Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Andreas Wimmer

Migration von Prozessen in der Programmiersprache Erlang

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Wimmer

Migration von Prozessen in der Programmiersprache Erlang

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klauck
Zweitgutachter: Prof. Dr. Meisel

Eingereicht am: 28. Oktober 2013

Thema der Arbeit
Migration von Prozessen in der Programmiersprache Erlang

Stichworte
Prozess Migration, Verteilte Systeme, Erlang

Kurzzusammenfassung
Anwendungen werden in immer größerem Umfang verteilt, wobei immer mehr Wert auf Mobilität gelegt wird. Diese Arbeit beschäftigt sich daher mit der Umsetzung eines Systems zur Migration von Prozessen. Dabei sollen Prozesse von einer Maschine auf eine andere migriert werden, ohne ihren Zustand zu verlieren. Zur Umsetzung eines solchen Systems wird die Programmiersprache Erlang genutzt, die aufgrund ihrer Eigenschaften in verteilten Systemen gewählt wurde. Diese Arbeit zeigt zwei mögliche Umsetzungen eines solchen Systems und erklärt zudem, inwiefern Erlang die Entwicklung unterstützt hat.

Title of the paper
Process migration in Erlang

Keywords
Process migration, Distributed Systems, Erlang

Abstract
Distribution as well as mobility of applications get more and more important. This thesis examines the implementation of a system for process migration. During this migration, processes are to be moved from one machine to another without losing their state. Because of its characteristics in distributed systems, Erlang is used for implementing the system. This thesis shows two possible approaches of implementing such a system and explains in which way Erlang was of assistance during developement.

Inhaltsverzeichnis

Listings

1 Einführung

1.1 Motivation

Während viele Online-Anwendungen die Möglichkeit bieten, die Arbeit an einer Datei auf einem anderen Gerät oder Computer fortzusetzen, ist diese Fähigkeit bei Desktop-Anwendungen nicht zu finden. Und dabei ist nicht gemeint, eine Datei mit der gleichen Anwendung auf einem anderen Gerät zu öffnen.

Dies liegt vor allem an der Tatsache, dass diese Eigenschaft bei Desktop-Anwendungen nur bedingt notwendig ist. Da es ja letztendlich ausreicht, wenn man die Möglichkeit hat, wie oben beschrieben, an seinen Dateien auf einem anderen Rechner weiterzuarbeiten. Trotzdem wurde während den vorbereitenden Gesprächen zu dieser Arbeit die Frage gestellt, ob und vor allem wie dieses System umgesetzt werden könnte. Die Aufgabe des Systems ist es, eine Anwendung oder in diesem Fall einen Prozess von einem Gerät auf ein anderes zu transportieren, ohne dass er seinen aktuellen Zustand verliert.

Durch ein solches System wäre es möglich, Ressourcen in einem System zu verschieben, indem Prozesse an die Stellen verlagert werden, an denen sie gebraucht werden. Sollten zum Beispiel besonders viele Daten an einem Ort anfallen, wäre es aus Optimierungssicht durchaus sinnvoller, den Prozess an diesen Ort zu migrieren, als dass man die Daten zeitintensiv zu einem verarbeitenden Prozess schickt.

Motiviert wurde die Arbeit zusätzlich durch die Frage, ob es mit Erlang grundsätzlich möglich ist ein System, wie es oben beschrieben wurde, zu entwickeln und auf welche Einschränkungen man bei der Umsetzung der Aufgaben stoßen würde. Zudem

war es mir ein persönliches Anliegen, an einer Aufgabe im Themenbereich *Verteilte Systeme* arbeiten zu können, wodurch die folgende Arbeit entstanden ist.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist daher die Entwicklung einer Systemstruktur, die es ermöglicht, laufende Prozesse von ihrer aktuellen Position an einen neuen Ort zu migrieren, ohne dass es zu einem Abbruch der Kommunikation zwischen dem Benutzer und dem migrierenden Prozess kommt.

Das System soll bei dieser Aufgabe auf die Erhaltung des aktuellen Zustands des migrierten Prozesses achten. Das bedeutet, dass ein mit bestimmten Werten gestarteter Prozess diese Werte bei einem anstehenden Umzug nicht verliert. Dabei soll es keine Rolle spielen, ob sich diese Werte während der Laufzeit des Prozesses jemals geändert haben oder nicht.

Zur Umsetzung der Aufgabe kommt die Programmiersprache Erlang zum Einsatz. Daher soll diese Arbeit zudem aufzeigen, inwiefern Erlang bei der Implementation der Lösung unterstützend gewirkt hat und in welchen Situationen Probleme entstanden sind.

1.3 Erste Gedanken zur Umsetzung

Ein besonderes Problem bei dieser Aufgabe stellt die Aufrechterhaltung der Kommunikation zu diesem migrierten Prozess dar. Dies liegt vor allem daran, wie Prozesse in Erlang miteinander kommunizieren. Da ein Prozess natürlich die genaue Position seines "Ansprechpartners" benötigt, müssen ihm diese Daten auch mitgeteilt werden. Um dies zu bewerkstelligen, werden in dieser Arbeit zwei mögliche Methoden vorgestellt.

1. Als erste Methode wird ein System mit **Nameservice** implementiert, der immer über die aktuellen "Adressen" aller migrierender Prozesse verfügt. Damit ist es möglich, durch den eindeutigen Namen eines Prozesses auf seine Adresse zurückzuschließen und dadurch die Kommunikation aufrecht zu erhalten. Dieser Name

gehört daher zu einem der Werte, die sich nach einer Migration nicht ändern dürfen.

2. Als Alternative wird ein System entwickelt, bei dem der migrierende Prozess eine **Homestation** zur Verfügung hat (siehe Abb. 1.1). Diese Homestation fungiert dabei wie eine Art Proxy zwischen dem migrierenden Prozess und seinen Kommunikationspartnern, und hat immer die aktuelle Adresse ihres migrierenden Prozesses. Die Homestation hält hier im Gegensatz zum Nameservice nur *eine* Prozess-Adresse. Das bedeutet, dass jede Homestation für nur einen Prozess zuständig ist. Dabei ändert sich die Adresse der Homestation selbst nie. Jede Nachricht an den mobilen Prozess wird bei dieser Methode an die Homestation geschickt, woraufhin diese die Nachricht an den Prozess weiterleitet.

3. Aus Optimierungsgründen wird bei dieser Methode mit einer Homestation eine Erweiterung implementiert, die eine **direkte Kommunikation** zulässt (siehe Abb. 1.1). Dies sorgt dafür, dass es, bis zur nächsten Migration des Empfängers, nicht mehr zu der Umleitung der Nachrichten über die Homestation kommt und die Kommunikation damit beschleunigt wird.

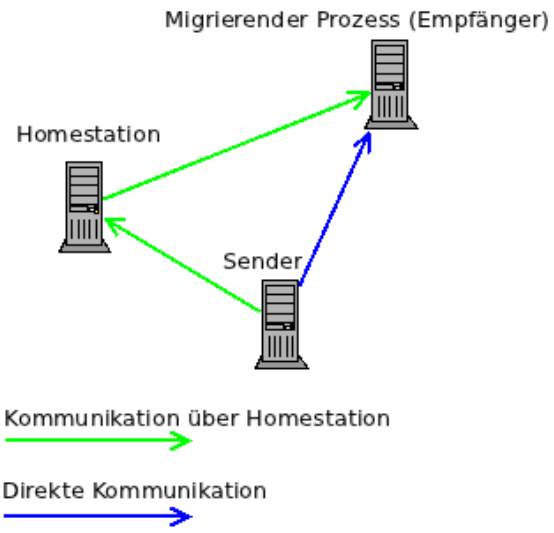

Abbildung 1.1: Kommunikationswege im Homestation-System

1.4 Aufbau der Arbeit

Diese Arbeit soll in Kapitel 2 einen kurzen Einblick in schon existerende Systeme zur Prozessmigration geben, um einen Vergleich zu dem hier entwickelten System ziehen zu können.

Kapitel 3 gibt anschließend eine Einführung in die theoretischen Grundlagen, die dieser Arbeit zugrunde liegen. Dadurch wird dem Leser die Möglichkeit gegeben, Erlang näher kennen zu lernen.

Kapitel 4 beschreibt im Detail, wie das System implementiert und umgesetzt wurde. Dabei wird auch ein Szenario beschrieben, für welches das System eingesetzt werden könnte. Dieses Szenario soll der Veranschaulichung der einzelnen Komponenten dienen. Das Szenario selbst wird dabei nicht umgesetzt.

Anschließend wird in Kapitel 5 ein Einblick in die Erweiterbarkeit des Systems gegeben. Hierbei wird darauf eingegangen weche Maßnahmen getroffen werden können, um das System auszubauen.

Kapitel 6 enthält abschließend das Fazit, indem noch einmal zusammenfassend darauf eingegangen wird, ob sich Erlang als hilfreich erwiesen hat und in welchem Umfang.

2 Bisherige Ansätze der Migration von Prozessen

Systeme zur Prozessmigration exisiteren mit einer Vielzahl von verschiedenen Lösungen, die es sich alle zur Aufgabe machen, einen Prozess von einem Ort zu einem anderen zu migrieren.

Bisherige Ansätze eine solche Migration von Prozessen durchzuführen finden vor allem auf Betriebssystemebene statt [SPJ+91][BD02]. Dazu zählen zum Beispiel speziell für die Migration entwickelte Applikationen, die auf Migrationsbibliotheken oder eine bestimmte Zusammensetzung von Betriebssystemfunktionen aufbauen. Andere Ansätze wiederum setzen einen, für die Migration modifizierten Betriebssystemkernel vorraus. Dieses Vorgehen findet vor allem auf dem Gebiet der parallel laufenden Berechnungen Anwendung. Das Problem hierbei ist, dass die Allgemeinheit der Anwendungen, die heutzutage in Betrieb sind, aus diesen Lösungen keinen Nutzen ziehen kann [BD02].

Prozessmigration mit einem virtuellen Betriebssystem

Ein Ansatz der sich diesem Problem widmet, und der herausgestellt werden sollte, ist die Nutzung eines virtuellen Bestriebssystems [BD02]. Auch dort wird erwähnt, dass es gerade für mobile Anwendung oder Anwendungen in verteilten Systemen von großem Vorteil sein kann, laufende Prozesse zu migrieren. Vorteile die besondere Beachtung erhalten sind unter anderem:

1. Kritische Anwendungen, die auf einer Maschine M_1 laufen, können auf eine andere Maschine M_2 transferiert werden, sollte M_1 abgeschaltet werden müssen.

2. Prozessorintensive Anwendungen könnten die Resourcen von entfernten Maschinen nutzen, um Berechnungen durchzuführen.

Um eine erste Gemeinsamkeit aufzuzeigen, kann erwähnt werden, dass dies natürlich auch Eigenschaften sind, die von dem in dieser Arbeit implementierten System erreicht werden.

Um Anwendungen zu migrieren, nutzen die Autoren in Ihrem Ansatz die Entkopplung des Betriebssystems von seiner physischen Umgebung. Hierbei besteht keine Notwendigkeit diese Anwendungen vorher zu modifizieren. Genutzt werden kann dafür ein gewöhnliches Betriebssystem.

Um im späteren Verlauf der Arbeit einen Bezug zu dieser Implementation herzustellen, soll im Folgenden eine kurze Beschreibung der entwickelten Komponenten dienen.

Der virtualizing System Manager kontrolliert das virtuelle Betriebssystem. Er befindet sich auf einer eigenen Maschine im System, und steuert einzelne Maschinen, indem er mit den *virtualizing Executives* interagiert.

Der virtualizing Executive befindet sich auf jeder Maschine im System und holt sich mit Hilfe des *virtualizing Interceptors* die einzelnen Prozesse der Maschine, auf der er sich befindet. Über ein User-Interface kann dem virtualizing Executive Befehle gegeben werden.

Der virtualizing Interceptor wird in jede gestartete Anwendung eingefügt, um den aktuellen Zustand dieser Anwendung zu überwachen.

Es existiert in diesem System also ein 3-teiliger Aufbau, bei dem der virtualizing System Manager mit den einzelnen virtualizing Executives kommuniziert, von denen wiederum jeder mit den verschiedenen virtualizing Interceptoren auf ihren Maschinen kommuniziert.

In ihrer Arbeit gehen die Autoren auf zwei Migrationsmodelle ein. Die *Minimal State Migration* und die *Full State Migration*. Die Full State Migration ist notwendig, wenn der migrierende Prozess sehr komplex wird. Darunter verstehen die Autoren Prozesse,

die Netzwerk-, Threading- und Dateikomponenten beinhalten. Eine Migration in diesem Umfang ist im Verlauf der hier vorgestellten Arbeit nicht vorgesehen.

Die hier vorgestellte Methode ähnelt der Minimal State Migration. Dabei handelt es sich um Prozesse, die über einen eingeschränkten Umfang, ihren Zustand betreffend, verfügen.

Der Ansatz in der oben erwähnten Arbeit sieht vor, den Zustand eines Prozesses aus *Heap*, *".data"-Area* und *Handles* zusammenzustellen. Die ".data"-Area beinhaltet globale und statische Variablen, die vom Prozess genutzt werden. Ein Handle wird für Zugriffe auf Dateien des Directory-Systems genutzt. Da sich dessen Namensraum ausserhalb des Prozesses befindet, werden vom Betriebssystem Handles zur Verfügung gestellt. Diese sind eindeutige Referenzwerte zu den benötigten Dateien [Doe10].

Mit Hilfe dieser Informationen kann der Prozess auf der neuen Maschine wiederhergestellt werden. Dabei wird er, wie der Prozess in dieser Arbeit auch, neu gestartet und anschließend werden die eben beschriebenen Informationen genutzt, um den Prozess in seinen vorherigen Zustand zu versetzen.

Da sich der hier bearbeitete Ansatz in die Riege der Migrationssysteme einreiht, bei denen das gesamte System für eine Migration entwickelt wird, sind einige Unterschiede zu der obrigen Methode vorhanden.

1. In dem hier entwickelten System kann nicht jeder beliebige Prozess migriert werden. Es müssen dafür spezielle Prozesse implementiert werden, die die benötigten Eigenschaften besitzen, um in einer Migration genutzt werden zu können.

2. Wichtig ist auch, dass das System keine Migration auf Betriebssystemebene vorsieht, sondern sich die Migration innerhalb eines Erlang-Systems abspielt. Daher werden andere Informationen benötigt, um den alten Prozesszustand wieder herzustellen.

3. Diese Informationen können bzw. müssen vom Entwickler für jeden Prozess der migrieren soll einzeln festgelegt werden. Zudem muss nach der Festlegung dieser

Informationen eine Methode entwickelt werden, um diese nach einer Migration an den neuen Prozess zu übergeben.

3 Theoretische Grundlagen

Da die Aufgabe, mit der sich die hier vorgestellte Arbeit befasst, in der Programmiersprache Erlang gelöst wurde, beschäftigt sich dieses Kapitel mit einer kleinen Einführung in die theoretischen Grundlagen von Erlang. Diese sind notwendig, um die Abläufe in Erlang zu verstehen und nachvollziehen zu können. Dabei wird zuerst ein allgemeiner Einblick in die Programmiersprache gegeben, bevor auf die elementaren Eigenschaften eingegangen wird, die bei der Implementation eingesetzt wurden.

3.1 Erlang allgemein

Erlang wurde 1986 im Ericsson Computer Science Laboratory entwickelt. Benannt nach dem Mathematiker Agner Krarup Erlang reiht sich die Sprache damit in die Liste der Programmiersprachen ein, die Namen verstorbener Mathematiker tragen [Dä00].

