellcode kann in den entsprechenden Erlang Code übersetzt oder direkt in eine `.beam` Datei kompiliert werden.

Lisp Flavoured Erlang (LFE)[50] stammt von einem der Hauptentwickler von Erlang, Robert Virding. Es bringt die Syntax von Lisp auf die ErlangVM.

Diese drei Projekte bringen jeweils unterschiedliche Syntaxfamilien auf die ErlangVM und zeigen, dass man nicht auf Erlang beschränkt ist, wenn man die Vorzüge der ErlangVM nutzen möchte. Alle drei Projekte unterliegen aktiver Entwicklung.

6.2.3 Distributed Key/Value Store

Ein Bereich der populärer werdenden sogenannten NoSQL Datenbanken sind die sogenannten Dynamo-inspirierten Systeme. Diese basieren auf dem von Amazon veröffentlichten Dynamo-Paper. In diesem wird die Architektur eines verteilten Datenbanksystems mit hoher Schreib-Verfügbarkeit erläutert, welches bei Amazon z.B. für den Warenkorb im Einsatz ist (vgl. DeCandia u. a. (2007)). Dynamo bietet die beiden Eigenschaften Verfügbarkeit (Availability) und Partitionstoleranz (Partition Tolerance) des sogenannten CAP- oder Brewer-Theorem (Brewer (2000)). Dieses besagt, dass ein verteiltes System nur zwei

[48] http://wiki.reia-lang.org/wiki/Reia_Programming_Language
[49] http://efene.tumblr.com/
[50] http://github.com/rvirding/lfe

der drei Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz (CAP Consistent, Available, Partition-tolerant) gleichzeitig garantieren kann (Beweis siehe Gilbert und Lynch (2002)). Systeme, die auf dem Dynamo-Ansatz basieren, werden deswegen auch „Eventually Consistent“ (vgl. Vogels (2008)) genannt, da sie nicht zu jedem Zeitpunkt die Konsistenz der Daten garantieren (wobei hier mit Konsistenz gemeint ist, dass mehrere abfragende Klienten denselben Datensatz sehen). Sollten zwei konkurrierende Versionen eines Datensatzes vorhanden sein, werden beide an den Klienten ausgeliefert und die abfragende Anwendung muss sich um das Zusammenfügen der Daten kümmern. Das oben erwähnte Datenbanksystem Hibari bietet bspw. primär die Eigenschaften Konsistenz und Verfügbarkeit. Im Kontrast zu ACID bei traditionellen Datenbanksystemen findet sich für diese Art von verteilten Systemen auch die Bezeichnung BASE (Basically Available, Soft state, Eventual consistency).

Eine Implementierung in Erlang, die sich auf die Ideen von Dynamo stützt, ist Riak. Interessant für die Vorlesung „Verteilte Systeme“ ist die Kombination von mehreren Ansätzen aus dem Bereich der verteilten Systeme zu einem System. Zum Einsatz kommen z.B. Vector Clocks für die Versionierung von Datensätzen (zwei Blogeinträge[51] [52] beschreiben dies genauer), Distributed Hash Tables, Replikation und Hinted Handoff für die Fehlerhandhabung. Der Sourcecode ist im Online-Repository[53] von Basho, der Firma hinter Riak, verfügbar und im Wiki des Projektes[54] gibt es weitere Erläuterungen zu den eingesetzten Techniken.

Für die Einbindung von Riak in Anwendungen, bzw. das Abfragen der Datenbasis, wird eine auf den oben beschriebenen Protocol Buffers basierende Schnittstelle sowie eine REST Schnittstelle, die auf dem ebenfalls oben erwähnten Webmachine aufbaut, bereitgestellt.

Die Datenabfrage bei komplexen Anfragen an die Datenbasis basiert auf dem von Google veröffentlichten, verteilten Algorithmus MapReduce (Dean und Ghemawat (2004)). Die Grundidee des Algorithmus besteht darin, im Map Schritt Werte einzusammeln, die im Reduce Schritt zum Endergebnis zusammengefügt werden. Die Map-Phase wird dabei auf mehrere Nodes aufgeteilt, sodass eine parallele Berechnung ermöglicht wird. In Riak wird zudem versucht die Daten-Lokalität zu erhöhen, indem die Berechnung der Map-Phase auf die Nodes aufgeteilt wird, auf denen die entsprechenden Daten liegen, sodass weniger Kommunikations-Overhead entsteht[55].

In Riak finden einige Techniken und Algorithmen aus dem Bereich der verteilten Systeme Anwendung, sodass ein Riak- (bzw. Dynamo-) Klon eine Grundlage für die dritte bzw. vierte Aufgabe darstellen könnte. Alle verwendeten Techniken zu implementieren wäre sicher zu viel für das Praktikum, aber eine Auswahl in einer Basisversion eines Distributed Key/Value Store zu implementieren würde die praktische Anwendung von Algorithmen aus

[51] http://blog.basho.com/2010/01/29/why-vector-clocks-are-easy/
[52] http://blog.basho.com/2010/04/05/why-vector-clocks-are-hard/
[53] http://hg.basho.com/riak/src
[54] http://wiki.basho.com/display/RIAK/Riak
[55] http://wiki.basho.com/display/RIAK/MapReduce

der Vorlesung verdeutlichen. Eine kleine Artikelserie[56], die leider nicht zu Ende geführt wurde, beschäftigt sich mit den ersten Schritten so eines Projektes und der Quellcode der vorgestellten Lösung ist ziemlich kompakt und leicht verständlich.

6.3 Zusammenfassung und Bewertung

In dieser Arbeit wurde die Eignung der Programmiersprache Erlang für den Einsatz in der Vorlesung „Verteilte Systeme“ und dem zugehörigen Praktikum untersucht. In der Einführung wurde die Syntax von Erlang erläutert und auf Besonderheiten bei der Spracheinarbeitung hingewiesen. Es wurde gezeigt, dass sich die ersten beiden Aufgaben des Praktikums zur Vorlesung „Verteilte Systeme“ in Erlang lösen lassen. Es wurden mehrere Versionen mit unterschiedlichen Schwerpunkten bei der Implementierung vorgestellt sowie die Zusammenarbeit mit anderen Programmiersprachen untersucht. Bei der Vorstellung der unterschiedlichen Lösungen wurde auf Besonderheiten aufmerksam gemacht, auf die der Autor bei der Implementierung gestoßen ist. Im letzten Kapitel wurde auf Konzepte in Erlang eingegangen, die im Rahmen der Vorlesung von Interesse sind und ein Vergleich mit den Konzepten angestellt, die durch den Einsatz von CORBA in den Mittelpunkt gerückt sind. Der Ausblick hat gezeigt, dass es im Umfeld von Erlang eine Reihe an weiteren, für die Vorlesung interessante Themen gibt, wie z.B. andere RPC-Lösungen oder verteilte Datenbanken. Aufbauend auf diesen Ergebnissen folgt an dieser Stelle die abschließende Bewertung zur Eignung von Erlang im Hinblick auf den Einsatz im Umfeld der Vorlesung „Verteilte Systeme“ an der HAW Hamburg.

