

Ausarbeitung

„Erlang“

im Rahmen des Seminars „Programmiersprachen“

Eric Dombrowski

Themensteller: Prof. Dr. Herbert Kuchen
Betreuer: Prof. Dr. Herbert Kuchen
Institut für Wirtschaftsinformatik
Praktische Informatik in der Wirtschaft

Inhaltsverzeichnis

1	Idee und Entstehung	1
2	Grundlagen	2
2.1	Konzepte von Erlang	2
2.2	Programmieren mit Erlang	4
3	Erlang im Detail	6
3.1	Besonderheiten von Erlang.....	6
3.1.1	Nebenläufiges Programmieren	6
3.1.2	Verteiltes Programmieren.....	12
3.1.3	Real-Time	13
3.1.4	Codeersetzung	14
3.1.5	Robustheit.....	16
3.2	Softwareentwicklung mit Erlang.....	17
3.3	Quicksort	18
3.3.1	Implementierung.....	18
3.3.2	Vergleich	19
4	Fazit	20
	Quicksort in Java	21
	Literaturverzeichnis	22

1 Idee und Entstehung

Die folgenden Ausführungen orientieren sich überwiegend an [Ar96] und [Ar03].

Im Rahmen des Seminars „Programmiersprachen“ soll ein Überblick über verschiedene Programmiersprachen mit ihren jeweiligen Eigenschaften und Einsatzgebieten gegeben werden. Erlang gehört zur Kategorie der deklarativen Sprachen. Die Motivation für die Entwicklung von Erlang entstand bei einer Untersuchung, in wieweit sich Paradigmen moderner deklarativer Programmierung zur Erstellung großer industrieller Switching-Systeme eignen. Durch einen deklarativen Ansatz versprach man sich kürzeren Code und bessere Verifizierungsmöglichkeiten sowie eine generelle Vereinfachung der Entwicklung von Systemen im Telekommunikationsbereich. Da bei anfänglichen Tests keine von über zwanzig verschiedenen Programmiersprachen die vorgegebenen Problemstellungen vollständig abbilden konnte, wurde der Entschluss gefasst, mit der Neuentwicklung einer High-Level-Programmiersprache zu beginnen. Schnell wurde erkannt, dass sich die Anforderungen großer Telekommunikationssysteme stark mit denen allgemeiner industrieller Anwendungen überdecken. Im Ericsson Computer Science Laboratory (CSLab) begann die Entwicklung mit dem Ziel, eine einfache, effiziente und nicht zu umfangreiche Sprache zu schaffen, die sich gut zur Programmierung robuster, großer und nebenläufiger Anwendungen für den industriellen Einsatz eignet. In den vorherigen Tests mit Sprachen wie z.B. Prolog wurde die Wichtigkeit einer sicheren Fehlerbehandlung deutlich und man entschied sich für ein Ausführungsmodell ohne Backtracking. Details zu Prolog sind z.B. in [De96] zu finden. Angesichts des industriellen Einsatzes lag der Fokus bei der Abbildung von Nebenläufigkeit nicht in der Ausreizung aller durch Parallelisierung von Programmen theoretisch erzielbaren Performance-Vorteile, sondern auf der Möglichkeit, natürliche Nebenläufigkeit, wie man sie z.B. bei ereignisorientierten Anwendungen vorfindet, unkompliziert abbilden zu können. Nachdem bereits versucht worden war, Prolog um ein Prozessmodell zu erweitern, entstand 1987 eine erste in Prolog eingebettete Version von Erlang. Auf dieser basierten einige zu Testzwecken entwickelte Prototypen, die jedoch eine niedrige Performance aufwiesen. Daraufhin wurde Joe's Abstract Machine (JAM), eine virtuelle Stack-Maschine zur Ausführung von Erlang-Bytecode, und ein entsprechender Compiler entwickelt. JAM führte Erlang-Programme bis zu siebzigfach schneller als der zuvor eingesetzte Prolog-Interpreter aus (vgl. [Ar03, S. 5]). Informationen zu JAM sind in [Ar92] zu finden. Erlang wurde 1990 der Öffentlichkeit

vorgestellt und fortan in kleineren Projekten, wie z.B. einem Server zur Kontrolle mobiler DECT-Telefone, eingesetzt. Der nächste Schritt zu besserer Performance war Bogdan's Erlang Abstract Machine (BEAM), die hierfür erzeugten Code nach C übersetzt. Details zu BEAM sind in [Ha94] nachzulesen. Ericsson gründete 1993 die Tochtergesellschaft Erlang Systems AB zur Vermarktung von Erlang, zugehörigen Tools sowie Trainings- und Consulting-Angeboten. Bei der späteren Entwicklung des AXD301-Switching-Systems entstand die Open Telecom Platform (OTP), eine Weiterentwicklung der bisherigen Erlang-Bibliotheken. Im Jahre 1998 zog sich Ericsson aus der Entwicklung der Sprache zurück, nachdem das AXD301-Switching-System in über 1,7 Millionen Zeilen Erlang-Code fertig gestellt worden war (vgl. [Ar03, S. 18]). Dieses ist bis heute auf Ericssons Webseiten zu finden. Erlang wurde inklusive OTP unter Open-Source-Lizenz neu veröffentlicht und zum Mittelpunkt vieler Forschungsprojekte. Am Swedish Institute of Computer Science entstand beispielsweise ein Tool zur Unterstützung der Verifikation von Erlang-Programmen (vgl. [Da01]).

2 Grundlagen

2.1 Konzepte von Erlang

Grundsätzlich ist Erlang eine funktionale Programmiersprache mit deklarativer Semantik. Seiteneffekte sind trotz des funktionalen Ansatzes durch z.B. Nachrichtenversand möglich. Wie auch in anderen funktionalen Programmiersprachen, liegt der Fokus in Erlang auf Rekursionen, Listen und Tupeln. Arrays und Schleifen werden nicht direkt unterstützt. Funktionen können als First-Class-Werte zur Laufzeit erzeugt, als Parameter weitergereicht oder als Resultat zurückgegeben werden. Die Sprache hat, anders als beispielsweise Haskell (vgl. [Bi98]), ein schwaches Typsystem. Durch die dynamische Typüberprüfung können einerseits Programme schnell und unkompliziert geschrieben werden, andererseits werden jedoch eventuelle Typfehler nicht vom Compiler erkannt und treten zur Laufzeit auf. Bei der Definition von Funktionen oder Fallunterscheidungen wird auf Pattern-Matching zurückgegriffen, wobei immer die erste zu den Aufrufparametern passende Klausel ausgewählt wird. Das Konzept des Single-Assignment, wodurch der Wert einer Variablen nach der ersten Zuweisung nicht verändert werden kann, vereinfacht die Verifizierung von Programmen. Durch den Einsatz von Modulen lassen sich auch große Programme strukturieren und übersichtlich gestalten. Erlang berücksichtigt die explizite Abbildung

