

*Bei diesem Artikel handelt es sich um eine Arbeitsversion.
Dieser Artikel ist nicht vollständig und ebenso wenig fehlerfrei.
Dennoch denken wir, dass er bereits eine gute Übersicht zu Erlang
und seinen Kernkonzepten gibt. Für kritische Hinweise und
Korrekturvorschläge sind wir unter info@traveling-light.de erreichbar.*

Copyright 2006, Barr & Böckmann.

Moderne Programmierung nebenläufiger und verteilter Anwendungen mit Erlang.

Volkert Barr, Carsten Böckmann

Die eierlegende Wollmilchsau unter den Programmiersprachen, die leicht verteilbare fehlerfreie Programme mit aller kürzesten Entwicklungszeiten verspricht, ist noch nicht gefunden worden. Und doch bringt die Suche danach immer mal wieder Juwelen ans Tageslicht. Mit Erlang soll hier eine viel versprechende Sprache, die ihren Ursprung in Telekommunikationsbereich hat, präsentiert werden.

In diesem Artikel wollen wir die Programmiersprache Erlang vorstellen, die sich bestens für die Erstellung skalierbarer und hochverfügbarer Anwendungen eignet, um zu demonstrieren, dass ein Blick über den Java-Tellerrand durchaus lohnend sein kann. Typische hochverfügbare Anwendungen sind Netzwerkdienste und Web-Services, aber auch Überwachungs- und Steuerungssysteme, wie sie in vielen Bereichen, etwa in der Telekommunikationsbranche, zum Einsatz kommen. Erlang ist eine Programmiersprache zur Entwicklung nebenläufiger und verteilter Anwendungen mit weichen Echtzeitanforderungen und entstammt dem Computer Science Lab der Firma Ericsson [??]. Erlang bietet Konstrukte zur effizienten Lösung von Problemen aus diesem Anwendungsumfeld: Programme werden als handhabbare Einheiten in Module unterteilt. Mechanismen zur Fehlererkennung unterstützen die Entwicklung robuster Systeme. Funktionalitäten können zur Laufzeit ausgetauscht werden.

Zentraler Baustein Erlangs ist ein prozessbasiertes Kommunikationsmodell. Nebenläufigkeitsaspekte werden dabei explizit beschrieben, d.h. der Programmierer kann genau kontrollieren, welche Verarbeitungsschritte sequenziell und welche parallel durchgeführt werden sollen. Der Nachrichtenaustausch zwischen den Prozessen geschieht asynchron. Ein sendender Prozess kann demnach seine Arbeit fortsetzen, sobald die Nachricht verschickt wurde und braucht nicht auf eine Bestätigung vom empfangenden Prozess zu warten. Dies ist die einzige Form der Kommunikation zwischen Prozessen. Daher lassen sich Anwendungen leicht von einem Einprozessor-System auf ein Mehrprozessor-System oder auch auf viele durch ein Netzwerk verbundener Rechner verteilen. Aufgrund der leichten Erlernbarkeit des Nebenläufigkeitsmodells lassen sich in Erlang innerhalb kurzer Zeit komplexe verteilte Anwendungen realisieren. Im industriellen Kontext bedeutet dies kürzere Entwicklungszeiten und damit einhergehend Kosteneinsparungen.

Erlang hat seine Wurzeln in der logischen und funktionalen Programmierung (Lisp, ML, Prolog) aber auch in nebenläufigen Sprachen (Chill, Ada 83). Um einen größeren Kreis von Anwendern zu erreichen, ist Erlang mittlerweile ein Open-Source Projekt [3] und erfreut sich seit Jahren wachsender Beliebtheit. Seinen Anspruch als Entwicklungsumgebung unterstreicht Erlang durch die Open Telecom Platform (Artikel "Erlang / Open Telecom Platform"). Diese hält eine umfangreiche Sammlung von Bibliotheken, generischen Komponenten und Möglichkeiten der Anbindung von C, C++ und Java-Anwendungen wie

Moderne Programmierung nebenläufiger und verteilter Anwendungen mit Erlang. Draft 2.2

Copyright 2006, Barr & Böckmann

auch eine CORBA-Schnittstelle bereit. Erlang gibt es für Windows, Solaris, Linux und Mac OS X sowie für das bekannte Echtzeitbetriebssystem VxWorks.