Die ursprüngliche Idee hinter Erlang war es eine Programmiersprache zu entwickeln mit der es möglich ist Programme zu schreiben, die “ewig” laufen. Diese Eigenschaft war nötig, da Erlang für Telefonanwendungen entworfen wurde, und es gerade in diesem Gebiet wichtig ist, Ausfälle in Form von Fehlern oder Auszeiten aufgrund von Updates zu vermeiden. Der Nutzen dieser Eigenschaften beschränkte sich zum Entwicklungszeitpunkt von Erlang eher auf kleine Problemgebiete [Arm07]. Um ein so großes Projekt wie eine Telefonanwendung mit diesen Fähigkeiten auszustatten, wurde damit begonnen existierende Programmiersprachen auf ihre Tauglichkeit, bezüglich dieser Problemstellung zu untersuchen. Nach einigen Experimenten mit Prolog entwickelte sich daraus der erste Prototyp von Erlang in die Richtung einer funktionalen Programmiersprache [Dä00]. Dieser Prototyp sollte folgende entscheidende Designeigenschaften besitzen:

- Es soll auf einer virtuellen Maschine aufgebaut werden, die sich um Nebenläufigkeit, Speicherverwaltung, etc. kümmert, um die Programmiersprache unabhängig vom Betriebssystem zu machen und Portabilität zu gewährleisten.
- Es soll eine *symbolische* Programmiersprache mit Garbage-Collection, dynamischer Typisierung und Datentypen wie Atomen, Listen und Tupeln werden.
- Es soll *Endrekursion* unterstützt werden, damit alle Schleifen, selbst endlose wie sie bei Treibern vorkommen, durch Rekursion bearbeitet werden können.
- Es soll *asynchrones Versenden von Nachrichten* unterstützt werden, sowie ein selektiver *receive* Ausdruck.
- Es soll eine grundsätzliche Fehlerbehandlung durch z.B. *trap exits* ermöglicht werden. Dies erlaubt einen aggressiven Programmierstil.

[Dä00, pg 14]

Diese Eigenschaften zeigen schon, dass Erlang in eine Richtung gesteuert wurde, die massive Nebenläufigkeit und Verteilung der Systeme zulässt. Der nachfolgende Abschnitt ist angelehnt an Erlang Kompakt: Einführung in die Programmierung mit Erlang [Dor09, pg 3-5]. Anders als bei den meisten Programmiersprachen löst Erlang das Problem der großen Nebenläufigkeit nicht durch Multithreading. Der Grund hierfür ist, dass diese Methode aufgrund des hohen Ressourcenverbrauchs sowie Performance-Overheads schnell an seine Grenzen stoßen würde und damit praktikable Systeme nur schlecht bis gar nicht umgesetzt werden können.

Erlang baut daher auf leichtgewichtige, voneinenader unabhängige Prozesse auf, die zur Ressourcenschonung und Effizienz in Erlangs eigener Laufzeitumgebung (Erlang-VM) implementiert sind. Damit kann schon eine weitere wichtige Eigenschaft Erlangs, nämlich, dass Prozesse nicht auf den gleichen Daten arbeiten, beschrieben werden. Erlang implementiert daher das Konzept der *Message Passing Concurrency*, in welchem Prozesse einzig durch das Versenden und Empfangen von Nachrichten miteinander kommunizieren. Dabei spielt es keine Rolle ob die Prozesse lokal oder global laufen.

Abbildung 3.1 zeigt skizzenhaft wie einfach die Kommunikation in Erlang umgesetzt ist und soll verdeutlichen, dass Erlang nichts weiter als einen Sender benötigt, der eine Nachricht an den Empfänger schickt.

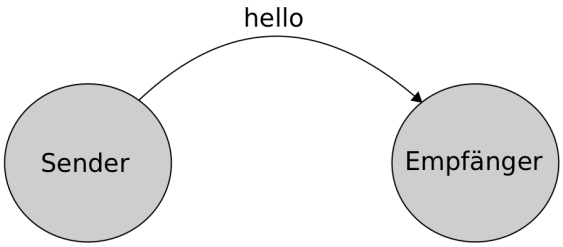

Abbildung 3.1: Einfache Kommunikation zwischen zwei Prozessen

Der dazugehörige Code ist hier in Listing 3.1 zu sehen. Darin wird auch deutlich, wie eine Funktion in Erlang auf Nachrichten reagiert und diese gegebenenfalls gezielt herausfiltern kann.

```
1  sender() ->                                  % Sende-Funktion
2      empfaenger ! hello                       % '!' ist der Sendebefehl von Erlang
3
4  empfaenger() ->                              % Empfaengerfunktion
5      receive                                  % Receive-Block wartet auf Nachrichten
6          hello ->                             % Bei Empfang von 'hello' wird
7              io:format("Received hello~n")    % dieser Befehl ausgefuehrt
8      end.
```

Listing 3.1: Message Sending

Neben der Tatsache, dass sich so einfach nebenläufige, verteilte Systeme programmieren lassen, ermöglicht Erlang damit auch den Aufbau einer *Überwachungsstruktur*. Indem Prozesse untereinander verbunden werden, können sie als *Wächter* auf Fehler der korrespondierenden Prozesse reagieren und diverse Maßnahmen einleiten. Dies kann von einem einfachen Neustart des fehlerhaften Prozesses, bis hin zu komplexen Strategien

reichen, um die Funktionalität des Systems aufrecht zu erhalten.

Durch die eigene virtuelle Maschine ist neben der Plattformunabhängigkeit auch eine Multicore-Parallelisierbarkeit der Prozesse möglich, was Erlang gerade in der Zeit, indem Rechner nicht mehr nur mit einem Kern arbeiten, zu einer echten Alternative werden lässt. Dies wird dadurch erreicht, dass die VM pro Rechnerkern einen eigenen Prozessscheduler startet auf dem die Erlang-Prozesse verteilt werden können.

3.1.1 Verzeichnisstruktur und die Erlang-Shell

Im weiteren Verlauf der Arbeit werden immer wieder Listings, wie zum Beispiel 3.2 zu sehen sein. Bei diesen Listings handelt es sich um Ausschnitte aus der Erlang-Shell. Während dieser Fall nur die Zuweisung eines Wertes an eine Variable zeigt, werden die folgenden Listings dieser Art zur Veranschaulichung der beschriebenen Abläufe genutzt.

Die Shell ist ein Emulator, der einem die Möglichkeit gibt Erlang-Programme auszuführen oder Code direkt einzugeben und zu testen. In dieser Arbeit wird die Shell genutzt, um sowohl das Migrationsprogramm zu testen, als auch Abläufe des Systems zu veranschaulichen. Zu diesem Zweck werden kurze Codeabschnitte gelistet.

```
Erlang R15B01 (erts-5.9.1) [source] [smp:4:4] [async-threads:0] [hipe]
    [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
1> X = 1.
1
2>
```

Listing 3.2: Die Erlang-Shell

An dieser Stelle soll noch ein kurzer Einblick in die verwendete Verzeichnisstruktur gegeben werden.

```
migrationProject/ ..... Hauptverzeichnis
├── src/ ..... Enthält die Source-Dateien des Projektes (.erl)
├── include/ ..... Enthält die Header-Dateien des Projektes (.hrl)
├── ebin/ ..... Enthält die kompilierten Dateien (.beam)
├── priv/ ..... Enthält ausführbare Dateien, die genutzt werden
└── test/ ..... Enthält die Test-Dateien eines Projektes
```

Wie zu sehen ist enthält das Projekt fünf Unterordner. In **src/** befinden sich die Erlang-Source-Dateien. Das **include/**-Verzeichnis beinhaltet sämtliche Header-Dateien, die das System benötigt. In **ebin/** werden die kompilierten Dateien gesammelt, die für den Start des Systems notwendig sind. Auf den genauen Ablauf des Kompiliervorgangs wird im weiteren Verlauf der Arbeit eingegangen. Die letzten beiden Verzeichnisse sind **priv/**, welches ausführbare Dateien enthält, die vom entwickelten System genutzt werden und **test/**, das alle Dateien mit Testroutinen enthält.

Sowohl **priv/**, als auch **test/** werden in dieser Arbeit nicht benötigt. In **include/** befinden sich die Header-Dateien. Diese beinhalten die Definitionen der Records, die genutzt werden, um den Zustand der einzelnen Prozesse zu beschreiben. *Records* sind Datenstrukturen, die einem Struct in C ähneln [AB13i]. Diese Record-Definitionen werden für gewöhnlich in Header-Dateien ausgelagert. **Priv/** wird nicht benutzt, da keine zusätzlichen ausführbaren Dateien notwendig waren, um das System zu entwickeln.

3.1.2 Das Modul

Als Modul wird in Erlang die Datei bezeichnet, in die der Code geschrieben wird. Sie sind meist so aufgebaut, dass sie bestimmte Funktionalitäten gruppieren oder eine Aufgabe, wie etwa die eines Namensdienstes zusammenfassen. So umfasst die Erlang-Bibliothek zum Beispiel die Module **lists** oder **erlang**, um nur 2 zu nennen. Lists ist dabei ein gutes Beispiel dafür, wie ein Modul genutzt wird um Funktionalitäten zu gruppieren. Das Modul enthält sämtliche Funktionen, die für die Manipulation von Listen nötig sind.

An dieser Stelle soll allerdings nur der Aufbau eines Moduls beschrieben werden. Das Beispiel in Listing 3.3 soll dabei zur Veranschaulichung dienen. Es addiert dafür zwei Zahlen, die als Argumente an die Funktion **start()** übergeben werden.

```
1 -module(easyprocess).
2 -export([start/2]).
3
4  start(Number1, Number2) ->
5      Result = add(Number1, Number2),
6      io:format("Result: ~p~n", [Result]).
7
8  add(N1, N2) ->
9      N1 + N2.
```

Listing 3.3: Modulaufbau

Im Kopf des Moduls, genauer in Zeile 1, befindet sich **-module(easyprocess)**. Dieser Eintrag ist das erste Attribut des Moduls und gibt an wie das Modul heißt. Dieses Attribut steht *immer* an erster Stelle und ist ein Atom, das mit dem Namen der Moduldatei übereinstimmen muss. Weiterhin gibt diese Zeile an wie das Modul von der "Außenwelt" aufgerufen wird. Dies nennt man einen *externen Aufruf*. Das passiert in der Form *Modul:Funktion(Argumente)*, und würde hier dementsprechend wie folgt aussehen:

```
easyprocess:start(10,20)
```

Zeile 2 gibt an welche Funktionen des Moduls *exportiert*. Darunter versteht man in Erlang, welche Funktionen für externe Aufrufe zugelassen sind. Hier werden die zu exportierenden Funktionen in einer Liste aufgezählt und haben dabei die Form *Name/Anzahl der Argumente*. Im Beispiellisting wäre das die Funktion **start()** mit zwei Argumenten. Die Funktion **add()** dagegen kann nicht von außen aufgerufen werden, und daher würde bei der Ausführung der Eingabe **easyprocess:add(10,20)** nur folgende Fehlerausgabe produziet werden:

```
** exception error: undefined function easyprocess:add/2
```

Diese Funktion **add()** kann also nur innerhalb des Moduls benutzt werden, wie in Zeile 5 des Listings 3.3 zu sehen ist. Dies nennt man einen *lokalen Aufruf*.

Der Modulkopf kann noch weitere Attribute aufzählen, dazu zählen unter anderem *-behaviour()*. Erlang benutzt Behaviours, um den Code in zwei Teile, einen generischen und einen spezifischen zu teilen. Dadurch müssen vom Entwickler nur noch die Callback-Funktionen implementiert werden [AB13e]. Im weiteren Verlauf der Arbeit werden diese Behaviours genutzt, um Server und Supervisor zu implementieren.

Des Weiteren existieren noch *-import()*, um weitere Module und Funktionen einzubinden oder *-compile()*, um dem Compiler mögliche Optionen mitzuteilen. Da diese aber für den weiteren Verlauf der Arbeit nicht von Bedeutung sind, wurde nicht weiter auf sie eingegangen. Eine gesamte Liste der Modulattribute ist in der Dokumentation zu finden [AB13h].

3.1.3 Benutzung eines Moduls

Um ein Modul nutzen zu können, muss es erst kompiliert werden. Dabei entsteht ein sogenanntes *Beam-File*. Beim Start einer Erlang-Shell sucht diese in dem Verzeichnis, von dem aus sie gestartet wurde, nach sämtlichen Beam-Files. Diese können nun in der Shell genutzt werden. Module, die Erlang mitliefert, müssen nicht als Beam-Files in besagtem Verzeichnis vorliegen.

Um ein Modul zu einem Beam-File zu kompilieren, gibt es in Erlang mehrere Möglichkeiten. Die einfachste ist die Nutzung der Erlang-Shell. Innerhalb dieser Shell kann jede Erlang-Datei durch den Befehl *c(<Dateiname>)* kompiliert werden. Diese Methode setzt voraus, dass sich die Datei im gleichen Verzeichnis befindet in dem auch die Erlang-Shell gestartet wurde. Andernfalls muss der Pfad zu gewünschter Datei statt nur dem Dateinamen angegeben werden.

Die in dieser Arbeit verwendete Alternative zu oben beschriebenem Ablauf, ist die

Nutzung von *Rebar*[1]. Rebar ist ein Werkzeug, dass speziell für Erlang-Projekte entwickelt wurde. Dabei handelt es sich um ein Script, das in erster Linie zum Erstellen, sowie Kompilieren und der Veröffentlichung von Projekten dient.

Die Benutzung des Scriptes ist dabei sehr einfach gestaltet. Der folgende Abschnitt beschreibt, wie mithilfe von Rebar Dateien kompiliert werden, um sie anschließend in der Erlang-Shell nutzen zu können.

```
$> ./rebar compile
==> gentests (compile)
Compiled src/homestation_server.erl
Compiled src/simpleprocess.erl
Compiled src/migration_server.erl
$>
```

Listing 3.4: Rebar - Kompilierung

Wie das Listing 3.4 zeigt, durchsucht Rebar das src/-Verzeichnis und kompiliert alle Erlang-Dateien. Dabei beschränkt sich das Script auf Dateien, die sich seit der letzten Kompilierung geändert haben. Sollte zu diesem Zeitpunkt noch kein ebin/-Verzeichnis existieren, wird dieses von Rebar erzeugt. Die eben erzeugten Beam-Dateien sind daraufhin in diesem Verzeichnis zu finden.

Beim Start der Erlang-Shell aus dem Hauptverzeichnis mit dem Zusatz *-pa ebin/* sucht die Shell im ebin/-Verzeichnis nach Beam-Dateien, um die Module dem Nutzer zur Verfügung zu stellen. Somit muss nicht immer jede Datei einzeln kompiliert werden und man kann die Erlang-Shell vom Hauptverzeichnis aus starten, ohne nach den einzelnen Dateien zu suchen [Heb10].

Dieser Abschnitt beschreibt nur die Kompilierungs-Funktion von Rebar. Ein umfangreiches Wiki, dass von der Git-Community rund um das Rebar-Projekt gepflegt wird, beschreibt die gesamte Funktionalität [Reb13]. Zudem wird im weiteren Verlauf genauer

[1] https://github.com/basho/rebar

darauf eingegangen, inwieweit Rebar bei der Erstellung des Migrations-Projektes genutzt wurde.

3.1.4 Typisierung

Wie in den vorherigen Listings schon zu erkennen war, ist Erlang eine *dynamisch typisierte* Sprache. Das liegt vor allem daran, wie Erlang in der Vergangenheit gewachsen ist. Es wurden zwar einige Versuche gestartet, ein Typsystem in Erlang zu implementieren, allerdings setzte sich keine dieser Ideen durch [Arm07].