Aus dem kommerziellen Bereich gibt es mehrere Erfahrungsberichte zur Einführung von Erlang. Ulf Wiger berichtet von den Erfahrungen aus dem Einsatz von Erlang bei Ericsson im Zuge der Entwicklung der AXD 301. Er beschreibt eine Produktivitätssteigerung beim Einsatz von Erlang, verglichen mit den anderen eingesetzten Sprachen, z.B. C++, da die Erlang Lösungen im Vergleich weniger Code Zeilen für die gleiche Funktionalität brauchen. Im Hinblick auf den Lernaufwand schreibt er „Finally, we have seen that programmers - even those with no previous exposure to functional languages - learn Erlang quickly ...“ (Wiger (2001)).

Auch in Fritchie u. a. (2000) wird die einfache Erlernbarkeit von Erlang hervorgehoben: „Even in isolation, a decent programmer can quickly come up to speed on the basics of Erlang, with greater ease than with many other more popular languages.“. Allerdings bezieht sich dieses Zitat auf den Kern der Sprache, für die Feinheiten der OTP-Bibliothek wird eine umfangreichere Einarbeitungszeit erwähnt. Dies ist im Kontext des Einsatzes von Erlang in der Vorlesung aber nicht relevant, da diese Arbeit gezeigt hat, dass eine Konzentration auf den Kern der Sprache im Praktium ausreichend und sinnvoll ist.

[56] http://lethain.com/feeds/series/a-distributed-key-value-store/

In Nyström u. a. (2007) wird der Einfluss der Sprachwahl auf die Komplexität und den Code-Umfang eines verteilten Systems anhand von konkreten Implementierungen von Teilen eines Telekommunikationssystems in den Sprachen Erlang, GdH Glasgow distributed Haskell und C++(mit CORBA und UDP) untersucht. Diese Untersuchung kommt zu dem Ergebnis, dass die im Vergleich zur C++/CORBA Lösung höherleveligen Kommunikationsmöglichkeiten von Erlang sowohl die Anwendungsgrösse als auch die Komplexität z.B. bei der Koordination heruntersetzen. Diese Ergebnisse sind auch in Bezug auf einen Einsatz im Praktikum interessant, ermöglicht Erlang doch bei Bedarf die Implementierung komplexerer Aufgabenstellungen.

An der Universität von A Coruna in Spanien wird Erlang in einem Teil eines Wahlkurses zum Thema funktionale Programmierung eingesetzt, als Beispiel von funktionaler Programmierung in der „richtigen Welt“. Die Studenten haben schon vorher Erfahrungen mit funktionaler Programmierung gesammelt, mit Erlang kommen sie aber erst in diesem Kurs in Berührung. Bei den Studenten, die diesen Kurs absolvieren, wird Erlang sehr positiv aufgenommen: „The Erlang part of the functional programming course is really appreciated by students and they demand even more time dedicated to Erlang-related topics. [...] Students that really get into the language feel that they are going backwards when returning to tradicional but more popular approaches such as Java.“ Gulias (2005). Dieser Aussage kann sich der Autor nach der Beschäftigung mit Erlang im Rahmen dieser Arbeit anschließen.

Auf Grund der in dieser Arbeit vorgestellten Ergebnisse kann eine klare Empfehlung für einen Einsatz von Erlang im Rahmen der Vorlesung gegeben werden. Die Sprache ist, nach einer kleinen Hürde zu Anfang, bedingt durch den kaum bekannten funktionalen Programmierstil, einfach zu erlernen. Eine Entwicklungsumgebung lässt sich auch zuhause schnell aufsetzen. Die Programmierung nebenläufiger Programme in Erlang ist dank in die Sprache integrierter Konstrukte für den Einsatz von Prozessen und Message Passing einfach und mit wenig Grundwissen realisierbar. Die seiteneffektfreie Programmierung ohne Locks , ohne Shared State und mit Single Assignment Variablen erleichtert die Fehlersuche im Vergleich mit C++/CORBA ebenso wie die Möglichkeit über die interaktive Erlang-Shell mit Prozessen zu interagieren.

Auch Konzepte aus der Vorlesung, die nicht direkt in der Sprache vorhanden sind, lassen sich mit den gegebenen Sprachmitteln gut verständlich und mit wenig Programmcode darstellen. Ein Punkt, den es allerdings zu beachten gilt und auf den an dieser Stelle noch einmal hingewiesen werden soll, ist die Sicherheit beim Einsatz von Distributed Erlang. Dieses wurde für kontrollierte, sichere Netzwerkumgebungen entworfen und sobald sich Nodes miteinander verbunden haben, haben sie vollen Zugriff aufeinander, können also auch auf das entfernte System im Rahmen der Nutzerrechte, mit denen der Erlang-Node gestartet wurde, zugreifen. Erlang sollte also je nach Sicherheitsbedürfnis ggf. unter einem Benutzer mit wenig Rechten auf dem System gestartet werden.

Die Möglichkeiten der Fehlerbehandlung in verteilten Systemen, wie sie in Erlang inte-

griert sind, finden sich in kaum einem anderen System in dem Maße wieder. Sie sind leicht verständlich und einsetzbar.

Bedingt durch die wahrscheinliche Entwicklung im Bereich der CPUs ist Erlang zudem zukünftig auch für den Einsatz in nicht-verteilten Systemen, also auf Multi-(bzw. Many-) Core Systemen, interessant. Das System ist Open Source und wird aktiv weiterentwickelt, was die Möglichkeit zum Einsatz auch in Zukunft wahrscheinlich erscheinen lässt. Kurz vor Abgabe dieser Arbeit ist die neue stabile Version R14B erschienen.