von Nebenläufigkeit in einem prozessbasierten Ansatz. Dabei werden Daten ausschließlich per asynchronen Nachrichtenversand ausgetauscht. Die Erzeugung und Verwaltung von Prozessen wurde im Vergleich zu anderen Sprachen wie z.B. C besonders effizient implementiert. Da Erlang-Prozesse stark isoliert sind und sich anders als bei üblichen Thread-Konzepten im Normalfall keine Ressourcen teilen können, lassen sich Einzelprozessoranwendungen verhältnismäßig leicht zu Multiprozessoranwendungen weiterentwickeln oder auf Netzwerken aus Einzelprozessoren ausführen. Es ist somit eine Basis für die Entwicklung verteilter Systeme geschaffen. Verteilte Systeme werden häufig in professionellen Anwendungen eingesetzt, um die Performance und die Ausfallsicherheit zu verbessern. Ein System kann beispielsweise auf mehrere sich gegenseitig überwachende Knoten verteilt werden. Weiterhin sind verteilte Systeme oft unkompliziert durch die Integration neuer Knoten zu erweitern und der wechselseitige Zugriff auf Ressourcen lässt sich anhand der zugrunde liegenden Knoten definieren und einschränken. Im Falle von geringen Abhängigkeiten zwischen enthaltenen Prozessen, sind verteilte Systeme häufig gut skalierbar und somit an z.B. durch die Umwelt vorgegebene Antwortzeiten anpassbar. Da in industriellen Steuerungsanlagen sowie Telekommunikationsanlagen schnell auf Ereignisse reagiert werden muss und gewisse Zeitrahmen nicht überschritten werden dürfen, ist eine besondere Integration der Zeit nötig, um wichtige Deadlines oder einzuhaltende Zeiträume definieren zu können. Erlang berücksichtigt direkt den Faktor Zeit und ermöglicht, z.B. über das Scheduling und die Definition von Timeouts, ein gut kontrollierbares Laufzeitverhalten. Der Einsatz von Erlang ist für Soft-Real-Time-Systeme spezifiziert, die Antwortzeiten im Bereich einer Millisekunde erfordern, wobei das genaue Verhalten abhängig von der zugrunde liegenden Soft- und Hardware-Plattform ist. Die Vorgaben müssen nur durchschnittlich eingehalten werden und können in Einzelfällen überschritten werden. Da große Telekommunikationssysteme in der Regel nicht für Updates oder Wartungen abgeschaltet werden können, erlaubt Erlang die Ersetzung von Code im laufenden Betrieb. Das Error Handling wurde auf nebenläufige und verteilte Anwendungen abgestimmt und bietet Möglichkeiten zur Überwachung von Prozessen und Netzwerkknoten. Für den Nachrichtenversand zwischen Netzwerkknoten sind Authentifizierungsverfahren vorgesehen. Weiterhin ist, wie in Java, ein für den Nutzer unsichtbares Memory-Management mit Garbage-Collector vorhanden, das die Gefahren beim Speicherzugriff minimiert. Physische Speicherzellen können nicht direkt ausgelesen oder manipuliert werden. Da sich die

Sprache Erlang durch Einschränkungen wie z.B. das Single-Assignment wenig für die effiziente Programmierung von Low-Level-Code anbietet, ist eine Integration für Programme anderer Sprachen vorgesehen. Bisherige Großprojekte mit Erlang wurden im Allgemeinen als eine Kombination aus verschiedenen Programmiersprachen realisiert. Die Anbindung an Systeme anderer Programmiersprachen wird in Erlang über Ports realisiert, die intern als besondere Prozesse ansprechbar sind. Erlang wird im Normalfall über eine für viele Plattformen emulierbare virtuelle Maschine ausgeführt und zeigt somit eine hohe Portabilität. Die umfangreiche Open Telecom Platform fungiert als Middleware zwischen Betriebssystem und Programmen in Erlang oder auch anderen Sprachen. Sie enthält neben dem Erlang-Run-Time-System, Basislösungen für übliche Anforderungen von Telekommunikationsanwendungen, wie z.B. Server- und Datenbankfunktionalitäten oder Schnittstellen und Kommunikationsprotokolle. Eine detaillierte Beschreibung der Open Telecom Platform befindet sich in [Ar03, Kap. 7].

2.2 Programmieren mit Erlang

Erlang verfügt über die üblichen Datentypen funktionaler Sprachen wie Zahlen, Atome, Tupel und Listen. Zahlen sind entweder Integers oder Floats und die zusammengesetzten Datentypen dürfen inhomogen sein, also aus verschiedenen Datentypen bestehen. Anonyme Funktionen werden über den Typ `fun` abgebildet. Records werden intern als Tupel repräsentiert und Strings basieren auf Listen aus Integer-Zahlen. Ein Datentyp Boolean ist nicht explizit definiert, die Atome `true` und `false` werden jedoch korrekt interpretiert. Neben dem Typ `Binary` zum Datenaustausch existieren zusätzlich Typen zur Unterstützung von nebenläufiger und verteilter Programmierung. Dabei bildet `Pid` Process-Identifizierer, `Port` Schnittstellen und `Reference` Referenzen ab. Es folgen einige Beispiele für Erlang-Datentypen.

```
List:      [atom1, 'Atom 2', {0,54}, {"String",5.678e9}]
Pid:       <0.37.0>
Reference: #Ref<0.0.0.108>
```

Erlang bietet die üblichen arithmetischen Operatoren sowie Vergleichsoperatoren. Diese werden ausführlich in [Ar96, S. 30 und 34] beschrieben. Zur Verdeutlichung des Aufbaus von Erlang-Modulen folgt das Modul `bsp0`.

```
-module (bsp0) .
-export ([start/0]) .
start() ->
    "Hello World!!!".
```

Module definieren einen eigenen Namensraum. Am Anfang jedes Moduls werden der Modul-Name und die öffentlichen, also extern sichtbaren Funktionen inklusive der Anzahl ihrer Parameter angegeben. Der Kopf einer Funktion besteht aus dem Funktionsnamen, der mit einem Kleinbuchstaben beginnen muss, und den Aufrufparametern. Nach dem Symbol „->“ folgt der Function-Body, in dem Erlang-Ausdrücke durch Kommata getrennt werden. Dieser wird durch einen Punkt beendet. Als Resultat wird das Ergebnis des letzten Ausdrucks zurückgegeben. Variablen muss bei der Deklaration ein Wert zugewiesen werden, wobei der Typ automatisch festgelegt wird und der Variablenname nur mit einem Großbuchstaben beginnen darf. Funktionen in Modulen werden durch `modul:funktion()` angesprochen. Der Aufruf `bsp0:start()` führt somit im obigen Beispiel zum Resultat "Hello World!!!". Wie in funktionalen Programmiersprachen üblich, gestattet Erlang das Überladen von Funktionsbezeichnern und Pattern-Matching als Auswahlkriterium. Dies soll am folgenden Beispiel veranschaulicht werden.

```
search([]) ->
    no_int;
search([Head|Tail]) when integer(Head) ->
    Head;
search(List) when list(List) ->
    search(tl(List));
search(Other) ->
    badvalue.
```