Die Änderung eines Typs in einen anderen geschieht in Erlang durch Funktionen der Form: **erlang:<start_typ>_to_<ziel_typ>()**. Ein Beispiel dafür wäre die Änderung einer List in einen Integer-Wert, und sähe wie folgt aus: **erlang:list_to_integer("13")**.

3.1.5 Hot Code Swapping

Dieses Kapitel beschreibt eine Besonderheit Erlangs, das *Hot Code Swapping*. Diese Funktionalität findet zwar keinen Gebrauch im weiteren Verlauf der Arbeit, sollte aber durch seine Ungewöhnlichkeit erwähnt werden, da es eine der Fähigkeiten ist, die dafür verantwortlich ist, dass Erlang in vielen Systemen mit langer Laufzeit eingesetzt wird.

Diese Eigenschaft ist ausschlaggebend dafür, Ausfallzeiten eines Systems während Updates vermeiden zu können. Dabei erlaubt es Erlang eine neue Version des Codes zu laden und im laufenden Betrieb des Systems auszuführen, ohne dass das System angehalten werden muss.

Im Codebeispiel 3.5 wird gezeigt wie in Erlang ein Modul zur Laufzeit geändert werden kann, ohne dass es zu Ausfällen kommt. Dafür wurde ein Modul geschrieben, das nur aus einer Schleife besteht, die auf Nachrichten wartet. Nach dem Start dieses Moduls läuft der Prozess an und wartet auf die erste eintreffende Nachricht. Die Funktion ***loop()*** schreibt bei Erhalt der Nachricht “test” die Zeile “Alte Version” auf die Konsole. Wenn jetzt das Modul ohne Rücksichtnahme auf den bereits laufenden Prozess bearbeitet wird und man den Teil des Codes der für die Ausgabe zuständig ist (Zeile 4) durch

```
1  loop() ->
2      receive
3          test ->
4              io:format("Alte Version~n"), % hier wird der Code geaendert
5              loop();
6
7          update ->
8              code:purge(?MODULE),
9              compile:file(?MODULE),
10             code:load_file(?MODULE),
11             ?MODULE:loop()
12     end.
```

Listing 3.5: Hot Code Swapping

io:format("Neue Version n") ersetzt, dann wird der Prozess nach Erhalt der Nachricht "update" den alten Code verwerfen. Die Änderung findet dementsprechend lokal statt und die neue Code-Datei muss die schon vorhandene ersetzen. Das dadurch veränderte Modul wird durch die Funktion ***compile:file()*** kompiliert und der dabei erzeugte Code in der darauffolgenden Zeile neu geladen. Anschließend wird die Funktion ***?MODULE:loop()*** aufgerufen. Von jetzt an läuft der Prozess mit dem neuen Code und wird dementsprechend auch bei Erhalt der Nachricht "test" die Zeile "Neue Version" ausgeben.

Rot zeigt an, an welcher Stelle der alte Code aufgerufen wird, während Grün den Aufruf des neuen Codes hervorhebt. Dabei ist zu erkennen, dass die Funktion ***loop()*** mit *?MODULE* vorweg aufgerufen werden muss damit der neue Code benutzt wird. Verantwortlich für diese Funktionalität ist der *Code Server* von Erlang. Dieser kann bis zu zwei Versionen eines Moduls in seinem Speicher halten, welche beide gleichzeitig laufen können. Dazu sollte man das Konzept der *lokalen* und *externen* Aufrufe nochmals erwähnt werden. Ein lokaler Aufruf kann Funktionen aufrufen, die nicht im Modulkopf exportiert wurden, in diesem Fall also ***loop()***. Ein externer Aufruf dagegen kann nur mit exportierten Funktionen arbeiten und sieht wie ***?MODULE:loop()*** aus. Sollten zwei Versionen eines Moduls in der VM geladen sein, dann arbeiten alle lokalen Aufrufe auf der aktuell *laufenden* Version, während externe Aufrufe immer auf der neuesten Version eines Moduls arbeiten, das auf dem Code Server zur Verfügung steht. Lokale Aufrufe,

die auf einen externen folgen, arbeiten auf der neuen Version weiter [Heb10].

Das folgende Listing 3.6 zeigt den soeben beschriebenen Ablauf nochmal beispielhaft in der Erlang-Shell. Die Änderung am Code werden zwischen Zeile 3 und 4 durchgeführt.

```
Erlang R15B01 (erts-5.9.1) [source] [smp:4:4] [async-threads:0] [hipe]
    [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
1> c(codeswapping).
{ok,codeswapping}
2> Empfaenger = spawn(codeswapping, loop, []).
<0.39.0>
3> Empfaenger ! test.
Alte Version
test
4> Empfaenger ! update.
update
5> Empfaenger ! test.
Neue Version
test
6>
```

Listing 3.6: Code Swapping Shell

Der hier gezeigte Ausschnitt zeigt die im Verlauf der Arbeit genutzte Version von Erlang. Dabei handelt es sich um die 1998 veröffentlichte OTP (Open Telecom Platform). OTP ist eine Open-Source Distribution von Erlang und beinhaltet neben Compiler und Interpreter auch noch alle notwendigen Bibliotheken und Protokolle, um selbst Erlang-Systeme entwickeln zu können.

Diese Zusammenfassung soll einen ersten Einblick in einige Fähigkeiten der Programmiersprache Erlang geben und ihre Stärken aufzeigen. Die folgenden Kapitel befassen sich mit den Grundlagen, die für das weitere Verständnis dieser Arbeit benötigt werden.

3.2 Kommunikation zwischen Prozessen

Wie schon im vorherigen Abschnitt beschrieben, kommunizieren Prozesse in Erlang durch den Austausch von Nachrichten. Um dies zu erreichen, ist es notwendig die genaue *Adresse* des Empfängers zu kennen. Der Einfachheit halber wurde im oben gezeigten Listing 3.6 noch nicht näher darauf eingegangen, wie ein Prozess genau angesprochen wird. Daher wurde der Empfänger auch einfach nur *Empfaenger* genannt. Im Folgenden wird vorgestellt welche Möglichkeiten Erlang bietet, Prozesse zu adressieren.

3.2.1 Die Prozess-ID

Sobald ein Prozess gestartet wird, läuft dieser auf einer Node. Eine Node ist in diesem Fall eine gestartete Erlang-Shell, auf der beliebig viele Prozesse laufen können. Diese Node ist Teil der *Adresse* eines Prozesses und somit vergleichbar mit der Stadt in der man wohnt. So wie verschiedene Städte gleiche Straßennamen haben können, so können auch Prozesse die auf unterschiedlichen Nodes laufen in ihrer *lokalen* Umgebung durchaus die gleiche ID haben. Von daher ist die Node auf der ein Prozess gestartet wurde, ein wesentlicher Bestandteil zur Identifikation eines Prozesses.

Die erste Möglichkeit einen Prozess anzusprechen, die hier vorgestellt werden soll, ist mit Hilfe der Prozess-ID (PID). Hierbei wird unterschieden zwischen lokalen und globalen PIDs. Diese PID ist für jeden Prozess in einem Erlang-System eindeutig[2] und hat folgenden Aufbau: <A.B.C>, also zum Beispiel <0.64.0>.

- "A" ist der Node-Anteil der PID. Hier wird die Relevanz der Node für die Identifikation eines Prozesses noch einmal deutlich. Dabei zeigt eine **0** an dieser Stelle, dass es sich um die lokale PID handelt.
- "B" sind die ersten 15 Bits der Prozessnummer.
- "C" sind die Bits 16-18 der Prozessnummer.

Sobald eine lokale PID von einer Node zu einer anderen geschickt wird, wird sie aus oben genannten Gründen den "A"-Teil verändern und damit deutlich machen auf welcher

[2] Wie oben beschrieben, muss dies nicht für die lokalen PIDs von Prozessen auf unterschiedlichen Nodes gelten.

Node der Prozess im Moment läuft. Daher wird eine PID, die zuerst <0.64.0> lautete, sobald sie auf einer fremden Node ankommt, zum Beispiel <6817.64.0> heißen [arc09][3].

Zur Veranschaulichung soll das folgende Beispiel helfen, und die Vorgänge noch einmal sichtbar machen.

```
(eins@thinkpad)1> EinsPid = simpleprocess:start(10,20).
<0.45.0>
(eins@thinkpad)2> simpleprocess:send({self(), show}, zwei@thinkpad).
My local PID: <0.38.0> looked like: "<9338.38.0>" as global PID when
    received by the corresponding Process
ok
(eins@thinkpad)3>
```

Listing 3.7: Globale/Lokale PIDs

In Listing 3.7 und dem dazugehörigen Bild (siehe Abb. 3.2) wird auf 2 Nodes jeweils ein Prozess des Moduls *simpleprocess* gestartet, der Nachrichten der Form **{Pid, show}** erwartet und darauf mit der von uns erhaltenen PID antwortet. Bei dem oben gezeigten Auszug wurde sich auf die Node beschränkt auf der der Sende- und Empfangsvorgang betrachtet werden kann (hier: *eins@thinkpad*). Nebenläufig läuft dementsprechend noch einmal der gleiche Prozess auf einer Node mit Namen *zwei@thinkpad*.

In Befehl 1 wird der Prozess gestartet und die PID des Prozesses in der Variable *EinsPid* gespeichert. Daraufhin wird in Befehl 2 dem Prozess auf der Node *zwei@thinkpad* die Nachricht **{self(),show}** geschickt. **self()** ist eine Funktion des Moduls *erlang* [AB13a], welche die PID des aufrufenden Prozesses zurückgibt, dies ist in unserem Fall also die PID der Erlang-Shell. Damit wird dem Empfänger mitgeteilt, an wen er die Antwort schicken soll. Die zweite Zeile des Befehls 2 zeigt die Ausgabe, die nach Erhalt der Antwort angezeigt wird. Diese Ausgabe beinhaltet sowohl unsere lokale, als auch die entsprechende globale PID der Erlang-Shell und zeigt auch, dass sich erwartungsgemäß nur der *Node-Teil* der PID geändert hat.

[3] Bei der Quelle handelt es sich um einen Eintrag auf www.stackoverflow.com. Es ist daher nicht vollständig sichergestellt, dass die daraus zitierten Stellen der Wahrheit entsprechen. Sollte dies nicht der Fall sein, wäre die Beschreibung der PID ungültig. Dies hätte allerdings keine weiteren Auswirkungen auf den Verlauf der Arbeit

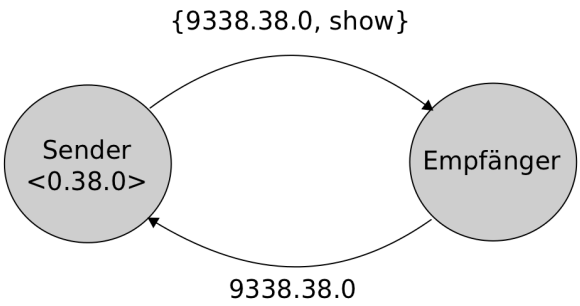

Abbildung 3.2: Abbildung zu Listing 3.7

3.2.2 Namensregistrierung

Eine weitere Möglichkeit mit einem Prozess zu kommunizieren ist es, seine PID unter einem Namen zu registrieren. Dadurch ist es nicht länger notwendig mit der doch recht unhandlichen PID arbeiten zu müssen, sondern man kann jederzeit auf den Namen zurückgreifen.

Dazu wird auf lokaler Ebene die Funktion **register()** aus dem Modul *erlang* [AB13a], oder auf globaler Ebene die Funktion **register_name()** aus dem Modul *global* [AB13b] benutzt. Auf die Unterschiede wird im Verlauf dieses Abschnittes noch näher eingegangen. Gemeinsam haben beide Funktionen aber, dass sie als Eingabe an erster Stelle einen eindeutigen Namen und an zweiter Stelle die PID des Prozesses erhalten, die mit dem Namen von nun an identfiziert werden soll. Sollte der Name schon benutzt werden, oder die PID nicht exisiteren, werden beide Funktionen mit einer Fehlermeldung reagieren, wodurch das doppelte Belegen von Namen schon von Erlang selbst abgefangen wird.

Der Unterschied zwischen beiden Funktionen besteht darin, dass **register()** lokal einer PID einen Namen zuweist, dieser aber von entfernten Nodes nur genutzt werden kann, wenn auch der Name der Node, auf dem der Prozess registriert wurde, bekannt ist (siehe: 3.8). Dagegen macht die Funktion **register_name()** den Namen systemweit bekannt, indem sie alle Nodes des Netzwerks über einen neuen globalen Namen informiert.

```
(eins@thinkpad)1> F = fun() -> receive hallo -> io:format("Received
    message~n") end end.
#Fun<erl_eval.20.82930912>
(eins@thinkpad)2> Pid = spawn(F).
<0.41.0>
(eins@thinkpad)3> register(myfun, Pid).
yes
(eins@thinkpad)4> myfun ! hallo.
Received message
<0.41.0>
(eins@thinkpad)5>
```

Listing 3.8: Lokale Registrierung

In Listing 3.8 wird in den Zeilen 1 und 2 zuerst eine Funktion definiert und dann gespawned. Diese Funktion besteht einzig aus einem Receive-Block, der die Nachricht *hallo* herausfiltert und daraufhin die Nachricht "Received message" auf die Konsole schreibt. Die PID wird dann in Zeile 3 unter dem Namen **myfun** registriert. Zeile 4 zeigt wie dieser Name daraufhin genutzt werden kann, um der Funktion eine Nachricht zuzusenden.

Das nächste Beispiel zeigt den Vorgang einer *globalen* Registrierung. Dabei ist zu erwähnen, dass Nodes sich untereinander erst "bekannt machen" müssen damit sie auch über globale Namen, die auf einer Node erzeugt werden, in Kenntnis gesetzt werden. Dies passiert mit der Funktion **ping()** des Moduls *net_adm* [AB13c], die aber zur Vereinfachung des Beispiels 3.9 in diesem Abschnitt nicht näher betrachtet wird. Das Beispiel zeigt daher nur wie mit der Funktion **send()**, die ebenfalls Bestandteil des Moduls *global* ist, an einen global registrierten Namen Nachrichten geschickt werden können.

Listing 3.9 zeigt analog zum lokalen Beispiel, wie der Name genutzt werden kann, um einem Prozess eine Nachricht zukommen zu lassen. Der größte Unterschied hierbei ist die schon erwähnte Funktion **send()**, die auf globaler Ebene genutzt werden muss.

```
(eins@thinkpad)1> F = fun() -> receive hallo -> io:format("Received
    message~n") end end.
#Fun<erl_eval.20.82930912>
(eins@thinkpad)2> Pid = spawn(F).
<0.41.0>
(eins@thinkpad)3> gobal:register_name(myfun, Pid).
yes
(eins@thinkpad)4> global:send(myfun, hallo).
Received message
<0.41.0>
(eins@thinkpad)5>
```

Listing 3.9: Globale Registrierung

3.3 Aufbau eines Erlangnetzes

Wie im letzten Kapitel erwähnt ist es notwendig, Nodes miteinander bekannt zu machen, um die Vorzüge der globalen Namensregistrierung nutzen zu können. Dies geschieht in Erlang durch ein initiales Pingen. Dabei steht die oben genannte Funktion **ping()** zur Verfügung. Der Ablauf ist wie folgt:

Node **eins** nutzt die Funktion um einen Ping an Node **zwei** zu schicken. Die erfolgreiche Durchführung des Pings, wird durch die Ausgabe eines “pong” auf der Konsole bestätigt. Daraufhin sind diese beiden Nodes verbunden. Sollte die versuchte Verbindung fehlschlagen wird ein “pang” auf der Konsole ausgegeben. Die Funktion **names()**, die ebenfalls aus dem Modul *net_adm* [AB13c] stammt, zeigt danach alle Nodes an, die dem Netz angehören. Dabei zählt es zu den Eigenschaften Erlangs, dass mit einem erfolgreichen Ping auch sämtliche, schon bekannte Nodes ausgetauscht werden.
Sobald nun die Nodes voneinander wissen können die vorgestellten Funktionen des Moduls *global* genutzt werden, um untereinander zu kommunizieren.