Durch den funktionalen Ansatz im sequentiellen Teil von Erlang kommt desweiteren ein Programmierparadigma zum Einsatz, welches sonst im Lehrplan nur in Ansätzen vorgesehen ist.

Das Erlernen von Erlang ist somit auch über den Fokus der Vorlesung „Verteilte Systeme“ hinaus eine lohnenswerte Investition.

Literaturverzeichnis

[Agha 1985] AGHA, Gul A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. (1985). – URL http://dspace.mit.edu/bitstream/handle/1721.1/6952/AITR-844.pdf?sequence=2

[Armstrong 2002] ARMSTRONG, Joe: Getting Erlang to talk to the outside world. In: *ERLANG '02: Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*. New York, NY, USA : ACM, 2002, S. 64–72. – URL http://www.sics.se/~joe/ubf/site/ubf.pdf. – ISBN 1-58113-592-0

[Armstrong 2003] ARMSTRONG, Joe: *Making reliable distributed systems in the presence of software errors*, The Royal Institute of Technology, Stockholm, Sweden, Dissertation, 2003. – URL http://www.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*. Raleigh : Pragmatic Programmers, 2007. – ISBN 978-1934356005

[Brewer 2000] BREWER, Eric A.: Towards robust distributed systems. In: *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM, 2000, S. 7. – ISBN 1-58113-183-6

[Cesarini und Thompson 2009] CESARINI, Francesco ; THOMPSON, Simon: *Erlang Programming: A Concurrent Approach to Software Development*. Sebastopol : O'Reilly Media, 2009. – ISBN 978-0596518189

[Claessen und Svensson 2005] CLAESSEN, Koen ; SVENSSON, Hans: A semantics for distributed Erlang. In: *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*. New York, NY, USA : ACM, 2005, S. 78–87. – ISBN 1-59593-066-3

[Dean und Ghemawat 2004] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: simplified data processing on large clusters. In: *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, USENIX Association, 2004, S. 137 – 149. – URL http://www.cs.princeton.edu/courses/archive/spring06/cos592/bib/map-reduce-dean04.pdf

[DeCandia u. a. 2007] DECANDIA, Giuseppe ; HASTORUN, Deniz ; JAMPANI, Madan ; KAKULAPATI, Gunavardhan ; LAKSHMAN, Avinash ; PILCHIN, Alex ; SIVASUBRAMANIAN, Swaminathan ; VOSSHALL, Peter ; VOGELS, Werner: Dynamo: Amazon's Highly Available Key-value Store. In: *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA : ACM, 2007, S. 205–220. – URL http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf. – ISBN 978-1-59593-591-5

[Fielding 2000] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000. – URL http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[Fritchie u. a. 2000] FRITCHIE, Scott L. ; LARSON, Jim ; CHRISTENSON, Nick ; JONESY, Debi ; ÖHMAN, Lennart: Sendmail Meets Erlang: Experiences Using Erlang for Email Applications. (2000). – URL http://www.erlang.se/euc/00/euc00-sendmail.pdf

[Gilbert und Lynch 2002] GILBERT, Seth ; LYNCH, Nancy: Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. In: *In ACM SIGACT News*, ACM, 2002. – URL http://people.csail.mit.edu/sethg/pubs/BrewersConjecture-SigAct.pdf

[Gulias 2005] GULIAS, Victor M.: Teaching Functional Programming and Erlang: The Galician Experience. (2005). – URL http://ftp.sunet.se/pub/lang/erlang/euc/05/Victor.pdf

[Hewitt u. a. 1973] HEWITT, Carl ; BISHOP, Peter ; STEIGER, Richard: A universal modular ACTOR formalism for artificial intelligence. In: *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973, S. 235–245

[Nyström u. a. 2007] NYSTRÖM, Jan ; TRINDER, Phil ; KING, David: Evaluating high-level distributed language constructs. In: *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. New York, NY, USA : ACM, 2007, S. 203–212. – ISBN 978-1-59593-815-2

[Svensson und Fredlund 2007a] SVENSSON, Hans ; FREDLUND, Lars-Ake: A more accurate semantics for distributed erlang. In: *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*. New York, NY, USA : ACM, 2007, S. 43–54. – ISBN 978-1-59593-675-2

[Svensson und Fredlund 2007b] SVENSSON, Hans ; FREDLUND, Lars-Ake: Programming distributed erlang applications: pitfalls and recipes. In: *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*. New York, NY, USA : ACM, 2007, S. 37–42. – ISBN 978-1-59593-675-2

[Vinoski 2008] VINOSKI, Steve: Convenience Over Correctness. In: *IEEE Internet Computing* 12 (2008), S. 89–92. – URL http://steve.vinoski.net/pdf/IEEE-Convenience_Over_Correctness.pdf. – ISSN 1089-7801

[Vogels 2008] VOGELS, Werner: Eventually Consistent. In: *Queue* 6 (2008), Nr. 6, S. 14–19. – URL http://portal.acm.org/ft_gateway.cfm?id=1466448&type=pdf&coll=GUIDE&dl=GUIDE&CFID=102724662&CFTOKEN=82771226. – ISSN 1542-7730

[Waldo u. a. 1994] WALDO, Jim ; WYANT, Geoff ; WOLLRATH, Ann ; KENDALL, Sam: A Note on Distributed Computing. (1994). – URL http://labs.oracle.com/features/tenyears/volcd/papers/5Waldo.pdf

[Wiger 2001] WIGER, Ulf T.: Four-fold Increase in Productivity and Quality. (2001). – URL http://www.erlang.se/publications/Ulf_Wiger.pdf

Anhang

Hier finden sich die ursprünglich im Praktikum zur Vorlesung „Verteilte Systeme“ eingesetzten Aufgabenstellungen zu Aufgabe 1[57] und Aufgabe 2[58] sowie der Inhalt der beiliegenden CD.