Die Funktion `search/1` gibt die erste Integer-Zahl einer Liste zurück. Hierzu werden beim Aufruf die vier durch Semikolon getrennten Klauseln mit dem übergebenen Parameter verglichen. Der Function-Body der ersten passenden Klausel wird ausgewählt. Erfüllt der Parameter keine Klausel, resultiert eine Fehlermeldung. Wurde eine leere Liste übergeben, wird demnach `no_int` zurückgegeben. Ist die Liste nicht leer, wird der Listenkopf, wenn dieser eine Integer-Zahl ist, zurückgegeben oder andernfalls, wenn dieser keine Integer-Zahl ist, die Funktion `search/1` rekursiv mit dem Listenrumpf aufgerufen. In jedem anderen Fall resultiert `badvalue`. Auf diese Weise wird eine beim Funktionsaufruf übergebene Liste rekursiv bis zur ersten Integer-Zahl durchlaufen. Die Typtests auf `Integer` bzw. `List` werden im obigen Beispiel in so genannten Guards vollzogen, die im Kopf der Klausel stehen. Eine Klausel kann nur erfüllt werden, wenn alle enthaltenen Guards wahr sind. Guards werden durch `when`, gefolgt von einem Boolean-Ausdruck, definiert. Hierfür stehen z.B. die üblichen Vergleichsoperatoren sowie einige Funktionen für Typtests bereit. Diese sind in [Ar96,

Kap. 2.5.5] nachzulesen. Im Function-Body werden Bedingungen und Fallunterscheidungen über die Primitive `if` und `case` eingebunden. Ist der Wert einer Variablen nicht von Bedeutung, kann das Symbol „_“ als Platzhalter eingesetzt werden. Dies gilt z.B. auch für die Zuweisung von Variablen.

```
Adresse = {"Leonardo-Campus", 3, "Münster"},  
{_, Hausnummer, _} = Adresse.
```

Die Variable `Hausnummer` hat nach den obigen Ausdrücken den Wert 3.

3 Erlang im Detail

3.1 Besonderheiten von Erlang

3.1.1 Nebenläufiges Programmieren

Die Programmiersprache Erlang unterstützt nebenläufiges Programmieren direkt durch Prozesse. Diese werden unabhängig voneinander ausgeführt, wobei alle bei ihrer Erzeugung dieselbe Priorität haben und wie normale Datenobjekte in Erlang zu verwenden sind. Ein Prozess basiert auf einer Funktion, deren Name und Modul zur Erzeugung übergeben werden, und terminiert mit dem Ende dieser Funktion. Jeder Prozess hat einen eindeutigen und unveränderlichen Process-Identifizier. Die Funktion `spawn(Modul, Funktion, Parameterliste)` erzeugt einen Prozess auf Basis von `Funktion` und liefert, ohne die Ausführung der übergebenen Funktion abzuwarten, den Process-Identifizier als Resultat. Prozesse werden in Erlang grundsätzlich hierarchisch gruppiert. Der erste erzeugte Prozess ist der Group-Leader und jeder weitere hieraus erzeugte Prozess wird der Gruppe des aufrufenden Prozesses in der nächst tieferen Hierarchieebene zugeordnet. Auf diese Weise entstehen Bäume mit dem ersten Prozess einer Gruppe als Wurzel. Ist der Identifizier eines Prozesses bekannt, können diesem Prozess Nachrichten geschickt werden. Soll der Nachrichtenaustausch auch ohne Kenntnis des Process-Identifiziers möglich sein, können Prozesse mit der Funktion `register(Name, Pid)` unter einem eindeutigen Namen registriert werden. Terminiert ein Prozess, kann z.B. ein anderer Prozess unter gleichem Namen registriert werden, um diesen zu ersetzen und die Kommunikation aufrecht zu erhalten. Die Kommunikation zwischen Prozessen erfolgt in Erlang per asynchronem Nachrichtenaustausch. Die Prozesse werden somit nicht bis zur Fertigstellung des Nachrichtenversands unterbrochen. Bei synchronem Nachrichtenaustausch könnte der

Versender durch Fehler im Empfängerprozess blockiert werden. Prozesse erhalten keine Informationen, ob von ihnen versandte Nachrichten zugestellt werden konnten. Mit dem Symbol „!“ kann wie folgt eine Nachricht an einen Prozess mit bekanntem Process-Identifizier bzw. registriertem Namen verschickt werden.

```
PIDoderName ! Nachricht
```

Ist der Empfängerprozess im System aktiv, wird die Nachricht in dessen Postfach abgelegt. Jeder Prozess besitzt ein solches Postfach, in dem Nachrichten in der Reihenfolge ihrer Ankunft angeordnet sind. Besonders anzumerken ist, dass Nachrichten in Erlang keinem vorgegebenen Aufbau unterliegen. Ist der Identifizier des Absenders nicht in der Nachricht enthalten, kann auch nicht nachvollzogen werden, welcher Prozess diese gesendet hat. Nachrichten können beliebig aus Erlang-Typen aufgebaut werden und somit leicht an verschiedene Protokolle angepasst werden. Der in Erlang implementierte Empfangsmechanismus ist selektiv und schützt die Prozesse vor unerwarteten Nachrichten. Die Verantwortung, das Postfach rechtzeitig zu leeren, um darin effizient suchen zu können, liegt beim Programmierer. Erwartet ein Prozess eine Nachricht, prüft er sein Postfach mit einem `receive`-Ausdruck. Dabei werden die Nachrichten in der Reihenfolge ihrer Ankunft anhand Pattern-Matching überprüft. Gibt es Nachrichten im Postfach des Prozesses, auf die eine Klausel von `receive` zutrifft, wird die erste passende Nachricht entfernt und der zugehörige Code des `receive`-Ausdrucks ausgeführt. Es resultiert, wie auch bei Funktionen, das Ergebnis des letzten Ausdrucks. Gibt es keine passende Nachricht, wird der aufrufende Prozess bis zum Eintreffen neuer Nachrichten suspendiert. Bei neu eintreffenden Nachrichten wird der Prozess reaktiviert, um erneut wie oben beschrieben zu verfahren. Im folgenden Beispiel sind nacheinander die Nachrichten `nachricht1`, `nachricht2` und `nachricht3` im Postfach eines Prozesses eingegangen. Nun werden beispielhaft einige Nachrichten per `receive` aus dem Postfach entfernt. Im ersten `receive`-Ausdruck erfüllt `nachricht1` keine Klausel und es wird die zweite Nachricht geprüft. Diese wird aus dem Postfach entfernt, weil sie die zweite Klausel erfüllt. Es resultiert `reaktion2`.

```
receive
  nachricht3 -> reaktion3;
  nachricht2 -> reaktion2;
end.
```

Postfach:		
nachricht1	->	nachricht1
nachricht2		nachricht3
nachricht3		

Der nächste `receive`-Ausdruck entfernt `nachricht1` aus dem Postfach und es resultiert `sonst`.

```

receive
    nachricht3 -> reaktion3;
    Sonst -> sonst;
end.