Sicherheit

Dieser Abschnitt soll nicht zu ausführlich auf die Sicherheitsaspekte in Erlang eingehen, da diese für die vorliegende Arbeit keine allzu große Rolle spielen. Ein grundlegendes Problem sollte im Weiteren dennoch erwähnt werden.

Denn der Aufbau eines Netzes auf die eben genannte Art und Weise birgt natürlich das Risiko, dass sich mit einem einfachen *Ping* auch fremde Nodes in ein Netz eingliedern können. Diesem Problem begegnet Erlang auf einem sehr einfachen Weg. Und zwar, indem gefordert wird, dass man beim Start einer Erlang-Node im Internet, den Parameter **-setcookie <cookie>** anfügt. Wenn nun alle Nodes, die vom Entwickler aus auch Teil des Netzes werden sollen, mit dem gleichen *Cookie* in diesem Parameter gestartet werden, ist zumindest auf einfacher Ebene sichergestellt, dass fremde Nodes sich nicht per Ping bei dem Netz melden können. Dies liegt daran, dass Erlang das Pingen einer Node, die mit einem Cookie gestartet wurde nicht zulässt, solange man seine Node nicht mit dem gleichen Cookie gestartet hat. Die angepingte Node in unserem Netz reagiert auf einen solchen ungewollten Ping mit folgender Ausgabe, um auf einen versuchten Zugriff hinzuweisen.

```
=ERROR REPORT==== 10-Jun-2013::16:25:11 ===
** Connection attempt from disallowed node zwei@thinkpad **
```

Die Sender-Node dagegen bekommt nur das schon erwähnte *pang* angezeigt, und sonst keine weiteren Hinweise darauf, dass der Ping aufgrund eines gesetzten Cookies fehlgeschlagen ist.

3.4 Fazit

Die eben aufgezeigten theoretischen Grundlagen decken natürlich nicht im Ansatz alle Eigenschaften der Programmiersprache Erlang ab, sollen aber einen Einblick in die Sprache geben und beim weiteren Verständnis der Arbeit helfen. Weiterhin sei erwähnt, dass nicht alle hier beschriebenen Fähigkeiten Erlangs auch von signifkanter Relevanz für die Umsetzung der Arbeit sind. Diese wurden allerdings aufgrund ihrer Besonderheit aufgezählt und zeigen wie sich Erlang von anderen Programmiersprachen abhebt oder können zum weiteren Ausbau des in dieser Arbeit umgesetzten Systems dienen.

Besonderes Augenmerk sollte daher auf die Abschnitte *Erlang allgemein* 3.1, und *Kommunikation zwischen Prozessen* 3.2 gelegt werden, da diese eine große Rolle im weiteren Verlauf der Dokumentation spielen werden.

4 Umsetzung

Dieses Kapitel beschäftigt sich mit der Umsetzung des Migrationssystems und beschreibt die einzelnen Komponenten, sowie den Ablauf der Entwicklung.

Im Laufe der folgenden Abschnitte werden wichtige Codeabschnitte zusätzlich hervorgehoben, um eine genaue Erklärung der Abläufe zuzulassen.

Der Aufbau des Kapitels ist wie folgt. Zuerst wird ein Szenario gezeigt, das der Veranschaulichung der weiteren Umsetzung dienen soll. Im darauf folgenden Abschnitt werden die vorbereitenden Aufgaben beschrieben, die nötig waren, um mit der Entwicklung der beiden Systeme zu beginnen. Anschließend wird auf die Entwicklung der beiden Systeme (Migration mit Nameservice und Migration mit Homestation) eingegangen. Dabei wird die Entstehung der einzelnen Komponenten beschrieben und deren Funktionalität erklärt.

4.1 Szenario

Wie schon erwähnt gibt es unterschiedliche Anwendungsgebiete für ein Migrationssystem, wie es in dieser Arbeit vorgestellt wird. Zum Beispiel könnte es eingesetzt werden, um in einem schon exisitierenden System die Last dynamisch verteilen zu können. Vergleichbar mit der in Kapitel 2 erwähnten Idee, könnten Prozesse auf stark ausgelasteten Servern, auf weniger ausgelastete ausgelagert wird.

In diesem Fall diente bei der Umsetzung das Spiel *Scotland Yard* [Spi00] als Szenario. Dieses Szenario macht es möglich, die Grundfunktionalität der einzelnen Komponenten des Systems zu veranschaulichen und somit deren Erklärung zu vereinfachen.

Bei Scotland Yard handelt es sich um ein Brettspiel, bei dem bis zu 5 *Agenten* versuchen, einen *flüchtigen Verbrecher* zu fangen. Das Spielfeld besteht im Original aus einer Karte der Stadt London. Die Spieler können sich dabei mithilfe diverser Verkehrsmittel von einem Ort zum nächsten bewegen. Dabei sind sie beschränkt in der Anzahl und der Art der zur Verfügung gestellten *Fahrtickets*. Das Spiel ist zuende, sobald ein Agent auf dem Feld des Verbrechers landet oder sobald die Agenten alle Tickets aufgebraucht haben, und der Verbrecher zu diesem Zeitpunkt noch nicht gefangen wurde.

In diesem Szenario würde der migrierende Prozess zur Implemenation der Agenten und des flüchtigen Verbrechers genutzt werden. Damit wäre es den "Spielfiguren" möglich, sich auf dem Spielfeld zu bewegen. Das Spielfeld wäre ein Netz aus vielen Nodes, die den "spielenden" Prozessen als Stationen zur Verfügung stehen.

Bei einem tatsächlichen Einsatz des in dieser Arbeit beschriebenen Systems, im Szenario *Scotland Yard*, müsste natürlich noch sichergestellt werden, dass es nicht möglich ist, von jeder Node aus jede andere zu erreichen. Zudem wäre es notwendig, dass der Verbrecherprozess, wie im eigentlichen Spielverlauf vorgesehen, eine Liste der schon besuchten Nodes hält, und diese zu festgelegten Zeitpunkten allen Agenten offen legt. Im Spiel dient diese Offenlegung als Hilfestellung für die Agenten, um ihnen einen Eindruck der Bewegung des Verbrechers zu geben.

Das folgende Kapiel behandelt jedoch nicht die Umsetzung eines Scotland-Yard Spiels mit migrierenden Prozessen. Vielmehr soll dieses Szenario Anhaltspunkte liefern, um sich eine bessere Vorstellung von der Funktion der einzelnen Komponenten machen zu können. Dabei wird immer wieder auf dieses Szenario zurückgegriffen und gegebenenfalls eine Verbindung zu einem Spielelement hergestellt, wenn es der Erklärung oder der Verdeutlichung der Funktion einer Komponente dient.

4.2 Vorbereitende Aufgaben

Ein besonderes Anliegen war es, dass das Modul des migrierenden Prozesses nicht schon auf der Ziel-Node verfügbar sein muss, um gestartet zu werden. Dabei wird während einer Migration, die aktuelle Version des Moduls an die Ziel-Node geschickt. Um das zu erreichen, benötigt es einen Sender-Prozess, der Bestandteil des Systems auf der Quell-Node ist und einen Empfänger-Prozess, der auf der Ziel-Node gestartet werden muss.

Bevor also mit der Entwicklung des gesamten Systems begonnen wurde, wurde sichergestellt, dass das Senden des Beam-Files den Vorgaben entsprechend funktioniert. Dazu gehören sowohl das Module *filesender.erl*, dessen Funktion **send_file** auf der Quell-Node gestartet wird, als auch das Modul *tcp_echo.erl*, dessen Funktion **start** korrespondierend dazu auf der Ziel-Node läuft. Dabei wartet der auf der Ziel-Node gestartete Prozess auf eine Verbindungsanfrage durch **send_file** und spawned einen *file_name_receiver*, der ankommende Pakete entgegennimmt [C.12].

Der tcp_echo-Prozess auf der Ziel-Node wird mit dem Namen *freeNode* auf dieser Node registriert. Unter diesem Namen wird er im Verlauf der Arbeit auch von sämtlichen Prozessen angesprochen, die mit diesem kommunizieren müssen.

Sobald das Beam-File erfolgreich gesendet wurde, ist es möglich, den Prozess auf der neuen Node zu starten. Der Sende-Ablauf kann in folgendem Diagramm (siehe Abb. 4.1) betrachtet werden.

Nachdem dieser Systemteil erfolgreich getestet wurde, konnte mit der Implementation des ersten Migrationssystems begonnen werden. Der folgende Abschnitt beschreibt die Migration mit einem Nameservice. Dieser dient zur Lokalisierung des migrierenden Prozesses.

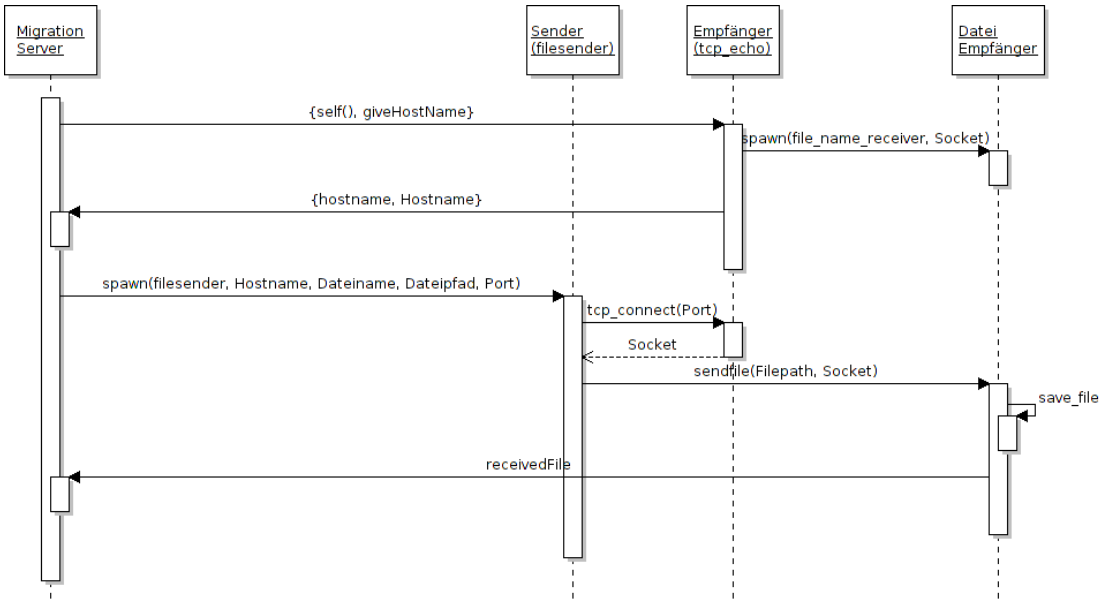

Abbildung 4.1: Ablauf beim Versenden des Beam-Files

4.3 Migration mit einem Nameservice

Bei der Migration unter Verwendung eines Nameservice handelt es sich um die erste, der in dieser Arbeit implementierten Methoden eines Migrationssystems. In diesem Fall spielt der Nameservice die Rolle der Komponente, die zu jeder Zeit über die aktuelle Position des migrierenden Prozesses Bescheid weiß. Auf diese Weise, kann die aktuelle Position jederzeit erfragt werden, um mit dem migrierten Prozess zu kommunizieren.

4.3.1 Aufbau des Testsystems

Der Testaufbau des Systems wird in Abbildung 4.2 gezeigt und besteht in diesem Fall aus **drei** Nodes. Die Erste Node is die *HQ-Node*. Auf dieser werden der Supervisor, der Nameservice und der Migration Server gestartet. Die zweite Node kann als *Quell-Node* gesehen werden. Auf ihr wird der migrierende Prozess mit seinen Startwerten gestartet. Die letzte Node in diesem Aufbau, ist die *Ziel-Node*. Auf ihr läuft der Empfängerprozess für die Beam-Datei und auf diese Node soll der Prozess der Quell-Node migriert werden.

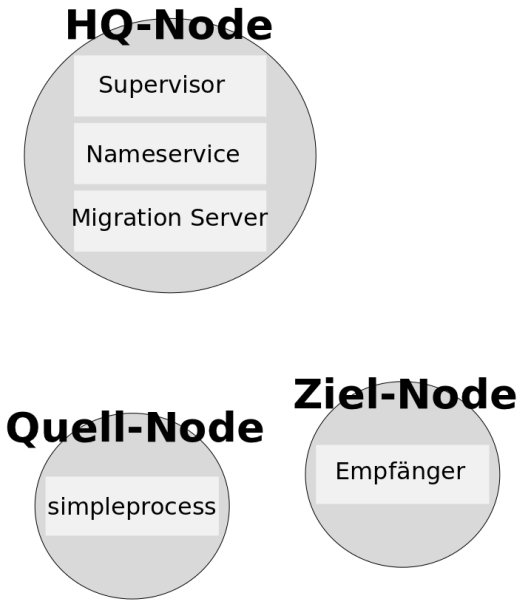

Abbildung 4.2: Aufbau des Testsystems für die Migration mit einem Nameservice

4.3.2 Komponenten

Dieser Abschnitt gibt einen Überblick über die verwendeten und entwickelten Komponenten des Systems.

Rebar

Wie angekündigt folgt eine kurze Erklärung, inwieweit *Rebar* bei der Erstellung des Systems genutzt wurde. Die zuvor beschriebene Kompilierungsfunktion (siehe 3.1.3) wird hier nicht noch einmal erklärt, wurde aber genutzt.

Rebar wurde in dieser Arbeit zur Erzeugung des Skeletts eines neuen Projekts genutzt. Der Ablauf wird im Listing 4.1 vorgestellt.

```
$> ./rebar create-app appid=migration_nameservice
==> rebar_test (create-app)
Writing src/migration_nameservice.app.src
Writing src/migration_nameservice_app.erl
Writing src/migration_nameservice_sup.erl
```

Listing 4.1: Rebar - Systemerstellung

Die folgende Erklärung der erzeugten Dateien baut auf den Quellen [AB13d] und [Chr12] auf. Das Verzeichnis *src/*, falls noch nicht vorhanden, wird hier von Rebar erstellt und die erzeugten Dateien dort abgelegt.

- **migration_nameservice.app.src** Hierbei handelt es sich um die Appliaktionsspezifikation. Diese Spezifikation gibt unter anderem Informationen über die aktuelle Versionsnummer, sowie eine Beschreibung des Projekts und es werden die Abhängigkeiten der Applikation aufgeführt. Eine genaue Beschreibung der Datei gibt die Erlang-Dokumentation [AB13d]. In dieser Spezifikation wird unter anderem festgelegt, wie die Applikation heißt und welches *Callback-Modul* die Startfunktion der Anwendung beinhaltet.

- **migration_nameservice_app.erl** Bei dieser Datei handelt es sich um das eben erwähnte *Callback-Modul*. Hier befindet sich die Startfunktion, sowie die Stopfunktion, mit der die Anwendung auch wieder angehalten werden kann.

Durch diese Dateien bekommt die Anwendung ihre Struktur, mit der es im Verlauf der Umsetzung einfacher wird das System zu starten.