Inhalt der CD

- Die Eclipse-Projekte mit den vorgestellten Lösungen der Aufgaben
- Diese Arbeit im PDF-Format
- Quellen, die in digitaler Form vorliegen
- Schnappschüsse der zitierten Webseiten
- Die Folien und Aufgaben aus der Vorlesung „Verteilte Systeme“

Aufgabe 1: Message of the Day

Implementieren Sie eine kleine eigene Client/Server-Anwendung; ein Server verwaltet Nachrichten (Textzeilen): die Nachrichten des Tages, die von unterschiedlichen Clients (den Redakteuren) ihm zugesendet werden. Diese Nachrichten sind eindeutig nummeriert. Clients fragen in bestimmten Abständen als Leser die aktuellen Nachrichten ab. Damit ein Client nicht immer alle Nachrichten erhält, erinnert sich der Server an einen Client und welche Nachricht er ihm zuletzt zugestellt hat. Meldet sich dieser jedoch eine gewisse Zeit lang nicht, so vergisst der Server diesen Client. Im Nachfolgenden die detaillierte Beschreibung.

[57] http://users.informatik.haw-hamburg.de/ klauck/VerteilteSysteme/aufg1-rmi.html
[58] http://users.informatik.haw-hamburg.de/ klauck/VerteilteSysteme/aufg2-jt.html

Funktionalität

Server

1. Ein Client bekommt auf Anfrage gemäß Nachrichtennummerierung eine noch nicht an ihn ausgelieferte und beim Server bekannte Textzeile geliefert. In einem Flag wird ihm mitgeteilt, ob es noch weitere, für ihn unbekante Nachrichten gibt.

2. Ein Client, der seit ** Sekunden keine Abfrage mehr gemacht hat, wird beim Server vergessen (unabhängig davon, wann er die letzte Textzeile als Redakteur übertragen hat!), also dort gelöscht. Bei einer erneuten Abfrage (nach dem Vergessen) wird er wie ein unbekannter Client behandelt. Der timeout von ** Sekunden ist für alle Clients gleich und wird über die GUI des Servers gesetzt.

3. Der Server hält nur die letzten *** Textzeilen vor. Alle anderen werden gelöscht. Die Anzahl der Textzeilen *** wird dem Server beim Start oder per GUI übergeben.

4. Die Textzeilen werden vom Server durchnummeriert (beginnend bei 0) und stellen eine eindeutige ID für jede Textzeile dar. Ein Client hat sich beim Server vor dem Versenden einer Textzeile diese Nummer zu besorgen! In der Zustellung einer Nachricht an den Server ist diese Nummer jeder Textzeile voranzustellen (rechtsbündig ohne führenden Nullen, sondern mit führenden blank), d.h. die Nummer wird der Textzeile einfach vorne (als vier Zeichen langer Text) an gestellt resultierend in einer neuen Textzeile. Als Vereinbarung werden vier Zeichen dafür reserviert (ggf. wird bei Null wieder angefangen zu zählen). Der Server merkt sich nicht, von wem eine Textzeile gesendet wurde, insbesondere schaut er nicht in die Textzeile hinein! Der Server fügt einer empfangenen Nachricht die Empfangszeit vorne an.

5. Da die Textzeilen bzgl. der Nummerierung in zusammenhängender Reihenfolge erscheinen sollen und Nachrichten mit Textzeilen verloren gehen können bzw. in nicht sortierter Reihenfolge eintreffen können, arbeitet der Server intern mit einer Deliveryqueue und einer Holdbackqueue. In der Deliveryqueue stehen die Nachrichten, die an Clients ausgeliefert werden können, maximal *** viele Textzeilen. In der Holdbackqueue stehen alle Textzeilen, die nicht ausgeliefert werden dürfen, d.h. zwischen der größten Nummer in der Deliveryqueue und der kleinsten Nummer in der Holdbackqueue fehlt mindestens eine Nummer. Der Server fügt entweder nach einem timeout (3 Sekunden) oder bei einem zugrossen Abstand der Nummern (erste Textzeilennummer der Holdbackqueue - erwartete Textzeilennummer in der Deliveryqueue > 10) eine Fehlertextzeile in die Deliveryqueue ein, etwa: »»»Fehler: »»Nachricht 4 vermisst (8): 20:32:53,750, damit die Holdbackqueue eingetroffene Textzeilen an die Deliveryqueue übertragen kann. Sollte die vermisste Nachricht jedoch noch rechtzeitig vor dem Löschen eintreffen, ersetzt der Server (sofern vorhanden) die Fehlertextzeile in der Deliveryqueue durch die neue Nachricht. Es wird nur die Lücke zwischen Deliveryqueue und Holdbackqueue gefüllt (bis zur ersten Textzeilenummer in der

Holdbackqueue), es werden keine Lücken innerhalb der Holdbackqueue gefüllt, also Lücken die nach der kleinsten in der Holdbackqueue vorhandenen Textzeilennummer vorkommen!

6. Der Server terminiert sich, wenn die Differenz von aktueller Systemzeit und Zeit der letzten Abfrage eines Clients länger als seine Wartezeit beträgt, d.h. seine Wartezeit wird durch Abfragen der Clients erneut gestartet bzw. bestimmt die maximale Zeit, die der Server üngenutzt"läuft. Der Server meldet sich beim Namensdienst vor der Terminierung ab!

7. Dieser Dienst wird unter dem Namen Messages bei dem Namensdienst registriert.

8. Der Server ist in Java zu implementieren und muss auf jedem Rechner im Labor startbar sein! Erzeugen Sie dazu eine ausführbare Datei! (Ein Ordner mit den zugehörigen *.class Dateien reicht aus, es muss keine *.jar erzeugt werden, siehe hierzu z.B. ein Tutorial von Sun: jar-Tutorial. bzw. jar cvfm <jar-dateiname> Manifest.txt *.class broadcast/*.class).

Client

1. Ein Client sendet in bestimmten Abständen, d.h. alle **** Sekunden, eine Textzeile an den Server, die seinen Namen (sein Rechnernamen (zB lab18), die Praktikumsgruppe (zB 1) und die Teamnummer (zB 03), also "lab18103"beinhalten) und seine aktuelle Systemzeit (die der Sendezeit enstprechen soll) beinhaltet und ggf. anderen Text, zum Beipiel "lab18103 Client 10:33:40 Nachrichteninhalt". Der Abstand von **** Sekunden wird bei der Initialisierung des Clients (z.B. beim Aufruf) gesetzt und nach dem Senden von 5 Textzeilen jeweils um ca. 50% (mindestens 1 Sekunde) per Zufall vergrößert oder verkleinert. Die Zeit **** darf nicht unter 1 Sekunde "rutschen".