```

Postfach:

nachricht1	->	nachricht3
nachricht3		

Gerade in Telekommunikationssystemen kommt häufig das Client-Server-Konzept zum Einsatz. Die Möglichkeit dieses unkompliziert in Erlang umzusetzen, soll anhand des folgenden Beispiels veranschaulicht werden. Clients können sich am Server an- und abmelden sowie Nachrichten über diesen an alle angemeldeten Clients senden. Weiterhin werden die über den Server gesendeten Nachrichten paketweise an einen Überwachungsprozess weitergeleitet.

```

-module (bsp1).
-export ([start/0, observerloop/0, serverloop/2, login/0,
          logout/0, exit/0, send/1]).

start() ->
    register(server, spawn(bsp1, serverloop, [[], []])),
    register(observer, spawn(bsp1, observerloop, [])).

%%Observer
observerloop() ->
    receive
        {From, Message} ->
            io:format("~w", [Message]),
            observerloop();
        {From, exit} ->
            exited
    end.

%%Server
serverloop(Receivers, History) when length(History) >= 5 ->
    observer ! {server, History},
    serverloop(Receivers, []);

serverloop(Receivers, History) ->
    receive
        {From, login} ->
            From ! {server, {From, logged_in}},
            serverloop([From|Receivers], History);
        {From, logout} ->
            From ! {server, {From, logged_out}},
            serverloop(lists:delete(From, Receivers),
                History);
        {From, exit} ->
            send(From, server_exit, Receivers, History);
        {From, {send, Message}} ->
            NewHistory = send(From, Message, Receivers,
                History),
            serverloop(Receivers, NewHistory);
        Other ->
            serverloop(Receivers, History)
    end.

```

```
send(From, Message, [], History) ->
    From ! {server,ok},
    [Message|History];

send(From, Message, [Receiver|Tail], History) ->
    case (From /= Receiver) of
        true ->
            Receiver ! {From,Message},
            send(From,Message,Tail,History);
        false ->
            send(From,Message,Tail,History)
    end.

%%Interface-Funktionen
login() ->
    request(login).
logout() ->
    request(logout).
exit() ->
    request(exit).
send(Message) ->
    request({send,Message}).
request(Request) ->
    server ! {self(), Request},
    receive
        {server, Response} ->
            Response
    after 10000 ->
        failed
    end.
```

Das gesamte Beispiel ist vereinfachend in einem einzigen Modul namens `bsp1` dargestellt und verfügt über keinerlei Fehlerbehandlung. Durch Aufruf der Funktion `start/0` werden Server- und Überwachungsprozess erzeugt und als `server` bzw. `observer` registriert. Die Prozesse basieren jeweils auf rekursiven Funktionen, die auf Nachrichten reagieren. Der auf der Funktion `observerloop/0` basierende Prozess `observer` gibt erhaltene Nachrichten aus und terminiert bei Nachrichten mit dem Schlüsselwort `exit`. Der Prozess `server` auf Basis der Funktion `serverloop/2` pflegt `Receivers`, die Liste der angemeldeten Prozesse, sowie `History`, die Liste der letzten 5 über den Server versandten Nachrichten. Akzeptierte Nachrichten sind Tupel aus dem Absender und `login`, `logout`, `exit`, oder `{send, _}` für die auszuführende Server-Funktion. Nachrichten, die nicht diesem Format entsprechen, werden über die Klausel `Other` aus dem Postfach entfernt und ignoriert. Nach der Bearbeitung einer Nachricht bleibt der Prozess `server`, außer beim Schlüsselwort `exit`, durch einen rekursiven Aufruf aktiv. Die Funktion `serverloop/2` liegt in zwei Varianten vor. Enthält die Liste `History` mindestens 5 Elemente, wird diese an den

Überwachungsprozess `observer` geschickt und im rekursiven Aufruf durch die leere Liste ersetzt. In allen anderen Fällen verarbeitet `serverloop/2`, wie zuvor beschrieben, Nachrichten an den ausführenden Prozess. Um die interne Kommunikationsstruktur des Servers, sowie dessen registrierten Namen geheim zu halten und die Übergabe falscher Process-Identifizierer zu vermeiden, wurden Interface-Funktionen definiert. Diese basieren alle auf der Funktion `request/1`, die Nachrichten aus dem mit der Funktion `self/0` ermittelten Identifizierer des aufrufenden Prozesses und dem als Parameter übergebenen Schlüsselwort an den Server-Prozess sendet. Das Resultat der Funktion ist die Antwort des Server-Prozesses oder `failed` falls der Server nicht innerhalb von zehn Sekunden reagiert. Auf die Verwendung von Timeouts werde ich später weiter eingehen. Da jede Interface-Funktion mit dem Aufruf von `request/1` endet, erhält der aufrufende Prozess die Antwort von `server` direkt als Resultat der Interface-Funktion und ist somit bis zur Reaktion des Server-Prozesses unterbrochen. Führt beispielsweise ein Prozess mit dem Identifizierer `<0.37.0>` den Code `bsp1:login()` aus, wird die Nachricht `{<0.37.0>,login}`, wie in Abbildung 1 dargestellt, an `server` gesendet. Dieser antwortet mit `{server,{<0.37.0>,logged_in}}` und aktualisiert `Receivers` bei einem rekursiven Aufruf von `serverloop/2` zu `[<0.37.0>|Receivers]`. Für den aufrufenden Prozess ist nur `{<0.37.0>,logged_in}` als Resultat von `login/0` sichtbar.

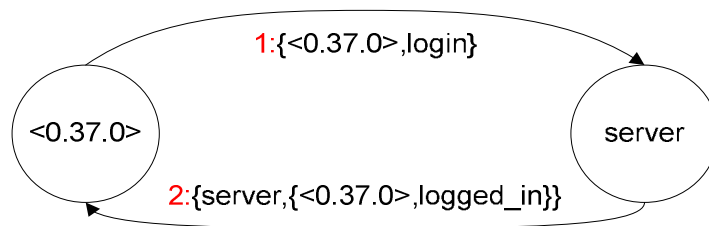


Abbildung 1: Nachrichtenaustausch bei der Anmeldung am Server (Reihenfolge in rot)

Über die Funktion `send(From,Message,Receivers,History)` wird es möglich, Nachrichten an alle mit `server` verbundenen Prozesse zu versenden, ohne diese zu kennen. Es sind wiederum zwei Varianten dieser Funktion zu unterscheiden. Ist die Liste `Receivers` leer, erhält der aufrufende Prozess eine Bestätigung und es resultiert die um `Message` erweiterte Liste `History`. Andernfalls wird das erste Element von `Receivers` entfernt und die Nachricht `{From, Message}` an den enthaltenen Identifizierer geschickt, sofern dieser nicht dem Absender entspricht. Es folgt ein rekursiver

Aufruf. Die Funktion `send/4` verdeutlicht die Flexibilität beim Nachrichtenaustausch in Erlang. Da Nachrichten beispielsweise keine vom Erlang-System generierten Absenderinformationen enthalten, ist für den empfangenden Prozess nicht ersichtlich, dass die Nachrichten über den Server anstatt direkt vom Absender verschickt wurden. In Abbildung 2 ist ein Nachrichtenversand über den Server grafisch dargestellt. Es sind vier Prozesse angemeldet, `Receivers` ist die Liste [`<0.37.0>`, `<0.35.0>`, `<0.33.0>`, `<0.31.0>`] und der Prozess mit dem Identifier `<0.37.0>` ruft `bsp1:send("Testnachricht")` auf.