Migrierender Prozess

Im folgenden Abschnitt wird näher auf die mögliche “Spielfigur” eingegangen. Also den Prozess, der im hier entwickelten System von einer Position zur nächsten migriert. Da das Hauptaugenmerk der Arbeit auf der Migration eines Prozesses liegt, und nicht darauf, wie umfangreich ein solcher Prozess sein kann, wird für die durchgeführte Migration ein sehr einfacher Prozess genutzt. Dies entspricht der beschriebenen *Minimal State Migration*.

Der hier verwendete Prozess verwaltet zwei Variablen, deren initialen Werte beim Start übergeben werden.

```
simpleprocess:start(10,20)
```

Der erste Parameter (10), wird im folgenden als "erste Variable" bezeichnet, der zweite Parameter dementsprechend als "zweite Variable". Zur Manipulation, sowie zur Anzeige dieser Variablen, wurden im *simpleprocess*-Modul mehrere API-Funktionen implementiert.

- **addto1(Value, Name)** Dieser Funktion werden ein Wert und der Name des Prozesses übergeben, dessen Variablen man ändern möchte. Daraufhin wird der übergebene Parameter *Value* auf den aktuellen Wert der ersten Variable addiert, die zum Start des Prozesses angegeben wurde.
- **addto2(Value, Name)** Diese Funktion führt analog zu addto1 die Addition auf die zweite Variable durch.
- **numbers(Name)** Durch den Aufruf dieser Funktion, gibt der Prozess den aktuellen Wert seiner beiden Variablen auf der Konsole aus.

```
1  % Der Aufruf der Funktion migrate(Process, Target)
2  % des Migration Servers, in der Form
3  % migrate(andy, 'target@thinkpad') wird an den migrierenden
4  % Prozess weitergeleitet und dort von einem receive-Block
5  % wie folgt herausgefiltert
6  ...
7  {migrate, To} ->      % "To" haelt hier das Atom 'target@thinkpad'
8      io:format("Received migrate!~n"),
9      State = [Number1, Number2, MyID],
10     gen_server:call({global, migration_server}, {migration_state, self(), State, To}),
11     loop(Number1, Number2, MyID);
12 ...
```

Listing 4.2: Initiierung der Migration

Bei Erhalt der Nachricht *{migrate, To}*, wobei die Variable *To* die Ziel-Node enthält, wird der Prozess über die anstehende Migration informiert (siehe Listing 4.2). Nach

einer Migration wird auf der Ziel-Node die Funktion **migrationComplete** gestartet. Diese Funktion hat den Nutzen, den Prozess in seinem alten Zustand, aber mit gleichem Namen, auf der neuen Node zu starten. Dazu wird der Prozess mit der Funktion **loop** *gespawned*. **loop** ist die Hauptfunktion des migrierenden Prozesses, in der sich auch der receive-Block für eingehende Nachrichten befindet.

Migration-Supervisor

Wie in Abschnitt 3 erwähnt, ermöglicht es Erlang, das System in einer baumartigen Überwachungsstruktur zu erstellen. Die folgende Abbildung 4.3 zeigt den Baum, wie er in diesem Migrationssytem zu finden ist. Dieser Prozess hat im Scotland Yard Szenario keine explizite Funktion. Bei einer Umsetzung des Spiels wäre er aber notwendig, um Ausfälle der “Spielleiter”-Prozesse zu korrigieren.

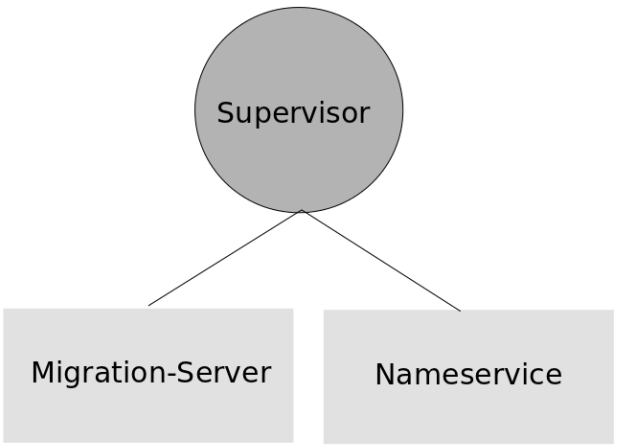

Abbildung 4.3: Supervisor-Baum des Migrationssystems

In dieser Arbeit wird nur ein Supervisor benötigt, der über die wichtigsten Komponenten des Systems wacht. Es ist allerdings nicht ungewöhnlich, dass ein Erlang-System aus einem sehr viel größeren Netz von Supervisorn und Workern (Prozesse, die kein Supervisor sind) besteht, in dem gegebenenfalls auch Supervisor-Prozesse von anderen Supervisor-Prozessen überwacht werden.

In diesem Fall ist es allerdings nicht notwendig, den Supervisor von einem weiteren Supervisor überwachen zu lassen, da es sich um ein sehr kleines System handelt.

Der Supervisor [AB13g] hat die Aufgabe, Prozesse zu starten und durch Fehler beendete Prozesse neu zu starten. Dass es sich um einen Supervisor handelt, kann an dem Modulattribut *-behaviour(supervisor)* erkannt werden.

Jedem Supervisor wird eine Restart-Strategie zugewiesen. Diese legt fest, wie ein Supervisor Prozesse neu startet. Die gewählte Restart-Strategie in diesem Fall, ist *one_for_one*. Das bedeutet, dass sobald entweder der Nameservice oder der Migration Server abstürzen, nur der abgestürzte Prozess neu gestartet wird und nicht **alle** Prozesse über die der Supervisor wacht. Die Entscheidung fiel auf diese Strategie, da die beiden Prozesse nicht voneinander abhängig sind. Damit ist sichergestellt, dass der Nameservice und der Migration Server sofort neugestartet werden, sollte es zu einem unvorhergesehenen Ausfall kommen. Eine alternative Strategie wäre zum Beispiel *one_for_all*, bei der sämtliche Kindprozess des Supervisors neugestartet werden, sobald auch nur ein Prozess unerwartet beendet wird. Kindprozesse, sind diejenigen Prozesse, auf die der Supervisor achten soll.

```
1  ...
2  NameserviceChild = {'nameservice_server', {'nameservice_server', start_link, []},
3      Restart, Shutdown, Type, ['nameservice_server']},
4  MigrationChild = {'migration_server', {'migration_server', start_link, []},
5      Restart, Shutdown, Type, ['migration_server']},
6  ...
```

Listing 4.3: Beschreibung der Kindprozesse im Supervisor

Listing 4.3 zeigt den Codeabschnitt des Supervisors, in dem die Kindprozesse beschrieben und festgelegt werden. Dabei werden zugleich die Funktionen angegeben, die bei Start, bzw. Neustart ausgeführt werden. In diesem Fall sind das in beiden Modulen (nameservice_server und migration_server) die Funktionen **start_link**, denen keine Argumente übergeben werden müssen.

Im Supervisor-Modul werden zudem noch Werte festgelegt, die bestimmen, wie oft der Nameservice und Migration Server innerhalb einer Zeitspanne maximal neugestartet werden. Dafür sind die Werte der Variablen *MaxRestarts* und *MaxSecondsBetweenRestarts* verantwortlich. Diese wurden bewusst klein gewählt, da es im Zuge dieser Arbeit nicht notwendig ist, sehr viele Restarts durchzuführen bis die Prozesse wieder wie gewünscht laufen. Sollten zu viele Restarts benötigt werden, wird von einem schwerwiegenderen Problem ausgegangen. Um dieses Problem zu lösen, müssten die Prozesse letztendlich sowieso angehalten werden.

Während der Entwicklung des Systems kam es durch Programmierfehler innerhalb einiger Funktionen zu Abstürzen eines Kindprozesses. Dabei konnte beobachtet werden, dass der Neustart des betroffenen Prozesses nur etwa 1 Sekunde dauerte. Das ist deswegen hervorzuheben, da es für die Migration, in der der migrierende Prozess möglichst dauerhaft erreichbar sein soll, von Bedeutung ist, wenn essentielle Prozesse schnell wieder erreichbar sind. Erlang ist in diesem Fall, durch diese spezielle Eigenschaft, durchaus hilfreich.

Nameservice

Als nächstes wird auf den *Nameservice* eingegangen. Wie schon erwähnt, ist es die Aufgabe des Nameservice, ständig über die aktuelle Position der migrierenden Prozesse Bescheid zu wissen.

Man kann sich den Nameservice im Scotland Yard Szenario also als denjenigen Prozess vorstellen, der das Spielende feststellen würde. Aufgrund der Tatsache, dass er wie erwähnt über die Positionen Bescheid weiß, wüsste er auch wie oft die einzelnen Prozesse schon migriert sind und wo sie sich im Moment befinden. Letzteres könnte genutzt werden, um festzustellen, ob sich ein Agentenprozess auf der selben Node wie der Verbrecherprozess befindet. Das würde laut Regeln zu einem Sieg der Agenten führen, und das Spiel würde damit beendet werden. Die Anzahl der durchgeführten Migrationen würden genutzt werden, um die aktuelle Anzahl der verbleibenden *Tickets* der Prozesse festzuhalten. Dafür könnte der Nameservice einen Zähler besitzen, der mit der zum Start vorgesehenen Anzahl an Tickets startet. Sobald der Zähler **0** erreicht, wäre das sekundäre

Ziel des Spiels erreicht und der Verbrecherprozess hätte gewonnen.

Wie der Nameservice jederzeit über die aktuelle Position bescheid weiß und auf was bei der Benutzung des Nameservice zu achten ist, wird im folgenden Abschnitt erklärt.

Ähnlich dem Supervisor wird der Nameservice mit dem Attribut *-behaviour(gen_server)* als Server definiert. Wie der Name schon sagt, handelt es sich hierbei allerdings um einen Server, und nicht um einen Supervisor. Das Modul bindet ein Headerfile ein, indem ein Record definiert ist, das den Zustand des gestarteten Nameservice darstellt. Dieses Record enthält ein Dictionary, in dem der Name eines Prozesses auf seine aktuelle Position abgebildet wird. Diese Datenstruktur ist vergleichbar mit einer Hashmap, wie sie in Java benutzt wird. Auf diese Weise speichert der Nameservice alle Prozesse ab, die sich bei ihm anmelden.

Um jedem Prozess, der sich anmeldet, einen eigenen Namen zuzuweisen, und die Anzahl der möglichen Anmeldungen zu beschränken, wurde die Funktion **getFreeId** implementiert. Diese vergibt Namen der Form *10010*, und zwar von 10001 bis 10011. Hierbei handelt es sich beim entgültigen Namen um ein *Atom*, und nicht um einen *Integer-Wert*. Dies hat den Vorteil, dass der Name schon zur Begrenzung der Anzahl der teilnehmenden Prozesse genutzt werden kann. Dazu wird das Atom während der Validierung in einen Integer-Wert umgewandelt. Nach einigen Testläufen ohne diese Funktion stellte sich heraus, dass die Wahl eines beliebigen Namens zu mehreren Problemen führte:

- Bei der weiteren Verarbeitung des Namen (z.B. mit Funktionen wie **atom_to_list**), kam es zu Fehlermeldungen, da Erlang Namen die nur aus Zahlen bestehen, als Integer interpretierte. Namen die dagegen aus Buchstaben bestanden, wurden korrekt als Atome erkannt. atom_to_list(test) würde dementsprechend keine Probleme bereiten. atom_to_list(1234) würde dagegen einen Fehler erzeugen, da es sich bei 1234 nicht um ein Atom handelt.
- Bei der Wahl eines schon vergebenen Namen muss die Rückmeldung vom migrierenden Prozess ausgewertet und darauf reagiert werden. Dies widersprach der Idee, diesen Prozess so klein und einfach wie nur möglich zu halten.

Nach Erhalt des Namen kann sich ein Prozess mit der Funktion **bind** bei dem Nameservice anmelden. An dieser Stelle muss ein Prozess seinen Namen und seine aktuelle Node übergeben. Sollte der Name noch nicht vergeben sein, so wird der Prozess dem Dictionary hinzugefügt, und kann von da an beim Nameservice erfragt werden. Dazu dient die Funktion **lookup**, die einem das Tupel *{ProcessName, Node}* liefert, wenn der ProzessName im Dictionary gefunden wurde. Mit diesem Tupel und der Sendefunktion von Erlang kann daraufhin sofort mit dem Prozess kommuniziert werden.

Da sich jeder Prozess nach einer erfolgreichen Migration mit seiner neuen Position beim Nameservice melden muss, wurde die Funktion **rebind** implementiert. Diese hat gegenüber *bind* den Vorteil, dass nicht zuerst überprüft wird, ob der Name schon im Dictionary hinterlegt ist. Es wird davon ausgegangen, dass die Funktion nur von Prozessen verwendet wird, die sich "zurückmelden".

Migration Server

Die Aufgabe des *Migration Server* ist die Durchführung der Migration eines Prozesses. Er ist daher im weitesten Sinne vergleichbar mit dem *virtualizing Executive* aus Kapitel 2. Im Scotland Yard Szenario könnte man ihn als eine Art Spielleiter betrachten, dessen Aufgabe es ist, für die korrekte Bewegung der Spielfiguren zu sorgen.

Angestoßen wird die Migration mit dem Aufruf der Funktion **migrate**. Dieser Funktion wird der Name des Prozesses übergeben, welcher migriert werden soll, sowie der Name der Ziel-Node.

Der Migration Server ist dabei auch für das Versenden der BEAM-Datei zuständig, wie im Abschnitt 4.2 erwähnt wurde.

4.3.3 Migrationsablauf

Im folgenden wird der genaue Ablauf einer Migration im System mit einem Nameservice beschrieben.

Start des Testsystems

Zuerst werden die oben beschriebenen Nodes gestartet. Dazu dient der Aufruf der Erlang-Shell, wie in Listing 4.4 am Beispiel der HQ-Node zu sehen ist.

```
$> erl -pa ebin -sname hq -setcookie secret
```

Listing 4.4: Start des Testsystems

Analog dazu werden noch die Quell-, und die Ziel-Node gestartet. Diese entsprechen den verschiedenen Stationen eines Scotland Yard Spielfeldes. Der Parameter für -sname wird dabei entsprechend angepasst. Hier wurde zur Verdeutlichung Quell- und Ziel-Node gewählt. Diese Namen sind natürlich variabel, und würden gegebenenfalls die Namen von Haltestellen in London annehmen, um das Spielfeld getreu dem Originalspiel nachzuempfinden. Nachdem die Nodes bereit sind, kann die Applikation auf der HQ-Node gestartet werden.

Hier kommt die Applikationsspezifikation aus Abschnitt 4.3.2 zum Einsatz. Bevor die dort erwähnte Anwendungsstruktur integriert wurde, war es notwendig jeden einzelnen Prozess manuell zu starten. Dazu wurde die **Startfunktion** von jedem Modul aufgerufen, um die Anwendung zum Laufen zu bekommen. Dank der Struktur mit Applikationsspezifikation und Callback-Modul ist nur noch folgender Aufruf auf der HQ-Node notwendig:

```
1> application:start(migration_nameservice).
ok
2>
```

Listing 4.5: Start der Anwendungen auf der HQ-Node (Nameservice-System)

Dieser Befehl ruft die Funktion **start** des Supervisors auf, dieser startet daraufhin sowohl den Migration Server, als auch den Nameservice.