2. Der Client fragt nach dem Senden von 7 Textzeilen solange aktuelle Textzeilen beim Server ab, bis er alle erhalten hat,d.h. alle bisher noch nicht erhaltene und beim Server noch vorhandene Textzeilen, und stellt sie in seiner GUI dar. Dabei gelten die vom Client gesendeten Textzeilen als bei diesem Client unbekannte Textzeilen! Alle unbekannten Textzeilen werden ihm also einzeln übermittelt bzw. pro Anfrage erhält er nur eine unbekannte Textzeile.

3. Bei der Initialisierung des Clients (z.B. beim Aufruf) wird seine Lebenszeit gesetzt. Ist diese Zeit erreicht, terminiert sich der Client selbst.

4. Der Client ist in C++ zu implementieren und muss auf jedem Rechner im Labor startbar sein! Erzeugen Sie dazu eine ausführbare Datei!

GUI

1. Server-GUI:

a) Der Server hat eine GUI über die die benötigten steuernden Werte gesetzt werden; diese sind zumindest der timeout für das löschen von Clients und die Warteszeit des Servers, nachdem der letzte Client Textzeilen abgefragt hat.

b) In der Server-GUI sind zumindest die letzten 15 erhaltenen Textzeilen und alle aktuell bekannten Clients darzustellen.

c) Bei den Textzeilen (die deren ID bzw. Nummer beinhaltet) ist deren Ankunftszeit beim Server mit auszugeben. (z.B. mittels java.text.SimpleDateFormat)

d) Bei den Clients ist deren letzte Zugriffszeit (im Sinne der Abfrage der Textzeilen!) mit auszugeben.

e) Diese Ausgaben sind ebenfalls alle in eine Datei Server.log zu schreiben.

2. Client-GUI

a) Der Client hat eine GUI (Standardausgabe reicht).

b) Die GUI zeigt die Textezeilen an, die vom Server gesendet werden inklusive seiner Abfragezeit (Sendezeitpunkt der Serveranfrage).

c) In der GUI ist anzuzeigen, wenn der Client eine Textzeile an den Server überträgt.

d) Alle Ausgaben sind ebenfalls in eine Datei client<Clientname>.log , wobei <Clientname> der Name des Clients ist (z.B. clientATlab22103.log), zu schreiben.

Fehlerbehandlung

1. Wann immer die Besorgung einer IOR schief geht, ist eine Fehlermeldung auszugeben und auch in der zugehörigen Protokolldatei zu vermerken. Handelt es sich um den Namensdienst, beenden sich die Prozesse. Handelt es sich um den Server, wartet der Client eine bestimmte Zeit und vertsucht es dann nochmal. Tritt wieder ein Fehler auf, beendet er sich (mit Fehlermeldung!).

2. Tritt während dem Lauf ein Fehler in der Kommunikation mit dem Server auf, erzeugt der Client eine entsprechende Fehlermeldung, die auch in der zugehörigen Protokolldatei zu vermerken ist. Er besorgt sich dann beim Namensdienst eine neue IOR. Tritt dieser Fehler insgesamt während der Lebenszeit des Clients öfters als 10-mal auf, terminiert sich der Client mit einer entsprechenden Fehlermeldung.

IDL

Die folgende IDL beschreibt die Schnittstelle des Servers für die Clients.

```
module broadcast{
  interface messages {
    /* Abfragen aller Nachrichten */
    string getmessages(in string rechnerid, out long msgID,
      out boolean getall);
    /* Senden einer Nachricht */
    long dropmessage(in string message, in long msgID);
    /* Abfragen der eindeutigen Nachrichtennummer */
    long getmsgid(in string rechnerid);
  };
};
```

`getmessages`: Fragt beim Server eine aktuelle Textzeile ab. rechnerid stellt den eindeutigen Namen des rufenden Clients dar. Dieser besteht aus der Zeichenkette des Rechneres, z.B. "lab18", der Praktikumsgruppe (1, 2 oder 3) und der Teamnummer (zu erfragen, z.B. 03), z.B. "lab18203". Als Rückgabewert erhält er eine für ihn aktuelle Textzeile zugestellt. In der Variabln msgID steht die Nachrichtennummer und in der Variablen getall signailiert der Server ihm, ob noch für ihn aktuelle Nachrichten vorhanden sind.getall.value = false bedeutet, es gibt noch weitere aktuelle Nachrichten, getall.value = true bedeutet, dass es keine aktuellen Nachrichten mehr gibt, d.h. weiter Aufrufe von getmessages sind nicht notwendig.

`dropmessage`: Sendet dem Server eine Textzeile, die den Namen des aufrufenden Clients und seine aktuelle Systemzeit, die durch eine 6-stellige Zahl repräsentiert wird (hh:mm:ss), und ggf. irgendeinem Text beinhaltet, sowie die zugeordnete (globale) Nummer der Textzeile (long msgID). In message sind also der Name, die Systemzeit und irgendein Text (als Nachricht des Clients) als eine Zeichenkette zusammengefasst, z.B. "lab18203 10:33:40 Zummsel". Es gibt keine definierten Trennungssysmbole. Als Rückgabewert erhält er einen Fehlercode, mit 1 = OK und 0 = error.

Aufgabe 2: verteilter Algorithmus

In dieser Aufgabe ist ein einfacher verteilter Algorithmus und dessen Verwaltung/Koordination zu implementieren. Jeder Ärbeits-"Prozess (ggT-Prozess) durchläuft den gleichen Algorithmus! Mit diesem Algorithmus kann man z.B. mit 42 ggT-Prozessen den ggT von 42 Zahlen "gleichzeitig"(nebenläufig) bestimmen. Alle benötigten Hintergrundinformationen sowie den Algorithmus finden Sie in der Datei ablauf.zip.

Lesen Sie sich die Aufgabenstellung sorgfaltig durch, damit nicht evtl. aufrgund einer Nachlässigkeit die erfolgreiche Bearbeitung gefährdet ist!

Aufgabenstellung

Das System für den verteilten Algorithmus ist so ausgelegt, dass es für eine längere Zeit installiert wird und dann für mehrere ggT-Berechnungen zur Verfügung steht, ohne dass die ggt-Prozesse oder der Koordinator neu gestartet werden müssen! Für die Implementierung werden im Wesentlichen drei Module benötigt:

1. Der Koordinator, der den verteilten Algorithmus verwaltet. Dazu gehören das "hochfahren"des Systems, den Start einer ggT-Berechnung, die Feststellung der Terminierung einer ggT-Berechnung und das "herunterfahren"des Systems. Der Koordinator verfügt über eine GUI, in der der Algorithmus "beobachtet"werden kann und in der die benötigten steuernden Werte gesetzt werden können.
2. Den Starter, der im Wesentlichen das Starten von Prozessen auf für den Koordinator entfernten Rechnern ermöglicht.
3. Der ggT-Prozess, der die eigentliche Arbeit leistet, also die Berechnung des ggT.