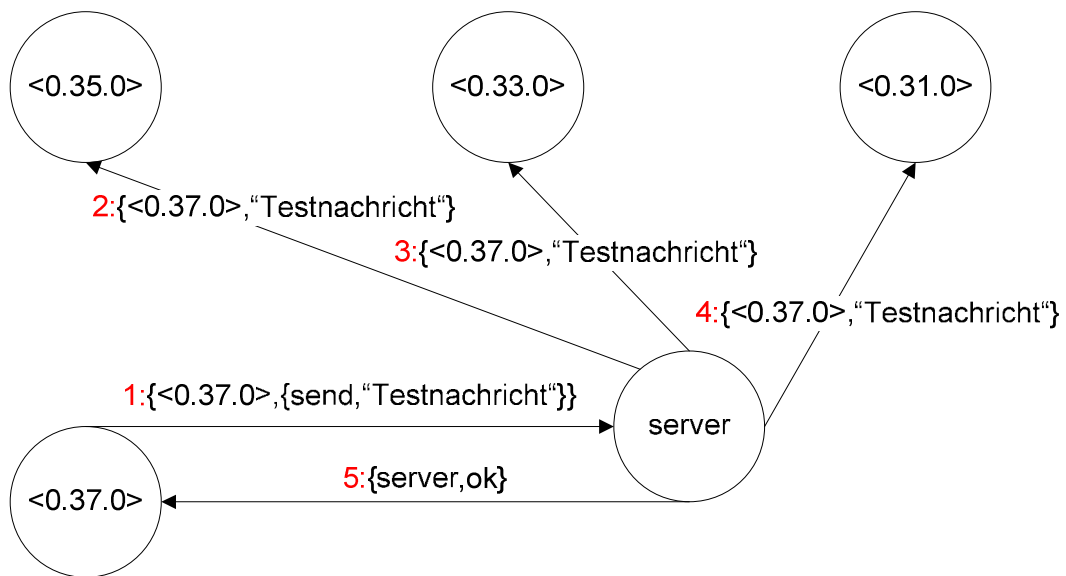


Abbildung 2: Nachrichtenaustausch beim Aufruf von `send/1` (Reihenfolge in rot)

In industriellen Steuerungs- und Kontrollanlagen sowie professionellen Telekommunikationsanlagen dürfen nebenläufige Prozesse sich nicht gegenseitig blockieren oder durch unendliches Warten auf Ereignisse ausfallen. Zusätzlich sollte es möglich sein, einzuhaltende Zeitrahmen oder maximale Wartezeiten auf z.B. Nachrichten abzubilden. Um solchen Anforderungen gerecht zu werden, können in Erlang `Timeouts` in einen `receive`-Ausdruck integriert werden.

```

receive
...
after Zeit ->
           reaktion
end
  
```

Die Zeit wird in Millisekunden als Integer-Zahl angegeben, wobei die genaue Einhaltung nicht garantiert werden kann, da diese durch die Auslastung und das Scheduling des zugrunde liegenden Systems beeinflusst wird. Hierbei sind zwei

besondere Werte aufzuführen. Hat `Zeit` im obigen Beispiel den Wert 0, resultiert `reaktion`, nachdem alle im Postfach befindlichen Nachrichten auf die Klauseln des `receive`-Ausdrucks geprüft wurden und keine Nachricht ausgewählt wurde. Auf diese Weise ist es möglich, bestimmte Nachrichten zu bevorzugen, indem man diese im `receive`-Ausdruck abrufen und weniger wichtige Nachrichten durch einen neuen `receive`-Ausdruck nach einem Timeout von 0 abfragt.

```
receive
    wichtig ->
        reaktion1
after 0 ->
    receive
        AndereNachrichten ->
            reaktion2
    end
end
```

Die normale First-Come-First-Served-Auswahl beim Prüfen auf bestimmte Nachrichten wird somit umgangen. Der zweite besondere Wert ist `infinity`. Dieser hat kein Timeout zur Folge. Werden die Zeiten für Timeouts erst zur Laufzeit berechnet, können somit im Code gesetzte Timeouts aufgehoben werden.

3.1.2 Verteiltes Programmieren

Erlang unterstützt direkt den Aufbau von Netzwerken mit einzelnen Erlang-Systemen als Knoten. Nach [Ar96, S. 88] sind diese Knoten untereinander lose gekoppelt und können das Netzwerk wie Prozesse dynamisch betreten oder verlassen. Ein Knoten wird im Netzwerk sichtbar, sobald er einen Prozess namens `net_kernel` ausführt, der die Funktion `alive(Name, Port)` aufruft. Dabei geht der aktuelle Knoten in den Zustand `alive`, wird unter `Name` beim Name-Server des Netzwerks angemeldet und ist fortan über `Name@Rechnername` ansprechbar. Der Erlang-Port `Port` stellt die Netzwerkschnittstelle dar, wobei die Netzwerkfunktionalität extern bereitgestellt werden muss. Das genaue Vorgehen wird detailliert in [Ar96, Kap. 9.7] beschrieben. Der Prozess `net_kernel` wird im Folgenden über Zugänge und Abgänge von Knoten im Netzwerk benachrichtigt und kontrolliert den Nachrichtenversand an andere Knoten sowie die Erzeugung von Prozessen auf dem eigenen Knoten. Beispiele solcher Nachrichten an den `net_kernel` können in [Ar96, S. 135] nachgelesen werden. Jedem Erlang-Knoten wird bei seiner Erzeugung das so genannte Magic-Cookie zugewiesen. Eine Authentifizierung beim Nachrichtenaustausch zwischen Knoten wird durch

Integration der Magic-Cookies in Nachrichten unterstützt. Nachrichten werden bei korrektem Magic-Cookie direkt zugestellt und bei falschem Magic-Cookie als `{From, badcookie, To, Nachricht}` an den Prozess `net_kernel` umgeleitet, um entsprechende Reaktionen zu ermöglichen. Weiterhin können zusammengehörige Knoten mit gleichen Magic-Cookies initialisiert werden, um zwischen diesen die Kommunikation zu vereinfachen. Ebenso könnten z.B. alle innerhalb einer Nutzer-Session erzeugten Knoten mit einem gemeinsamen Magic-Cookie initialisiert werden, das zu Anfang der Session eingelesen wird. Weitere Informationen zu Magic-Cookies sind in [Ar96, Kap. 9.7.1] zu finden. Zusätzlich bieten sich zur sicheren Authentifizierung die so genannten Referenzen an. Diese global eindeutigen Schlüssel können nur verglichen und nicht manipuliert werden. Eine Referenz wird mit der Funktion `make_ref()` erzeugt. Erlang unterstützt den Nutzer bei der Verwaltung von Knoten mit weiteren eingebauten Funktionen und pflegt eine Liste bekannter Knoten. Der Nachrichtenversand erfolgt auch in verteilten Systemen über den Process-Identifizier bzw. den registrierten Namen des Prozesses und den Namen des Zielknotens.

```
Pid ! Nachricht      bzw.      {Name, Knoten} ! Nachricht
```

Prozesse können bei zusätzlicher Angabe des Zielknotens auch auf fremden Knoten erzeugt werden. Mit den in Erlang gebotenen Möglichkeiten kann das obige Client-Server-Beispiel ohne großen Aufwand zu einer verteilten Anwendung erweitert werden. Durch leichte Veränderung der Funktion `request/1` funktioniert der Zugriff auf den Server als entfernter Aufruf, wenn Client-Interface und Server auf unterschiedliche Knoten verteilt sind.