Anschließend wird auf der Quell-Node der migrierende Prozess gestartet. Dazu wird

die Funktion **start** des Moduls *simpleprocess* aufgerufen, und dabei die Startwerte der beiden Variablen, die den Zustand des Prozesses beschreiben, als Parameter übergeben. Der Start des Prozesses wird durch die Ausgabe in Listing 4.6 bestätigt. Anschließend kann die Funktionalität durch die in 4.3.2 beschriebenen API-Funktionen getestet werden.

```
1> simpleprocess:start(10,20).
Received ID: '10001'
ok
2>
```

Listing 4.6: Start des migrierenden Prozesses

Um den Systemaufbau zu komplettieren, wird auf der Ziel-Node noch der Empfänger-Prozess *tcp_echo* gestartet. Dazu wird die Funktion **start**, unter Angabe des verwendeten Ports als Parameter, aufgerufen. Dieser Port wird genutzt, um die Beam-Datei zu übertragen. In diesem Fall handelt es sich dabei um den Port **8080**. Anschließend ist das System bereit eine Migration durchzuführen.

Durchführung der Migration

Nachdem alle Vorbereitungen, wie oben beschrieben, getroffen wurden, kann mit der Migration begonnen werden. Es wird also der Mechanismus beschrieben, der nötig wäre, um eine Scotland Yard Spielfigur von einer Station zur nächsten wechseln zu lassen (dies entspricht dem Ziehen einer Figur auf dem Spielfeld). Dabei spielt es keine Rolle, ob es sich um einen Agentenprozess oder den Verbrecherprozess handelt.

Die Migration wird mit der Funktion **migrate** des Migration Servers gestartet. Daraufhin beginnt folgender Ablauf (siehe: Abb. 4.4).

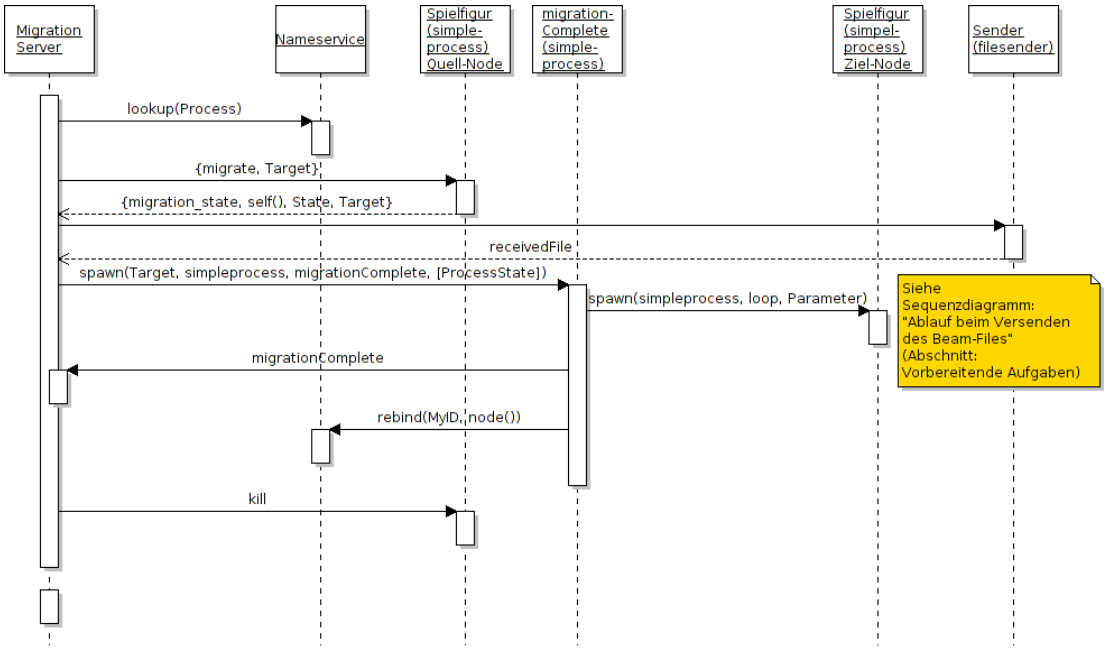

Abbildung 4.4: Ablauf der Migration

Nach dem Aufruf der **migrate**-Funktion, wird zuerst geprüft, ob ein Prozess mit dem übergebenen Namen existiert. Trifft das zu, teilt der Migration Server dem migrierenden Prozess (*simpleprocess*), das wäre an dieser Stelle die Spielfigur, mit, dass eine Migration initiiert wurde. Dazu wird die Position des Prozesses mit Hilfe des Namen, der der **migrate**-Funktion übergeben wurde, beim Nameservice abgefragt. Diese Position wird genutzt, um dem migrierenden Prozess die Nachricht *{migrate, Target}* zu schicken.

Daraufhin antwortet der Prozess dem Migration Server mit seinem aktuellen Zustand als Liste (die beiden Werte seiner Variablen und der Prozessname). Anschließend findet der in Abschnitt 4.2 beschriebene Ablauf statt, um die benötigte Beam-Datei zu der Ziel-Node zu schicken.

Nachdem die Datei an die Ziel-Node gesendet wurde, kann die eigentliche Migration fortgeführt werden. Dazu wird die Funktion **migrationComplete** des Moduls *simpleprocess* auf der Ziel-Node aufgerufen. Dies wird ebenfalls vom Migration Server durchgeführt. Dieser Funktion wird der zuvor gesicherte Zustand als Parameter übergeben.

Die Aufgabe dieser Funktion ist es, den migrierten Prozess auf der Ziel-Node mit dem alten Zustand zu starten. Dazu wird der Name des alten Prozesses genutzt, um den neuen Prozess unter dem selben Namen zu registrieren und diesen mit der neuen Node beim Nameservice zu hinterlegen (mit Hilfe der **rebind**-Funktion). Daraufhin wird dem Migration Server mitgeteilt, dass die Migration abgeschlossen ist, indem die Nachricht *migrationComplete* geschickt wird. An dieser Stelle ist deutlich zu sehen, inwiefern sich die hier entwickelte Methode und die in Abschnitt 2 beschriebene ähneln. Ebenso wie in der dort beschriebenen Methode, wird auf der Ziel-Node ein gänzlich *neuer* Prozess gestartet. Während der betriebssystemnahe Ansatz erst anschließend den Heap wiederherstellt und sämtliche andere Informationen in den Prozess einfügt, wird hier der Zustand des alten Prozess gleich mit der Startfunktion übergeben. Dadurch übernimmt der migrierte Prozess den Zustand sofort, sobald er auf der Quell-Node gestartet wurde.

Erst jetzt wird der Migration Server den alten Prozess auf der Quell-Node mit Hilfe der *kill*-Nachricht beenden. Dadurch ist sichergestellt, dass der alte Prozess bei einem Fehlschlag der Migration nicht zu früh beendet wird. Die “Spielfigur” wird also auch weiterhin auf der alten Node exisiteren, sollte die Migration nicht erfolgreich verlaufen.

Wie der erfolgreiche Verlauf einer Migration im Testsystem festgestellt werden kann, beschreibt der folgende Abschnitt.

Überprüfung des Erfolgs einer Migration

Im Folgenden wird gezeigt, wie der Erfolg einer Migration festgestellt wird. Zur Überwachung der Migration wird auf der Quell- und Ziel-Node der Prozessmanager *pman* [AB13f] gestartet. Der Prozessmanager zeigt alle Prozesse an, die auf einer Node laufen und gibt dabei Auskunft über Namen, sowie weitere Informationen, wie Anzahl der erhaltenen Nachrichten oder aktuelle PID. Zu den einzelnen Prozessen können zudem noch detailliertere Informationen abgerufen werden.

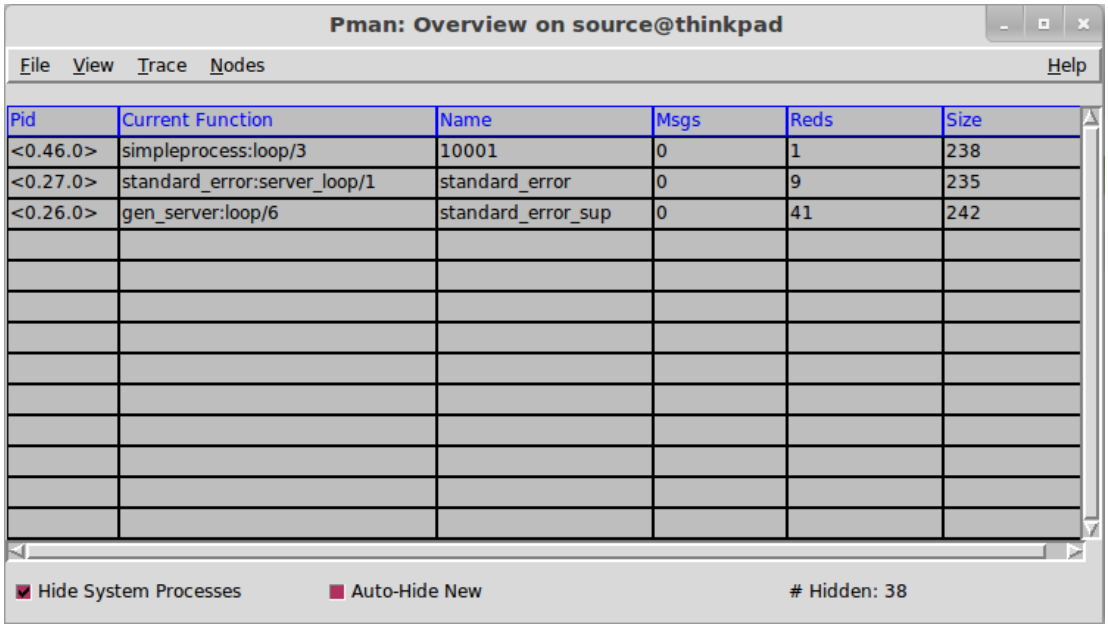

Abbildung 4.5: Prozessmanager - Auflistung aller Prozesse (ohne System Prozesse)

Der Erfolg der Migration kann wie folgt festgestellt werden. Da die Liste der aufgeführten Prozesse automatisch aktualisiert wird, kann nach einer Migration gesehen werden, wie der migrierte Prozess im Prozessmanager der Quell-Node verschwindet und im Prozessmanager auf Seiten der Ziel-Node auftaucht.

Da der Erhalt des Prozesszustandes zu den Anforderungen zählt, die zu Beginn an das System gestellt wurden, ist es notwendig diesen nach einer Migration zu überprüfen. Dazu zählen sowohl der Prozessname, als auch die Werte der beiden schon erwähnten Variablen. Dies entspräche im Szenario von Scotland Yard dem Wunsch, dass ein Spieler

seinen Namen behält, sobald er an einer neuen Station ankommt oder seine vollständige Liste der bisher besuchten Stationen unverändert bleibt.

Getestet wird in diesem Fall auf zwei Wege. Zum einen dient der eben vorgestellte Prozessmanager als Möglichkeit den Namen des Prozesses vor und nach der Migration zu erfahren, zum anderen werden die festgelegten Startwerte des migrierenden Prozesses vor und nach der Migration mit Hilfe der API-Funktion **numbers** angezeigt.

Dieser Vorgang dient hier als Test und muss bei einem realen Einsatz des Systems nicht bei jeder Migration durchgeführt werden.

4.4 Migration mit einer Homestation

In diesem Abschnitt wird die alternative Version des Migrationssystems vorgestellt, die mit Hilfe einer Homestation arbeitet. Die Homestation dient hierbei als Proxy zwischen dem Sender und dem migrierenden Prozess. Diese Methode hat einige Vorteile gegenüber der eben vorgestellten Implementation mit einem Nameservice.

Zu diesen Vorteilen zählt unter anderem, dass der Nameservice nicht mehr die Aufgabe hat, dem migrierenden Prozess im Vorfeld einen Namen zuzuweisen. Da die Homestation als Proxy dient, kann ihr Name als Name des migrierenden Prozesses angesehen werden, über den er jederzeit erreichbar ist. Das wird erreicht, indem die Homestation eines Prozesses immer den gleichen Namen behält. Dieser kann beim Start einer Homestation vom Nutzer gewählt werden. Diese Homestations können auch während des laufenden Betriebs hinzugefügt werden. Es ist daher nicht notwendig, den Namen des eigentlich migrierenden Prozesses zu kennen. Der Vollständigkeit halber sei erwähnt, dass dieser Prozess im weiteren Verlauf der Arbeit unter dem Namen *simpleprocess* registriert wird.

Einen weiteren Vorteil stellt die globale Registrierung der Homestation dar. Da diese zu jedem Zeitpunkt die aktuelle Position ihres migrierenden Prozesses hält, wird es daher unnötig, diese Position bei dem Nameservice abzufragen. Wie eben beschrieben reicht es daher, den Namen der Homestation des Prozesses zu kennen, mit dem man

kommunizieren möchte.

Des Weiteren wurde in dieser Implementation die in Abschnitt 1.3 erwähnte Optimierung in Form einer *direkten* Kommunikation zum migrierenden Prozess integriert.

Auch wenn dadurch auf den ersten Blick der Vorteil der Kommunikation über die Homestation hinfällig wird, hat diese Optimierung seinen Nutzen. Zur Erklärung soll das Szenario dienen, indem sich der *Sender* in sehr kurzer Sendedistanz zum migrierenden Prozess befindet (siehe Abb. 4.6). Sobald diese Entfernung kleiner ist als die zur Homestation, kommt es bei einer direkten Kommunikation zu einem Zeitgewinn und somit zu einer Optimierung des Sendevorgangs.

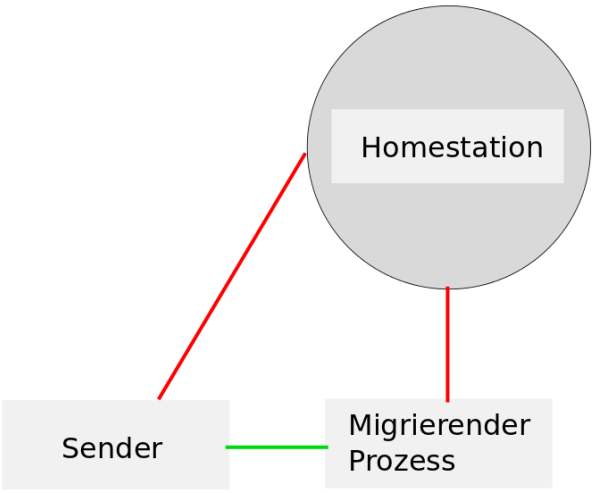

Abbildung 4.6: Vorteil einer direkten Kommunikation

4.4.1 Aufbau des Testsystems

Der Aufbau des Systems mit einer Homestation unterscheidet sich nur gering mit dem aus Abschnitt 4.3.1. Einzig die *Homestation* selbst wird zusätzlich vom Supervisor auf der HQ-Node gestartet. Dabei entsteht der Aufbau aus Abbildung 4.7.