Am Anfang wird das System hochgefahren. Dazu ist als erstes ein Namensdienst zu starten. Dann wird zunächst der Koordinator gestartet.

Das System geht nun in die Initialisierungsphase: Die Starter werden gestartet und erfragen beim Koordinator die steuernden Werte für die ggT-Prozesse. Diese werden gemäß den Vorgaben des Koordinators gestartet. Der Starter beendet sich dann selbst. Die ggT-Prozesse melden sich beim Koordinator an. Der Koordinator baut einen Ring auf und informiert die ggT-Prozesse über ihre Nachbarn. Das System ist nun bereit, ggT-Aufgaben zu lösen. Die ggT-Prozesse gehen in den Zustand "bereit".

Das System geht in die Arbeitsphase: Dazu wird zunächst das System für einen Lauf initialisiert: Der Koordinator informiert die ggT-Prozesse über ihre Werte #Mi. Er wählt 3 ggT-Prozesse per Zufall aus, die mit der Berechnung beginnen sollen. Meldet ein ggT-Prozess die Terminierung der aktuellen Berechnung, so erhält der Koordinator gleichzeitig von ihm das Endergebnis der Berechnung. Die ggT-Prozesse stehen dann für weitere Berechnungen zur Verfügung!

Erhält während einer Berechnung ein ggT-Prozess ** Sekunden lang keine Zahl y, startet er eine Terminierungsabstimmung: Dazu befragt er seinen rechten Nachbarn, ob dieser bereit ist, zu terminieren. Antwortet dieser mit Ja, sendet er dem Koordinator eine entsprechende Mitteilung über die Terminierung der aktuellen Berechnung. Wird er selbst in diesem Stadium, also als Initiator einer Abstimmung, von seinem linken Nachbarn gefragt, ob er terminieren kann, beantwortet er dies mit Ja. Wird er während seiner normalen Arbeitsphase,

also wenn er nicht Initiator einer Abstimmung ist, nach seiner Bereitschaft zur Terminierung gefragt, antwortet er mit Nein, wenn seit der letzten Zusendung einer Zahl weniger als **/2 Sekunden vergangen sind. Ansonsten leitet er die Frage weiter an seinen rechten Nachbarn und leitet die zugehörige Antwort zurück an seinen linken Nachbarn.

Das System geht in die Beendigungsphase: Der Starter ist bereits in der Initialisierungsphase beendet worden. Die ggT-Prozesse werden vom Koordinator über die Beendigung des Systems informiert. Stellt ein ggT-Prozess fest, dass er sich beenden soll (nur durch explizites close-Kommando durch den Koordinator!), hat er dies unverzüglich zu tun, unabhängig von seinem aktuellen Zustand. Die einzige dann noch zulässige Aktion ist, den Koordinator über die Beendigung zu informieren und sich beim Namensdienst wieder abzumelden. Hat der Koordinator von allen gemeldeten ggT-Prozessen die Information über deren Beendigung erhalten, beendet er sich selbst. Hat der Koordinator nach 5 Sekunden noch nicht alle Bestätigungen erhalten, beendet er sich selbst mit einer entsprechenden Fehlermeldung.

Der Ablauf ist in einer Powerpointpräsentation visualisiert (ablauf.zip); diese Präsentation wird innerhalb einer Vorlesung zusammen mit der Aufgabenstellung erklärt werden.

Funktionalität

Koordinator

1. Der Koordinator ist als Dienst chef beim CORBA-Namensdienst registriert.
2. Die benötigten steuernden Werte werden über die GUI eingegeben. Er hat als default-Wert bereits beliebige Werte gesetzt. Die Werte sind: ein Intervall für die Anzahl der ggT-Prozesse auf einem Rechner, ein Intervall für die Verzögerungszeit der ggT-Prozesse, ein Wert für den ** timeout und der gewünschte ggT.
3. Durch die GUI wird manuell per Knopfdruck der Koordinator in den Zustand ïnitial"gesetzt. Erst dann können sich Starter oder ggT-Prozesse bei ihm melden.
4. Ist der Koordinator im Zustand ïnitial", gibt er den Startern auf Anfrage die benötigten Informationen über Anzahl der zu startenden ggT-Prozesse, deren jeweilige Verzögerungszeit und deren **-timeout. Er wählt jeweils (für jeden Starter) dazu per Zufall einen Wert aus einem in der GUI vorgegebenem Intervall aus.
5. In seiner GUI werden die sich angemeldeten ggT-Prozesse angezeigt.
6. Wird manuell der Koordinator per Knopfdruck in den Zustand "bereit"versetzt, gibt er keinem Starter mehr Auskunft und registriert keine ggT-Prozesse mehr. Er baut nun den Ring auf, indem er per Zufall die ggT-Prozesse auswählt. Erst danach geht er in den Zustand "bereit".

7. Ist der Koordinator im Zustand "bereitïnformiert er per Knopfdruck alle ggT-Prozesse über deren jeweilige Startwerte und startet die Berechnung. Für die Startwerte liest er aus der GUI einen aktuellen gewünschten ggT aus und multipliziert ihn wie folgt: gewünschter_ggT * Zufallszahl_1_bis_100 * Zufallszahl_1_bis_100.
8. Er wählt dann per Zufall drei Prozesse aus, denen er zum Starten der Berechnung eine Zahl sendet, die sich wie folgt berechnet: gewünschter_ggT * Zufallszahl_100_bis_10000.
9. In seiner GUI zeigt der Koordinator die Nachrichten der ggT-Prozesse an, insbesondere zeigt er an, welcher Prozess ihm die Terminierung der aktuellen Berechnung gemeldet hat und was das Endergebnis ist.
10. Per Knopfdruck kann der Koordinator dann in den Zustand "beenden"versetzt werden oder eine neue ggt-Berechnung starten.
11. Ist der Koordinator im Zustand "beendenïnformiert er die ggT-Prozesse über die Beendigung und wartet maximal 5 Sekunden auf deren Bestätigung.
12. Haben alle ggT-Prozesse die Beendigung bestätigt, beendet er sich selbst mit entsprechender Meldung. Hat er nach 5 Sekunden nicht alle Bestätigungen erhalten, beendet er sich mit entsprechender Meldung. In seiner GUI gibt er die bei der Bestätigung von den ggT-Prozessen mitgesendeten Informationen aus.
13. Koordinator ist in Java zu implementieren und muss auf jedem Rechner im Labor startbar sein! Erzeugen Sie dazu eine ausführbare Datei!