3.1.3 Real-Time

Telekommunikationssysteme sowie Systeme zur Steuerung industrieller Anlagen sind meist Real-Time-Systeme, was bedeutet, dass diese Systeme schnell auf Ereignisse reagieren müssen. Da es im professionellen Einsatz teure Auswirkungen haben kann, wenn sich Prozesse gegenseitig blockieren oder diese willkürlich suspendiert werden, sollte dem Scheduling in diesem Bereich besondere Aufmerksamkeit gewidmet werden. Bei Erlang-Systemen wird nach [Ar96, S. 83] unabhängig von der Implementation ein faires Scheduling vorausgesetzt. Das bedeutet, dass jeder wartende Prozess auch ausgeführt wird, wobei möglichst die ursprüngliche Reihenfolge, in der die Prozesse aufgetreten sind, erhalten bleiben sollte. Kein Prozess darf das System für längere Zeit

blockieren. Um eine gleichmäßige Aufteilung der CPU-Zeit zu gewährleisten, wird jedem Prozess ein kurzes Intervall, das so genannte Time-Slice, für die Ausführung zugeteilt. Nach dessen Ablauf oder beim Warten auf eine Nachricht, wird der aktive Prozess unterbrochen und je nach eingesetztem Scheduling-Algorithmus der nächste zur Ausführung bereitstehende Prozess ausgewählt. Die genaue Umsetzung des Scheduling ist nicht vorgeschrieben, soweit die obigen Auflagen eingehalten werden und die durchschnittliche Antwortzeit des gesamten Systems im Bereich einer Millisekunde liegt. Das Scheduling richtet sich in Erlang-Systemen normalerweise nach abgearbeiteten Reduktionen eines Prozesses anstatt echter CPU-Zeit. Das für den Programmierer unsichtbare automatische Speichermanagement wurde ebenfalls mit dem Ziel einer kurzen Antwortzeit des Gesamtsystems konzipiert. Dieses darf das System nicht blockieren und sollte in kleineren als den Prozessen zur Verfügung stehenden Time-Slices im Hintergrund ausgeführt werden, um unerwartete Reaktionen des Systems sowie starke Schwankungen der Antwortzeit zu vermeiden. Da sich Prozesse im Allgemeinen in Wichtigkeit und Rechenaufwand unterscheiden, erlaubt Erlang die Vergabe von Prioritäten für Prozesse zur Manipulation der Häufigkeit ihrer Auswahl beim Scheduling. Bei der Erzeugung haben alle Prozesse die Priorität `normal` und werden ähnlich häufig ausgeführt. In Erlang stehen üblicher Weise die Prioritätsstufen `low`, `normal` und `high` zur Verfügung. Beispielsweise könnte einem Monitor-Prozess der Wert `low` genügen, wohingegen eine aufwendige zeitkritische Berechnung einen Prioritätswert `high` erfordern würde, um innerhalb gewisser Zeitschranken ein Resultat zu liefern. Weiterhin lässt sich der zeitliche Ablauf in Erlang-Systemen grob über die bereits angesprochenen Timeouts beeinflussen.

3.1.4 Codeersetzung

Eine Besonderheit, die Erlang für den Einsatz in Telekommunikationssystemen empfiehlt, ist die Code-Ersetzung. Entsprechende Systeme erfordern häufig eine ununterbrochene Ausführung und können nicht für Updates heruntergefahren und neu gestartet werden. Wird beispielsweise ein Switch heruntergefahren, ist die Netzwerkkommunikation unterbrochen und bereits gepufferte Daten können verloren gehen. Erlang erlaubt es, Code im laufenden Betrieb zu ersetzen, was die gleichzeitige Ausführung von Prozessen mit neuem neben Prozessen mit altem Code zur Folge haben kann. Nach [Ar03, S. 80] dürfen je Modul höchstens zwei verschiedene Code-Versionen im Erlang-System ausgeführt werden. Der Aufruf einer Funktion entscheidet

darüber, ob bei Rekursionen jeweils der aktuellste oder der seit dem ersten Aufruf bekannte Code ausgeführt wird. Soll der Code immer aktuell sein, muss der Funktionsaufruf als `modul:funktion()` formuliert sein. Bei einem Aufruf ohne die Angabe des Moduls bleiben Code-Updates während der Laufzeit bei rekursiven Aufrufen unberücksichtigt. Zur Verdeutlichung folgt das Beispielprogramm 9.1 aus [Ar96, Kap. 9.1.3] in leicht erweiterter Form.

```
-module(code_replace).
-export([test/0, loop_update/1, loop_normal/1, reception/1]).

test() ->
    register(update_process, spawn(code_replace, loop_update, [0])),
    register(normal_process, spawn(code_replace, loop_normal, [0])).

loop_update(N) ->
    %%Code-Updates werden übernommen
    reception(N),
    code_replace:loop_update(N+1).

loop_normal(N) ->
    %%Code ändert sich nicht bei Updates
    reception(N),
    loop_normal(N+1).

reception(N) ->
    receive
    X ->
        io:format('N = ~w Version A received ~w~n', [N, X])
    end.
```

Nach dem Kompilieren des Moduls `code_replace` und dem Aufruf von `code_replace:test()` warten die Prozesse `update_process` und `normal_process` auf Nachrichten. Der Zähler `N` erhöht sich mit jeder empfangenen Nachricht um 1 und wurde mit 0 initialisiert. Die Prozesse erhalten zunächst einige Nachrichten, wie hier in der Erlang-Shell der Distribution R9B-0 dargestellt.

```
4> normal_process ! nachricht1.
N = 0 Version A received nachricht1
nachricht1
5> update_process ! nachricht2.
N = 0 Version A received nachricht2
Nachricht2
...
9> update_process ! nachricht6.
N = 2 Version A received nachricht6
Nachricht6
10> normal_process ! nachricht7.
N = 3 Version A received nachricht7
Nachricht7
```

Wird nun die Ausgabe der Funktion `reception/1` auf `Version B` verändert, zeigt sich nach dem Kompilieren das folgende Verhalten beim Empfang neuer Nachrichten.

```
12> normal_process ! nachricht8.  
N = 4 Version A received nachricht8  
Nachricht8  
13> update_process ! nachricht9.  
N = 3 Version A received nachricht9  
Nachricht9  
14> normal_process ! nachricht10.  
N = 5 Version A received nachricht10  
Nachricht10  
15> update_process ! nachricht11.  
N = 4 Version B received nachricht11  
Nachricht11
```

Der rekursive Aufruf zum Code-Update wurde durch `nachricht9` ausgelöst und die neue Ausgabe nach `nachricht11` sichtbar. Demnach hat `update_process` den neuen Code übernommen, wobei der Wert des Zählers erhalten blieb.