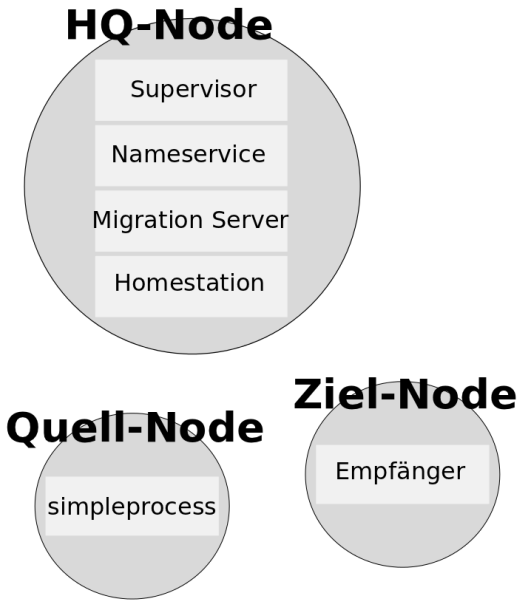

Abbildung 4.7: Aufbau des Testsystems für die Migration mit einer Homestation

An dieser Stelle ist zu erwähnen, dass der Supervisor zu Testzwecken schon eine Homestation mit Namen “andy” startet. Weitere Homestations werden daraufhin mit der Funktion **start** des Homestation-Moduls hinzugefügt:

```
1  start (HSName) ->
2      net_adm:ping('hq@thinkpad'),
3      ChildSpec = {HSName, {'homestation_server', start_link ,[HSName]},
4          permanent ,2000, worker, ['homestation_server']},
5      supervisor: start_child ({global , 'migration_sup'}, ChildSpec ).
```

Listing 4.7: Funktion zum Hinzufügen einer Homestation

Der Parameter *HSName* ist dabei der gewünschte Name. Wie in Listing 4.7 zu sehen ist, wird beim Hinzufügen einer Homestation die Fähigkeit eines Erlang-Supervisors genutzt, um ihm dynamisch Prozesse hinzuzufügen. Dazu wird zuerst in Zeile 3 eine

Spezifikation des Prozesses erstellt, die daraufhin als Parameter der Funktion **start_child** des Supervisors übergeben wird. Nachdem dadurch eine weitere Homestation dem System hinzugefügt wurde, kann ein weiterer migrierender Prozess gestartet werden und sich bei dieser Homestation anmelden.
Zu diesem Zeitpunkt betreut jede Homestation nur genau einen migrierenden Prozess. Sie ist auch nicht darauf ausgelegt, mehrere Prozesse gleichzeitig zu verwalten.
Sämtliche anderen Prozesse, die zuvor erwähnt wurden, werden weiterhin benötigt und bleiben daher auf der HQ-Node (Supervisor, Nameservice, Migration Server).

4.4.2 Komponenten

In diesem Abschnitt wird vor allem auf die Komponenten eingegangen, die sich in ihrer Funktionalität geändert haben, sowie auf die Homestation selbst. Daher soll folgender Abschnitt einen kurzen Überblick über die unveränderten Komponenten geben.

Unveränderte Komponenten

Zu diesen unveränderten Komponenten gehört der gesamte Teil des Systems, der sich mit der *Dateiübertragung* beschäftigt, wie er in Abschnitt 4.2 beschrieben ist. Zudem wurde auch in diesem System *Rebar* (4.3.2) eingesetzt, um die Skelette der benötigten Dateien zu generieren. Hinzu kam in diesem Fall noch die Generierung des Grundgerüsts der Homestation.

Auch der *Migration-Supervisor* ist in seiner Funktion gleich geblieben und hat einzig die zusätzliche Aufgabe bekommen, die Homestation zu starten und zu überwachen. Dazu wird analog zu der in Abschnitt 4.3.2 beschriebenen Vorgehensweise die Homestation zu den zu überwachenden Prozessen hinzugefügt. Der *Migration Server* konnte ohne Veränderungen übernommen werden und hat auch in diesem System die gleiche Aufgabe, wie schon zuvor beschrieben.

Homestation

Das einzige Modul, das nur in dieser Version des Migrationssystems vorkommt, ist die Homestation. Wie zuvor beschrieben, handelt es sich hierbei um einen Proxy zwischen

dem Sender und dem migrierenden Prozess. Der Homestation wird beim Start ein im System eindeutiger Name zugewiesen. Anschließend können die API-Funktionen des Moduls *homestation_server* genutzt werden, um mit dem entfernten Prozess zu kommunizieren. Darunter befinden sich auch sämtliche Funktionen, die gebraucht werden, um die Variablen des migrierenden Prozesses zu manipulieren.

start(HSName) Wie schon zuvor erläutert, handelt es sich hierbei um die Funktion, die genutzt wird, um dem System weitere Homestations hinzuzufügen.

newPos(Pos, HSName) Mit dieser Funktion kann der Homestation die neue Position ihres zugehörigen Prozesses nach einer Migration mitgeteilt werden. Sie wird daher vom migrierenden Prozess selbst verwendet.

clientPos(HSName) Beim Aufruf dieser Funktion, antwortet die Homestation mit der aktuellen Position ihres korrespondierenden Prozesses.

send(Msg, HSName) Diese Funktion stellt die Grundlage dar, mit der dem migrierenden Prozess eine beliebige Nachricht gesendet werden kann. Sie wurde vor allem für eine einfache Erweiterbarkeit der Befehle implementiert.

addto1(Value, HSName) Analog zu der gleichnamigen API-Funktion des simpleprocess-Moduls wird mit dieser Funktion die erste Variable des Prozesses manipuliert.

addto2(Value, HSName) Siehe *addto1(Value, HSName)*, mit dem Unterschied, dass die zweite Variable geändert wird.

numbers(HSName) Damit wird der korrespondierende Prozess aufgefordert, die aktuellen Werte seiner beiden Variablen auf die Konsole zu schreiben. Wie schon beschrieben, dient diese Funktion vor allem der Überprüfung der beiden Variablen vor und nach einer Migration.

request_direct_conversation(HSName) Initiiert die direkte Kommunikation mit dem migrierenden Prozess.

Diverse weitere Funktionen Zusätzlich zu den hier aufgezählten Funktionen existieren sämtliche Befehle, die genutzt werden um mit dem migrierenden Prozess zu

kommunizieren, auch in einer Variante für die direkte Kommunikation. Dabei folgen die Funktionen dem Muster: **direct_msg_addto1(Value, HSName)**. Die Wirkung bleibt dabei unverändert.

Eine wichtige Rolle für die Aufrechterhaltung der Verbindung zwischen Homestation und dazugehörigem Prozess stellt die Funktion **newPos** dar. Diese wird vom migrierenden Prozess beim Start, sowie nach jeder Migration zu einer neuen Node genutzt, um der Homestation seine aktuelle Position mitzuteilen. Diese Position wird daraufhin bis zur nächsten Migration von der Homestation verwaltet und kann mit Hilfe der Funktion **clientPos** abgefragt werden. Zum Speichern dieser Position dient der *Record*, der den Zustand der Homestation darstellt und über die Header-Datei eingebunden wird.

Die Homestation ist zusätzlich zu ihrer Funktion als Proxy noch für die Initiierung der direkten Kommunikation zwischen Sender und migrierendem Prozess zuständig. Dieser Vorgang wird im folgenden Abschnitt, nach einer kurzen Erklärung der Änderungen des migrierenden Prozesses, erläutert.

Migrierender Prozess

Der migrierende Prozess wurde, gegenüber dem zuvor Vorgestellten, an einigen Stellen verändert, um sich an die Vorgehensweise mit einer Homestation anzupassen. Eine kleine Änderung fand in der Startfunktion statt. An dieser Stelle wird nun, zusätzlich zu den beiden Zahlwerten, der Name der korrespondierenden Homestation als dritter Parameter übergeben. Damit wird dem migrierenden Prozess mitgeteilt, bei welcher Homestation er sich mit seiner aktuellen Position melden muss.

```
simpleprocess:start(10,20,andy)
```

Die größten Veränderungen wurden vorgenommen, um den direkten Sendevorgang durchführen zu können. Der *receive*-Ausdruck für **numbers** soll hier zur Verdeutlichung der Änderungen dienen (siehe Listing 4.8).

Wie zu sehen ist, wurde dem receive-Ausdruck für **numbers** noch ein *From* hinzugefügt. In der Homestation-Version des Systems fügt jeder sendenden Prozess seiner Nachricht noch seine PID hinzu. Diese wird beim Empfang daraufhin auf die Variable “From”

```
1  receive
2      ...
3      {From, numbers} ->
4          io:format("Number1 = ~p, Number2 = ~p~n", [Number1, Number2]),
5          From ! ok,
6          loop(Number1, Number2, MyID, HSName);
7      ...
8  end.
```

Listing 4.8: Änderungen am migrierenden Prozess

gematched. Wie zuvor beschrieben, ermöglicht das den direkten Sendevorgang. Durch das Mitsenden der eigenen PID kann der migrierende Prozess antworten. Das wird im weiteren Verlauf der direkten Kommunikation von Bedeutung sein.

Initiiert wird die direkte Kommunikation wie zuvor erwähnt, durch die Homestation mit Hilfe der Funktion **request_direct_conversation** (siehe Listing 4.9). Dieser Funktion wird der Name der Homestation (hier die Variable *HSName*) übergeben, mit deren korrespondierendem Prozess man in Verbindung treten möchte.

```
1  request_direct_conversation (HSName) ->
2      gen_server:cast({global, HSName}, {send, {self(), direct_msg_request}}),
3      receive
4          {direct_msg_answer, DirectPid} ->
5              io:format("Received DirectPid and saved it: ~p~n", [DirectPid]),
6              file: write_file("./"++atom_to_list(HSName),
7                               io_lib:fwrite("~p",[DirectPid]))
8          after
9              5000 ->
10                 io:format("No DirectPid received~n")
11     end.
```

Listing 4.9: Initiierung der direkten Kommunikation

Wie hier (4.9) zu sehen ist, wird in Zeile 2 die aktuelle PID des korrespondierenden Prozess erfragt. Die Antwort der Homestation wird daraufhin ausgewertet. Falls innerhalb von 5 Sekunden keine Antwort erhalten wird, wird das dem Nutzer mitgeteilt (Zeile

10). Anschließend wird diese PID in einer lokalen Datei abgespeichert, um im weiteren Verlauf der Kommunikation genutzt zu werden.

Wurde eine "direkte PID" erfolgreich erhalten, kann eine der erwähnten API-Funktionen des Homestation-Moduls für die direkte Kommunikation genutzt werden (siehe Listing 4.10).

```
1  direct_msg_numbers(HSName) ->
2      get_pid_from_file (HSName) ! {self (), numbers},
3      receivefun ().
```

Listing 4.10: API-Funktion zur direkten Kommunikation

In dieser Beispielfunktion wird wieder der Name der Homestation des Prozesses übergeben, mit dem man kommunizieren will. Anschließend wird die Funktion **get_pid_from_file** genutzt, um die zuvor in der Datei gespeicherte PID auszulesen. An diese PID wird jetzt die gewünschte Nachricht, in diesem Fall *numbers*, sowie die eigene PID gesendet.

Die Funktion **receivefun** (Zeile 3) ist der Grund, warum im migrierenden Prozess die Antwort-Funktionalität implementiert wurde. Die einzige Aufgabe dieser Funktion ist es, auf die Antwort des Empfängers, also des migrierenden Prozesses, zu warten. Sollte diese Antwort nicht innerhalb von 5 Sekunden eintreffen, wird davon ausgegangen, dass die PID, die zuvor aus der Datei ausgelesen wurde, nicht mehr aktuell ist. Daraufhin wird dem Nutzer vorgeschlagen, diese PID durch den erneuten Aufruf der oben beschriebenen Funktion **request_direct_conversation** zu erneuern.

Dieser Ablauf wurde gewählt, um zu verhindern, dass der migrierende Prozess eine Liste mit allen Prozessen halten muss, die direkt mit ihm kommunizieren oder kommuniziert haben. Da er nach jeder Migration eine neue PID erhält, wäre diese Liste notwendig, um jedem dieser Prozesse seine aktuelle PID mitzuteilen. Das kann in folgenden Fällen schnell unpraktisch werden:

- Einer oder mehrere Prozesse in der Liste kommunizieren schon länger nicht mehr mit dem migrierenden Prozess. Um dieses Problem zu beheben, müsste ein

Mechanismus implementiert werden, der inaktive Prozesse aus dieser Liste löscht. Das würde den Umfang des Prozesses deutlich erhöhen.

- Prozesse, die nicht dauerhaft laufen, und somit nicht imstande sind die aktuelle PID zu halten, müssten immer eine neue PID anfordern. Dies ist nicht der Fall, wenn die PID in einer lokalen Datei gespeichert wurde.
- Das Mitführen der Liste vergrößert den Zustand des migrierenden Prozesses. Dieser sollte aber so klein wie möglich gehalten werden, um eine erfolgreiche und schnelle Migration zu gewährleisten.

Nameservice

Sowohl Funktionsumfang, als auch Funktionsweise des Nameservice wurden nicht verändert. Allerdings bekommt der Nameservice bei der Migration mit einer Homestation eine neue Aufgabe.

Bisher wurde er eingesetzt, um die migrierenden Prozesse zu verwalten. Dadurch konnte zu jeder Zeit die aktuelle Position eines jeden Prozesses erfragt werden, der sich bei dem Nameservice angemeldet hat. In dem hier beschriebenen System ist dies aufgrund der Homestation nicht mehr nötig. Das ist der Grund, warum der Nameservice in der ersten Version dieses Systems mit einer Homestation gar nicht erst benötigt wurde. Zu diesem Zeitpunkt stützte sich das System allein auf die globale Registrierung der Prozesse. Allerdings machte eine Eigenheit Erlangs den erneuten Einsatz dieses Moduls notwendig.

Wie im Abschnitt 4.4.2 beschrieben, hat jede Homestation einen im System eindeutigen Namen. Mit diesem Namen registriert sich die Homestation beim Start global und kann daraufhin zu jeder Zeit über diesen Namen kontaktiert werden. Wenn eine weitere Homestation im System gestartet wird, die den gleichen Namen als Parameter übergeben bekommt, verhindert Erlang, dass sich dieser Prozess auch global registrieren kann. Dadurch wird eine Zwei- oder Mehrdeutigkeit eines Prozessnamen im System verhindert.

Ein Problem entsteht, wenn zwei Homestations mit gleichem Namen auf zwei unterschiedlichen Nodes gestartet werden, die sich zu diesem Zeitpunkt noch nicht “kennen”

(siehe Abb. 4.8). Wie in Abschnitt 3.2.2 beschrieben, müssen sich Nodes erst untereinander bekannt machen, damit eine Kommunikation zwischen ihnen stattfinden kann. Sobald diese beiden Nodes, auf denen gleichnamige Homestations laufen, in Verbindung treten, ist es daher an Erlang, diesen Konflikt aufzulösen. Erlang wird in diesem Fall zufällig einen der beiden Prozesse beenden [Heb10]. Da diese Zufälligkeit keine Grundlage für einen zuverlässigen Umgang mit den Homestations ist, wird der Nameservice benutzt.

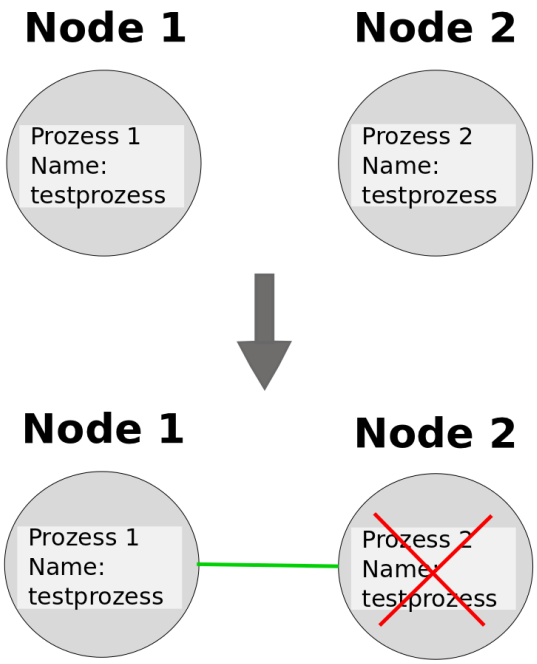

Abbildung 4.8: Namenskonflikt bei globaler Registrierung in Erlang

Die Aufgabe des Nameservice besteht in diesem Fall darin, die Namen der Homestations zu verwalten und sicherzustellen, dass ein Konflikt wie eben beschrieben nicht auftreten kann. Dies geschieht durch folgenden Codeabschnitt im Homestation-Modul (siehe 4.11).

```
1  start_link (HSName) ->
2      State =# state {stationnamen=HSName},
3      case nameservice_server:lookup(HSName) of
4          {_Node, _Name} ->
5              io:format("Name ~p already taken~n", [HSName]);
6          not_found ->
7              nameservice_server:bind(HSName, node()),
8              gen_server: start_link ({global, HSName}, ?MODULE, [State], [])
9      end.
```

Listing 4.11: Verfügbarkeitsabfrage des gewünschten Namen

Die Funktion **lookup** (Zeile 3) des Moduls *nameservice_server* stellt fest, ob der übergebene Name schon vergeben ist. Ist dies der Fall, bekommt der Nutzer die Ausgabe, dass der Name nicht verfügbar ist (Zeile 5). Andernfalls wird der Name beim Nameservice hinterlegt und die Homestation gestartet (Zeile 7+8). Dabei wird der Name global registriert.