Starter

1. Der Starter erfragt beim Koordinator die steuernden Werte, bis er diese vom Koordinator erhält oder er zehn Versuche durchgeführt hat.
2. Der Starter startet die ggT-Prozesse mit den zugehörigen Werten: der Verzögerungszeit, die Terminierungszeit und der Startnummer dieses Prozesses (also der wievielte gestartete ggT-Prozess er ist) und die Nummer seines Starters.
3. Sind alle ggT-Prozesse gestartet beendet sich der Starter dann selbst.
4. Beim starten des Starters wird ihm seine Nummer mitgegeben sowie die TeamID (die zu erfragen ist).
5. Der Starter ist in Java zu implementieren und muss auf jedem Rechner im Labor startbar sein! Sie können hier auch meine Vorgabe verwenden: starter.jar (Aufruf mit java -jar starter.jar <starterNR> <rechnername> <port> <pfadname des programms> <programmname>).

ggT-Prozess

1. Der ggT-Prozess ist mit dem Namen slave????? beim CORBA-Namensdienst registriert. Dabei ist ????? eine Zahl, die sich wie folgt zusammensetzt: <PraktikumsgruppenID><TeamID><Nummer des ggT-Prozess><Nummer des Starters>, also z.B. ist in der Praktikumsgruppe 3 (einstellige Zahl) von dem Team 03 (zweistellige Zahl) ein dritter ggT-Prozess (einstellige Zahl) von ihrem ersten Starter (einstellige Zahl) gestartet worden, so erhält dieser ggT-Prozess den Namen slave30331. In der Kommunikation zwischen ggT-Prozess und Koordinator oder anderen ggT-Prozessen wird stets auf den Zusatz slave verzichtet, also nur die Nummer übertragen!

2. Der ggT-Prozess meldet sich beim Koordinator mit seinem Namen an, bis dieser die Anmeldung akzeptiert oder er es zehn mal probiert hat.

3. Der ggT-Prozess erwartet nun steuernde Werte vom Koordinator (Nachbarn und seine Zahl Mi). Er hat bereits default-Werte zu Anfang gesetzt, die beliebig/frei gewählt wurden.

4. Der ggT-Prozess reagiert nun auf die jeweiligen Nachrichten. Wenn er z.B. eine Zahl erhält führt er den ggt-Algorithmus aus.

5. Ändert sich seine Zahl dadurch (also hat er echt etwas berechnet), informiert er zusätzlich den Koordinator darüber, indem er diesem seinen Namen, seine neue Zahl und die aktuelle Systemzeit überträgt.

6. Ändert sich seine Zahl dadurch nicht und war es das erste mal, dass er eine Zahl erhalten hat, gibt er eine Fehlermeldung aus und sendet seinem rechten Nachbarn seine Zahl #Mi. Diese Aktion wird ausschliesslich beim ersten Aufruf einer neuen Berechnung durchgeführt. Ansonsten macht der ggt-Prozess in diesem Fall gar nichts!

7. Für eine Berechnung braucht er jedoch eine gewisse Zeit (die Verzögerungszeit), die ihm vom Starter bei der Initialisierung mitgegeben wurde. Dies simuliert eine größere, Zeit intensivere Aufgabe. Der ggT-Prozess sollte in dieser Zeit einfach nichts tun (usleep).

8. Der ggT-Prozess beobachtet die Zeit seit dem letzten Empfang einer Zahl. Hat diese ** Sekunden überschritten, startet er eine Terminierungsanfrage. Es wird nur genau eine Terminierungsanfrage gestartet; eine weitere kann erst gestartet werden, wenn die erste (positiv oder negativ) beendet wurde!

9. Ist Terminierungsanfrage erfolgreich durchgeführt, sendet er dem Koordinator eine Mitteilung über die Terminierung der aktuellen Berechnung, die seinen Namen, den errechneten ggT und seine aktuelle Systemzeit beinhaltet. Die Abstimmung arbeitet wie folgt:

    a) Ein ggT-Prozess erhält die Anfrage nach der Terminierung und er ist nicht der Initiator: ist seit dem letzten Empfang einer Zahl mehr als **/2 (** halbe) Sekunden vergangen, dann leitet er die Anfrage an seinen rechten Nachbarn weiter

(implizites Zustimmen) und gibt dessen Antwort zurück an seinen linken Nachbarn. Sonst antwortet er seinem linken Nachbarn direkt mit Nein (explizites ablehnen).

b) Erhält ein initiierende Prozess von seinem linken Nachbarn die Anfrage nach der Terminierung, antwortet er diesem direkt mit Ja ohne weitere Aktionen durchzuführen.

c) Ein initiierender ggt-Prozess erhält ein Ja auf seine Terminierungsanfrage: Der ggt-Prozess meldet dem Koordinator die Terminierung und wartet anschliessend auf eine neue Berechnungsaufgabe oder das explizite close-Kommando zum Beenden.

10. Ein ggt-Prozess zählt intern die Anzahl aller Terminierungsanfragen (inklusive seiner initiierten Anfragen) sowie die Anzahl aller Antworten. In einer dritten Variablen werden Terminierungsanfragen gezählt (+1) und Antworten abgezogen (-1) .

11. Der ggT-Prozess ist in C++ zu implementieren und muss auf jedem Rechner im Labor startbar sein! Erzeugen Sie dazu eine ausführbare Datei!

GUI

1. Der Koordinator hat eine GUI über die die benötigten steuernden Werte gesetzt werden.

   a) In der GUI sind die angemeldeten ggT-Prozesse zumindest so lange anzuzeigen, bis der Koordinator in den Zustand "bereit"übergeht.

   b) Die GUI zeigt während einer Berechnung die Nachrichten der ggT-Prozesse an.

   c) Alle ausgegebenen Daten werden in der Datei chef.log mit protokolliert.