3.1.5 Robustheit

Um Anwendungen robust genug für das angestrebte Einsatzgebiet zu gestalten, bietet Erlang über übliche Konzepte, wie z.B. „try“, „catch“ und „throw“, hinausgehende Ansätze zur Behandlung von Fehlern. Prozesse können untereinander verlinkt werden und sich gegenseitig überwachen. Links sind generell bidirektional und werden z.B. mit der Funktion `link/1` erzeugt. Terminiert ein Prozess, sendet er allen verlinkten Prozessen eine Exit-Nachricht mit seinem Identifier und dem Grund für den Abbruch.

```
{'EXIT', Pid, Reason}
```

Verlinkten Prozessen ist es somit möglich zu reagieren, wenn ein Prozess terminiert. Wird das Standard-Verhalten bei Exit-Nachrichten nicht überschrieben, terminieren alle verlinkten Prozesse ebenfalls, wenn sie eine Exit-Nachricht mit `Reason` ungleich `normal` erhalten. Weiterhin ermöglicht die Funktion `monitor/2` eine Überwachung von Prozessen. Der aufrufende Prozess wird mit einer Nachricht über wichtige Zustandsänderungen des beobachteten Prozesses informiert. Das Monitoring ist nicht bidirektional und der aufrufende Prozess terminiert nicht, wenn Prozesse, die er beobachtet abnormal terminieren. Dies ist z.B. wünschenswert, wenn ein Server seine Clients überwachen soll. Netzwerkknoten werden über den `net_kernel` oder durch Aufruf der Funktion `monitor_node/2` überwacht. Die Verbindung zum entsprechenden Knoten wird automatisch aufgebaut. Wird der Knoten nicht gefunden,

dieser beendet oder die Netzwerkverbindung abgebrochen, erhält der ausführende Prozess die Nachricht `{nodedown, Node}`. Es ist jedoch nicht möglich, Fehler im Netzwerk zwischen überwachtem Knoten und Monitor-Prozess von Fehlern beim überwachten Knoten selbst zu unterscheiden. Mit den aufgeführten Möglichkeiten können unkompliziert Überwachungsprozesse geschaffen werden, die den Zustand anderer Prozesse kontrollieren und diese gegebenenfalls neu starten.

3.2 Softwareentwicklung mit Erlang

Als deklarative Sprache eignet sich Erlang nicht für Low-Level-Code. Durch Einschränkungen wie z.B. das Single-Assignment bei Variablen oder das Fehlen von Schleifen, kann zwar kürzerer und leichter zu verifizierender Code erzeugt werden, es lassen sich jedoch viele übliche Problemstellungen nicht effizient realisieren. Erlang ermöglicht daher über Ports und entsprechende Interfaces die Entwicklung von Systemen in Verbindung mit anderen Programmiersprachen. In bisherigen Praxisprojekten war es üblich, Erlang nur in gut unterstützten Bereichen, wie z.B. der Abbildung von Client-Server-Anwendungen, einzusetzen und das restliche System in anderen geeigneten Sprachen zu entwickeln. Beim Programmieren mit Erlang sollte Rekursionen besondere Aufmerksamkeit gewidmet werden. Hier kann viel Speicher durch die Verknüpfung rekursiver Aufrufe verschwendet werden, weil für deren Abarbeitung erst das Ergebnis der rekursiven Aufrufe abgewartet werden muss. Es werden folglich viele Informationen über noch auszuführende Operationen vorgehalten. Daher empfiehlt es sich, rekursive Funktionen durch so genannte Tail-Recursion in konstantem Speicher auszuführen. Dabei sollte, wie in [Ar96, Kap. 9.1.1] beschrieben, der letzte Ausdruck jeder Klausel der Funktion entweder eine Konstante oder ein einzelner erneuter Aufruf dieser Funktion ohne weitere äußere Verknüpfungen sein. Die Ausführung in konstantem Speicher verleiht Systemen mehr Stabilität und ist z.B. in Server-Anwendungen besonders vorteilhaft. Weiterhin stehen besondere Möglichkeiten beim Debugging in verteilten und auf Prozessen basierenden Anwendungen zur Verfügung. Es sind z.B. in der Erlang-Toolbar Debugging-Tools vorhanden, die explizit die Prozess- und Knotenstruktur berücksichtigen. Bei Sprachen, die nachträglich beispielsweise durch zusätzliche Bibliotheken um ein Prozesskonzept erweitert wurden, wird dies im Allgemeinen nicht direkt unterstützt. Eine mögliche Fehlerquelle stellen Seiteneffekte dar, die z.B. durch den Nachrichtenaustausch, die Erzeugung von

Prozessen oder Prozess-Dictionaries möglich sind. In Process-Dictionaries können auf Prozessebene globale Variablen abgelegt und manipuliert werden. Weiterhin sehe ich das umgesetzte Code-Replacement als gefährlich an. Dieses ist zwar in der Anwendung sehr komfortabel, kann jedoch bei grundsätzlichen Änderungen am Code durch die gleichzeitige Ausführung von Prozessen mit neuem neben Prozessen mit altem Code zu Problemen führen.

3.3 Quicksort

3.3.1 Implementierung

Da Quicksort in [Ar96, Kap. 3.3.1] bereits mit Erlang implementiert wurde, werde ich dieses Beispiel kurz vorstellen. Zur besseren Lesbarkeit, habe ich die Funktionen umbenannt. Dies ist zwar nicht das kürzest mögliche Erlang-Quicksort, jedoch ein guter Kompromiss im Hinblick auf die Performance. Ein mit aktuellen Erlang-Versionen möglicher Zweizeiler findet sich beispielsweise in [Ar03, S. 58].

```
quicksort(X) ->
    sort(X, []).

sort([Pivot|Rest], Tail) ->
    {Smaller,Bigger} = split(Pivot, Rest),
    sort(Smaller, [Pivot|sort(Bigger,Tail)]);
sort([], Tail) ->
    Tail.

split(Pivot, L) ->
    split(Pivot, L, [], []).

split(Pivot, [], Smaller, Bigger) ->
    {Smaller, Bigger};
split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->
    split(Pivot, T, [H|Smaller], Bigger);
split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->
    split(Pivot, T, Smaller, [H|Bigger]).
```

Wegen der fehlenden Unterstützung für Arrays wird in Erlang mit Listen gearbeitet und da somit kein effizienter freier Zugriff auf alle Elemente möglich ist, jeweils der Kopf der Liste als Pivot-Element definiert. Zur Steigerung der Performance, wurden die rekursiven Aufrufe von `sort/2` zur Erzeugung der geordneten Liste verschachtelt angeordnet anstatt miteinander verknüpft. Die Funktion `sort/2` erwartet eine ungeordnete Liste und eine geordnete Liste als Argumente. Die ungeordnete Liste wird mit der Funktion `split/2` anhand des Pivot-Elements rekursiv in zwei Listen geteilt.