Der Nameservice dient in diesem Falls also zur Validierung des gewünschten Namen einer Homestation. Es ist nicht mehr seine Hauptaufgabe, die Positionen der einzelnen Prozesse zu verwalten.

4.4.3 Migrationsablauf

Wie schon im Abschnitt des Nameservice-Systems, wird im Folgenden der genaue Ablauf einer Migration mit Hilfe einer Homestation beschrieben.

Start des Testsystems

Das Testsystem wird auf die gleiche Weise gestartet, wie in Abschnitt 4.3.3. Einzig der Name der Applikation ändert sich. Dies liegt an dem Eintrag in der Applikationsspezifikation. Daher wird der Befehl aus Listing 4.12 genutzt, um die Prozesse auf der HQ-Node zu starten.

Wie schon erwähnt wurde die Homestation dem Migration-Supervisor hinzugefügt, sodass es nicht notwendig ist, diese manuell zu starten.

```
1> application:start(migration).
ok
2>
```

Listing 4.12: Start der Anwendungen auf der HQ-Node (Homestation-System)

Der Start des migrierenden Prozesses auf der Quell-Node verläuft analog zu dem schon bekannten Aufruf (siehe 4.13).

```
1> simpleprocess:start(10,20,andy).
ok
2>
```

Listing 4.13: Start des migrierenden Prozesses

Der Parameter *andy* steht hier für den Namen der Homestation.

Beim Start des *tcp_echo*-Prozesses auf der Ziel-Node hat sich im Vergleich zum vorherigen System nichts geändert. Er wird also weiterhin mit seiner **start**-Funktion gestartet und dient daraufhin als Empfänger der zuvor beschriebenen Beam-Datei.

Durchführung der Migration

Die Migration unterscheidet sich im Wesentlichen von der ohne Homestation in dem Punkt, dass die Kommunikation zwischen Migration Server und dem migrierenden Prozess zu Beginn einer Migration über die Homestation läuft. Als weiterer Unterschied ist zu erwähnen, dass sich der migrierte Prozess am Ende der Migration mit seiner neuen Position bei der Homestation meldet. Der restliche Ablauf der Migration ist identisch mit der schon vorgestellen. Nichtsdestotrotz beschreibt der Ablauf 4.9 die Migration mit einer Homestation noch einmal genau, um einen direkten Vergleich zwischen den beiden Methoden ziehen zu können.

Auf die Überprüfung des Erfolgs der Migration wird in diesem Fall nicht näher eingegangen, da er sich von der in Abschnitt 4.3.3 vorgestellten nicht unterscheidet.

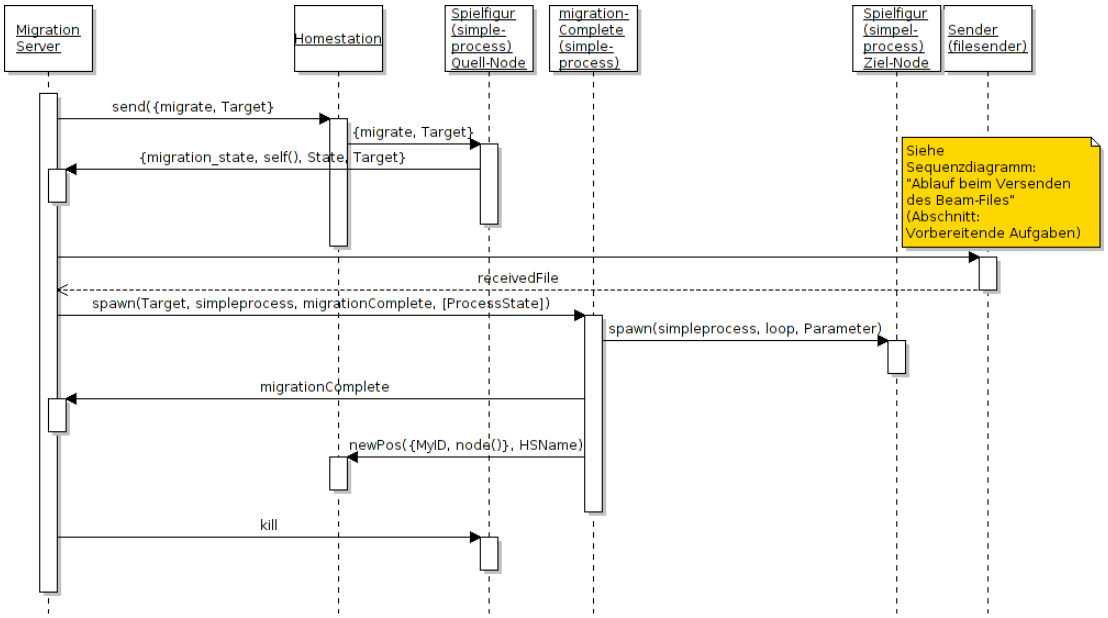

Abbildung 4.9: Ablauf der Migration mit einer Homestation

4.5 Zusammenfassung

Zusammenfassend kann gesagt werden, dass beide System ihre Vor- und Nachteile haben. Die Wahl des Systems ist daher hauptsächlich davon abhängig, wie viel Last auf das System zukommt und welche Aufgabe zu erfüllen ist.

Die Version mit einem Nameservice muss mindestens einen Prozess weniger (Homestation) betreuen und hat durch den Nameservice selbst eine zentrale Anlaufstelle, um Prozesse zu verwalten. Allerdings ist sie durch den Nameservice weniger flexibel, da sich dieser statisch auf einer Node befindet. Gegebenenfalls können dadurch lange Kommunikationswege entstehen.

Die Homestation-Version dagegen gibt die Möglichkeit noch flexibler in einem verteilten System zu agieren. Das liegt vor allem daran, dass jede Homestation auf einer anderen Maschine gestartet werden kann. Dadurch könnte schon bei der Planung eines großen Systems dafür gesorgt werden, die Kommunikationswege gering zu halten. Negativ zu bewerten ist dagegen der zusätzliche Aufwand, der entsteht, um später gestartete

Homestations in die Supervisorstruktur einzubinden. Durch die zusätzlichen Prozesse, in Form von Homestations, entsteht zudem eine gößere Fehleranfälligkeit.

5 Erweiterbarkeit

Dieser Abschnitt soll einen kurzen Einblick in mögliche Erweiterungen des bestehenden Systems geben.

Eine erste Idee, die sich in beiden Systemen implementieren ließe, wäre es, Erlangs *Hot Code Swapping* zu nutzen. Damit könnte man jeden hier beschriebenen Prozess, aber vor allem den migrierenden Prozess mit einer Update-Funktionalität ausstatten. Dadurch wäre es möglich im laufenden System die Funktionalität des migrierenden Prozesses zu ändern. Hieraus ergäbe sich ein großer Vorteil, betreffend des Einsatzgebiets des Systems.

Vorstellbar wäre ein Szenario, in dem ein solches System mit Update-Funktionalität genutzt wird, um Prozesse in unbekannte Umgebungen zu migrieren. Je nach Umgebung, auf die der Prozess trifft, könnte daraufhin Code (möglicherweise automatisiert) verändert werden, um sich der Umgebung anzupassen. Mit Hot Code Swapping ließe sich der neue Code unkompliziert im migrierten Prozess integrieren, ohne dass dieser gestoppt werden müsste.

Als Änderung des Homestation-Systems wäre es vorstellbar, dass eine Homestation nicht nur für einen einzelnen migrierenden Prozess zuständig ist. Dabei könnte es so implementiert werden, dass sich mehrere Prozesse an einer Homestation anmelden. Daraufhin würde man diese Prozesse gegebenenfalls aus einer Kombination von *Homestation-Name* und *Prozess-Name* ansprechen. Durch eine solche Erweiterung wäre es möglich, Prozesse mit ähnlichen Funktionen durch eine Homestation zu gruppieren. Zum Beispiel könnte eine Homestation existieren, die Prozesse zur Überwachung von Maschinen oder zur Durchführung von Berechnungen betreut. Eine solche Homestation könnte daraufhin gezielt im System platziert werden und ihre Prozesse an die Orte

schicken, an denen sie benötigt werden.

Als Erweiterung des Nameservice-Systems wäre eine Verteilung des Nameservice selbst denkbar. Dadurch könnte aus Optimierungsgründen dafür gesorgt werden, dass sich Prozesse in einer bestimmtem “Region” des Systems bei dem Nameservice melden, der am dichtesten gelegen ist. Dadurch würden wiederum die Kommunikationswege verringert. Eine solche Änderung würde zudem verhindern, dass ein einziger Nameservice zu einem Nadelöhr im System wird.

Es gibt sicher noch weitere Möglichkeiten das System zu erweitern. Die hier genannten können dabei als Basis dienen, auf denen weitere Ideen aufbauen.

6 Fazit

Das Ziel dieser Arbeit war es, mit Hilfe der Programmiersprache Erlang ein System zu entwickeln, das eine Prozessmigration zulässt. Diese Migration sollte von einer Node des Systems zu jeder beliebigen anderen Node des Systems möglich sein. Der Prozesszustand sollte dabei erhalten bleiben, sodass der Prozess nach einer Migration mit dem gleichen Zustand, den er auf der alten Node hatte, weiterlaufen kann.

Wie schon in der Einleitung zu dieser Arbeit beschrieben, ist Erlang für eine solche Aufgabenstellung sehr geeignet. Das liegt vor allem daran, dass die gesamte Sprache für die Arbeit in einem verteilten System ausgelegt ist. Erlang hat damit die Entwicklung des Systems stark unterstützt. Einen großen Anteil daran hat die Art und Weise, wie die Programmiersprache Nachrichten nutzt, um Prozesse kommunizieren zu lassen. Dadurch, sowie durch die einfache Handhabung der *receive*-Blöcke, konnten die einzelnen Komponenten, einschließlich der gesamten Kommunikation, gut geplant und anschließend umgesetzt werden.

Einzig das Problem der *globalen*-Registrierung von Prozessen auf zwei, nicht miteinander verbundenen Nodes, musste durch den Einsatz eines zusätzlichen Prozesses gelöst werden. Doch auch in diesem Fall war es möglich, die Lösung schnell und unkompliziert zu implementieren.

Abschließend ist also festzuhalten, dass das Ziel der Arbeit erreicht wurde. Ein funktionierendes Migrationssystem in der Programmiersprache Erlang konnte entwickelt werden und verfügt über mehrere Ansatzpunkte für Erweiterungen. Der Einsatz des Systems in einem *Scotland Yard Szenario* wäre dementsprechend mit dem hier vorliegenden

System gut zu erreichen. Aber auch für andere Anwendungsfälle, in denen Prozesse ihre Position verändern müssen, ist das entwickelte System nutzbar.

Literaturverzeichnis

[AB13a] Ericsson AB. Erlang Run-Time System Application (ERTS) Reference Manual Version 5.10.1, Modul: erlang. http://erlang.org/doc/man/erlang.html, 1997-2013. Accessed: 20-05-2013.

[AB13b] Ericsson AB. Erlang Run-Time System Application (ERTS) Reference Manual Version 5.10.1, Modul: global. http://erlang.org/doc/man/global.html, 1997-2013. Accessed: 20-05-2013.

[AB13c] Ericsson AB. Erlang Run-Time System Application (ERTS) Reference Manual Version 5.10.1, Modul: net_adm. http://erlang.org/doc/man/net_adm.html, 1997-2013. Accessed: 20-05-2013.

[AB13d] Ericsson AB. OTP Design Principles User's Guide Version 5.10.2. http://www.erlang.org/doc/design_principles/applications.html, 1997-2013. Accessed: 23-08-2013.

[AB13e] Ericsson AB. OTP Design Principles User's Guide Version 5.10.2, Behaviours. http://www.erlang.org/doc/design_principles/des_princ.html#id58199, 1997-2013. Accessed: 03-09-2013.

[AB13f] Ericsson AB. Pman Reference Manual Version 2.7.1.4. http://www.erlang.org/doc/man/pman.html, 1997-2013. Accessed: 26-09-2013.

[AB13g] Ericsson AB. STDLIB Reference Manual Version 1.19.2, Supervisor. http://www.erlang.org/doc/man/supervisor.html, 1997-2013. Accessed: 07-08-2013.

[AB13h] Ericsson AB. Erlang Run-Time System Application (ERTS) Reference Manual Version 5.10.2, Modules. http://www.erlang.org/doc/reference_manual/modules.html, 2003-2013. Accessed: 16-07-2013.

[AB13i] Ericsson AB. Erlang Run-Time System Application (ERTS) Reference Manual Version 5.10.2, Records. http://www.erlang.org/doc/reference_manual/records.html, 2003-2013. Accessed: 02-08-2013.

[arc09] http://stackoverflow.com/questions/243363/can-someone-explain-the-structure-of-a-pid-in-erlang, 2009. Accessed: 18-05-2013.

[Arm07] Joe Armstrong. A history of Erlang. *HOPL III Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[BD02] Tom Boyd and Partha Dasgupta. Process Migration: A Generalized Approach Using a Virtualizing Operating System. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS 02)*, Arizone State University, Tempe AZ, U.S.A., 2002.

[C.12] Giovanni C. http://stackoverflow.com/questions/13818587/erlang-send-file-and-filename, 2012. Accessed: 07-03-2013.

[Chr12] Hans Christian. An Erlang OTP tutorial for beginners. http://blog.bot.co.za/en/article/349/an-erlang-otp-tutorial-for-beginners, 2012. Accessed: 06-08-2013.

[Dä00] Bjarne Däcker. Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction. Master's thesis, Royal Institute of Technology, Stockholm, 2000.

[Doe10] Thomas W. Doeppner. *Operating Systems in Depth: Design and Programming*. Wiley, 2010.

[Dor09] Christoph Dornheim. Erlang Kompakt: Einfuhrung in die Programmierung mit Erlang. http://www.erlang-dach.org/download/Erlang-kompakt.pdf, 2009. Accessed: 06-05-2013.

[Heb10] Fred Hebert. Learn You Some Erlang. http://learnyousomeerlang.com, 2010. Accessed: 13-07-2013.

[Reb13] Rebar. Rebar - Wiki. https://github.com/basho/rebar/wiki, 2013. Accessed: 23-08-2013.

[Spi00] Ravensburger Spieleverlag. http://www.ravensburger.com/spielanleitungen/ecm/Spielanleitungen/Scotland_Yard_W_And_B_GB.pdf, 2000. Accessed: 17.09.2013.

[SPJ+91] Pradeep K. Sinha, Kyu Sung Park, Xiaohua Jia, Kentaro Shimizu, and Mamoru Maekawa. Process Migration in the GALAXY Distributed Operating System. In *Parallel Processing Symposium, 1991. Proceedings., Fifth International*, University of Tokyo, Tokyo, Japan, 1991.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28. Oktober 2013 Andreas Wimmer