2. Der Starter und die ggT-Prozesse protokollieren ihre Ausgaben in der Datei slave?????.log bzw. starter???.log. Dabei ist die Identität des ggT-Prozesses bzw. die ersten drei Ziffern der ggT-Prozesse eines Starters. 1. Zu protokollieren sind alle Informationen über den aktuellen internen Zustand, d.h. der Starter protokolliert die erhaltenen steuernden Werte und die Aufrufe der ggT-Prozesse. 2. Die ggT-Prozesse protokollieren ihre Zustandsänderungen und alle Nachrichten, die sie erhalten oder versenden.

3. Alle Prozesse protokollieren in den *.log Dateien alle ihre Fehlermeldungen.

Fehlerbehandlung

1. Wann immer die Besorgung einer IOR schief geht, ist eine Fehlermeldung auszugeben und auch in der zugehörigen Protokolldatei zu vermerken. Handelt es sich um einen ggT-Prozess, streicht der Koordinator ihn, sofern der Ring noch nicht aufgebaut

wurde, aus seiner Liste; ein ggT-Prozess, der die IOR einer seiner Nachbarn oder des Koordinators nicht erhält, beendet sich mit einer entsprechenden Fehlermeldung.

2. Tritt während dem Lauf ein Fehler in der Kommunikation auf, erzeugt der rufende Prozess eine entsprechende Fehlermeldung, die auch in der zugehörigen Protokolldatei zu vermerken ist. Er beendet sich dann selbst. Handelt es sich hier um den Koordinator, beendet er alle ggT-Prozesse.

IDL

Die folgende IDL beschreibt die Schnittstelle des Koordinators.

```
module chef{
  interface koordinator{
   /* initial */
   long getsteeringval(out long pnum, out long wtime,
     out long term);
   long hello(in long pid);
   /* bereit */
   oneway void briefmi(in long pid, in long pm,
     in long ptime);
   oneway void briefqu(in long pid, in long ptime);
   oneway void briefre(in long pid, in long ptime);
   oneway void terminated(in long pid, in long ggt,
     in long ptime, in long rcount, in long qcount);
   /* beenden */
   oneway void closed(in long pid, in long pm,
     in long ptime);
  };
};
```

`getsteeringval`: Fragt beim Koordinator die steuernden Werte ab; diese sind die Anzahl der ggT-Prozesse (pnum), die Verzögerungszeit für die ggT-Prozesse (wtime) in Sekunden und den **-timeout in Sekunden (term). Als Rückgabewert erhält er einen Fehlercode, mit 1 = OK und 0 = error.

`hello`: Ein ggT-Prozess sendet dem Koordinator seinen Namen, wobei auf den Zusatz slave verzichtet wird. Als Rückgabewert erhält er einen Fehlercode, mit 1 = OK und 0 = error.

`briefmi`: Ein ggT-Prozess sendet dem Koordinator seinen Namen (pid), seine aktuelle Zahl Mi (pm) und seine aktuelle Systemzeit (ptime), die durch eine 6-stellige Zahl repräsentiert wird (hhmmss).

`briefre`: Ein ggT-Prozess sendet dem Koordinator seinen Namen (pid) und seine aktuelle Systemzeit (ptime), die durch eine 6-stellige Zahl repräsentiert wird (hhmmss), wenn er als Initiator eine Antwort auf seine Anfrage erhält. Die Methode ist per synchronized als kritischer Abschnitt zu schützen.

`briefqu`: Ein ggT-Prozess sendet dem Koordinator seinen Namen (pid) und seine aktuelle Systemzeit (ptime), die durch eine 6-stellige Zahl repräsentiert wird (hhmmss), wenn er eine Terminierungsanfrage initiiert. Die Methode ist per synchronized als kritischer Abschnitt zu schützen.

`terminated`: Ein ggT-Prozess, der die Terminierung einer Berechnung festgestellt hat, sendet dem Koordinator seinen Namen (pid), und den berechneten ggT (ggt) sowie seine aktuelle Systemzeit (ptime), die durch eine 6-stellige Zahl repräsentiert wird (hhmmss). Zudem sendet er seine query- (qcount) und response-Zähler (rcount).

`closed`: Ein ggT-Prozess sendet dem Koordinator zur Bestätigung, dass er sich beendet, seinen Namen (pid), und den berechneten ggT (ggt) sowie seine aktuelle Systemzeit (ptime), die durch eine 6-stellige Zahl repräsentiert wird (hhmmss).

Die folgende IDL beschreibt die Schnittstelle der ggT-Prozesse.

```
module ggt{
  interface unit{
   /* initial */
   long setneighbors(in long pidleft,
     in long pidright);
   oneway void setpm(in long pm);
   /* rechnen */
   oneway void sendy(in long y);
   oneway void query(in long pid);
   oneway void response(in long pid,in long y);
   /* beenden */
   oneway void close( );
  };
};
```

`setneighbors`: Der Koordinator teilt einem ggT-Prozess seinen linken (pidleft) und rechten (pidright) Nachbarn des logischen Ringes mit, indem er die entsprechenden Nummeren der Nachbarn (ohne Zusatz slave) mitteilt. Als Rückgabewert erhält er einen Fehlercode, mit 1 = OK und 0 = error.

`setpm`: Der Koordinator teilt einem ggT-Prozess seine interne Zahl Mi (pm) mit.

`sendy`: Ein ggT-Prozess (oder beim Start der Koordinator) sendet die Zahl Mi (y) an einen (benachbarten) ggT-Prozess.

`query`: Ein ggT-Prozess hat den **-timeout festgestellt und befragt seinen rechten Nachbarn, ob er mit der Terminierung einverstanden ist. Der Prozessnamen (pid) ist der Name des Initiators der Abstimmung und wird durchgereicht oder als Initiator gesetzt. Diese Methode ist nicht als kritischer Abschnitt zu schützen!

`response`: Ein ggT-Prozess beantwortet die Frage nach der Terminierung (y =) mit 0 für Nein und 1 für Ja. Der Prozessnamen (pid) ist der Name des Initiators der Abstimmung und wird durchgereicht oder als Initiator gesetzt. Diese Methode ist nicht als kritischer Abschnitt zu schützen!

`close`: Der Koordinator (oder im Fehlerfall ein ggT-Prozess) sendet einem ggT-Prozess die Aufforderung, sich zu beenden.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. September 2010 Sebastian Meiser