In `Smaller` werden alle Elemente kleiner und in `Bigger` größer dem Pivot-Element eingefügt. Die geordnete Liste entsteht Stück für Stück beim rekursiven Aufruf `sort(Smaller, [Pivot|sort(Bigger,Tail)])`. Da der enthaltene Aufruf von `sort/2` als Parameter übergeben wird, wird dieser jeweils zunächst abgearbeitet. Zur Verdeutlichung der Funktionsweise dieser Quicksort-Implementierung folgen die Aufrufe von `sort/2` zur Bearbeitung von `quicksort([3,1,4,2])`.

```
sort([3,1,4,2],[])
  sort([2,1],[3|sort([4],[])])
    sort([], [4|sort([],[])])
      []

  sort([2,1],[3,4])
    sort([1],[2|sort([], [3,4])])
      [3,4]
    sort([1],[2,3,4])
      sort([], [1|sort([], [2,3,4])])
        [2,3,4]
      sort([], [1,2,3,4])
        [1,2,3,4]
```

Die Implementierung in Java ist in Anhang A zu finden. Ich habe hierbei ebenfalls das erste Element der zu sortierenden Array-Bereiche als Pivot-Element gewählt.

3.3.2 Vergleich

Aufgrund konzeptionell verschiedener Implementierungen von Quicksort in Erlang und Java, lassen sich diese nur oberflächlich vergleichen. Wie erwartet, ist die Erlang-Umsetzung in weniger Code-Zeilen möglich. Für einen groben Performance-Vergleich wurden Zufallszahlen sortiert. Diese wurden mit dem folgenden, hier in Erlang dargestellten Generator erzeugt.

```
ran(Seed) ->
  (125 * Seed + 1) rem 4096.
```

Die für die Ausführung benötigte Zeit, habe ich jeweils als Nachher-Vorher-Differenz aus `System.currentTimeMillis()` in Java und `statistics(wall_clock)` in Erlang ermittelt. Die in Tabelle 1 aufgeführten Zahlen sind Mittelwerte aus mehreren Messungen mit aus verschiedenen Startwerten erzeugten Zufallszahlen. Die Programme wurden zur besseren Vergleichbarkeit mit dem `gcj`-Compiler aus dem `gcc`-Paket in der Version 3.2.2 sowie dem High-Performance-Compiler aus Erlang R9B-0 in Maschinensprache übersetzt. Ergänzend sind die Sortierzeiten der Funktion `sort/1` des Erlang-Moduls `lists` aufgeführt. Der zugehörige Code kann beispielsweise in der

Datei `/lib/stdlib-1.11.0/src/lists.erl` der oben genannten Erlang-Distribution eingesehen werden.

Anzahl Zufallszahlen	Sortierzeit (ms)		
	Java Quicksort	Erlang Quicksort	Erlang lists:sort/1
1000	0,6	0,6	0,8
10000	4,8	8,2	12,6
100000	78,4	338,4	219,4
1000000	725,4	57707,8	2740

Tabelle 1: Sortierzeiten verschiedener Algorithmen

Offensichtlich ist Quicksort in Java erheblich schneller als in Erlang. Selbst der hoch optimierte Sortieralgorithmus des Moduls `lists` benötigt deutlich mehr Zeit für das Sortieren als Quicksort in Java.

4 Fazit

Erlang wurde von Ericsson für Anwendungen im Telekommunikationsbereich entwickelt und stellt hierfür mit der Open Telecom Platform eine vielfältige Basisarchitektur bereit. Im Gegensatz zu anderen funktionalen Sprachen, konnte sich Erlang bereits in sehr großen Systemen mit über einer Million Code-Zeilen bewähren. Erlang eignet sich gut zur Abbildung nebenläufiger sich gegenseitig kontrollierender Prozesse in z.B. Servern, Switches oder Telefonanlagen sowie zur Realisierung der Kommunikation großer Systeme. Erlang-Prozesse sind schlank genug, um beispielsweise in einer Telefonanlage jede Verbindung über einen eigenen Prozess abzubilden. Erlang zeichnet sich besonders durch die direkte Integration von Konzepten zur verteilten und nebenläufigen Programmierung, die resultierenden Debugging-Möglichkeiten und die Code-Ersetzung aus. Trotz des Ausstiegs von Ericsson aus der aktiven Entwicklung, wird Erlang weiterhin in Forschungsprojekten und kommerziellen Systemen eingesetzt. Software für industrielle Steuerungs- und Kontrollanlagen wird durch die Real-Time-Fähigkeiten zwar möglich, ist jedoch bisher nicht in Großprojekten realisiert worden. Durch die geringe Performance bei Low-Level-Code eignet sich Erlang nicht für die vollständige Entwicklung von Standardanwendungen.

Quicksort in Java

```
...
public static void swap(int[] a,int i, int j) {
    int h;
    h = a[i];
    a[i] = a[j];
    a[j] = h;
};

public static void sort(int[] a, int left, int right) {
    int pivot = a[left];
    int l = left;
    int r = right;
    while (l <= r) {
        while (a[l] < pivot) l++;
        while (a[r] > pivot) r--;
        if (l<=r) {
            swap(a, l, r);
            l++;
            r--;
        };
    };
    if (l < right) sort(a, l, right);
    if (r > left) sort(a, left, r);
};

public static void quicksort(int[] a) {
    sort(a, 0, (a.length-1));
};
...
```

Literaturverzeichnis

- [Ar92] Joe Armstrong u.a.: *Implementing a functional language for highly parallel real time applications*. 8th Int Conf. on Software Engineering for Telecommunication Switching Systems, Florenz, 1992.
- [Ar96] Joe Armstrong u.a.: *Concurrent Programming in ERLANG, Part I*, 2nd. ed., Prentice Hall, 1996. Online im Internet: URL: <http://www.erlang.org/download/erlang-book-part1.pdf>.
[Stand: 2004-05-11]
- [Ar03] Joe Armstrong: *Making reliable distributed systems in the presence of software errors*, Dissertation am Royal Institute of Technology, Stockholm, 2003. Online im Internet: URL: http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf.
[Stand: 2004-05-11]
- [Bi98] Richard Bird: *Introduction to Functional Programming using Haskell*, 2nd. ed., Prentice Hall, 1998.
- [Da01] Mads Dam u.a.: *Verification of Erlang Programs*, Forschungsprojekt am Swedish Institute of Computer Science, 2001. Online im Internet: URL: <http://www.sics.se/fdt/astec/>.
[Stand: 2004-05-11]
- [De96] Pierre Deransart u.a.: *Prolog: The Standard. Reference Manual*. Springer Verlag, 1996.
- [Ha94] Bogumil Hausman: *Turbo Erlang: Approaching the speed of C*, in: *Implementations of Logic Programming Systems*, hrsg. von Evan Tick und Giancarlo Succi, Kluwer Academic Publishers, 1994, S. 119-135.