Auf den nächsten Seiten werden wir gemeinsam in die Welt von Erlang abtauchen. Wir werden uns dabei nicht lange mit den Grundlagen dieser Sprache aufhalten, sondern direkt ins kalte Nass springen und uns auf die wesentlichen Aspekte der nebenläufigen und verteilten Erlangprogrammierung stürzen und nur wo notwendig grundlegende Sprachkonstrukte erläutern. Für eine komplette Einführung möchten wir alle interessierten Leser auf das sehr gute Buch „Concurrent Programming in Erlang“^[?] verweisen, umfangreiche Information sowie die Entwicklungsumgebung findet sich unter www.erlang.org.

Ein Streifzug durch die Welt der nebenläufigen ...

Wie in der Einleitung erwähnt, besitzt Erlang ein nachrichtenbasiertes Kommunikationsmodell, bei dem sich Prozesse gegenseitig Nachrichten zukommen lassen können. Prozesse laufen auf einem Erlang-Knoten (auch Node bezeichnet). Bei einem Erlang-Knoten handelt es sich um eine Virtuelle-Maschine, die als Byte-Code übersetzte Erlangprogramme ausführt. Ganz so wie bei einer Java Virtual Machine (JVM). Auf einem Erlang-Knoten können fast beliebig viele Erlangprozesse erzeugt werden. Anwendungen mit mehreren hundert Prozessen sind nicht untypisch. Um eine Nachricht an einen anderen Prozess zu verschicken, wird in Erlang eigentlich nur der Prozessidentifizier (Pid) des Partners benötigt oder sein mit diesem Identifizier systemweit registrierter Name. Es müssen vom Programmierer keine Sockets oder andere Kommunikationsstrukturen erschaffen und initialisiert werden. Allein die Anweisung der Form

```
Pid ! Message
```

genügt, um eine Nachricht zu verschicken. Der Nachrichtenversand ist also für den Entwickler völlig transparent.

Nachrichten können zwischen Prozessen eines Knotens, aber auch zwischen Prozessen verschiedener Knoten versendet werden. Nachrichten werden stets asynchron versendet, d.h. der Sender wartet nicht explizit darauf, dass der Empfänger die Nachricht direkt entgegennimmt, sondern er führt sein Programm nach absetzen der Nachricht weiter aus. Durch die asynchrone Kommunikation braucht nicht explizit zwischen Kommunikation und Synchronisation unterschieden werden. Programme lassen sich so auch einfacher auf unterschiedliche Prozessoren verteilen.

Damit keine Nachricht verloren geht, besitzt jeder Prozess eine Mailbox in der die empfangenen Nachrichten in Empfangsreihenfolge abgelegt werden. Die Mailbox wird mit der *receive..end*-Operation ausgelesen.

Als Nachrichten sind beliebige Erlang-Terme (Tupel, Listen, Basis Datentypen, komplexe Datentypen, ...) erlaubt. Das asynchrone Nachrichtenmodell ist allgemein gehalten und erweist sich als leicht anwendbar und flexibel. Um die Kommunikation so effizient wie möglich zu gestalten, hilft Erlang u.a. einen Cache für die in Nachrichten oftmals verwendeten Datentypen Atom bereit. Atome sind konstante Zeichenketten und können z.B. zur Kennzeichnung (tag) einer Nachricht dienen. Einmal wird ein Atom mit einer Referenznummer versehen und in einen Cache abgelegt. Beim nächsten Versenden wird dann nur noch die Referenznummer und nicht mehr das komplette Atom verschickt.

Schauen wir uns zunächst in einem Beispiel lokale Erlang-Prozesse an: In unserem Beispiel haben wir einen einfachen Flächenberechnungsserver geschrieben. Dieser Server soll auf Anfrage hin bestimmte Flächen berechnen und das Resultat an den Anfrager zurücksenden. Die Gesamtsumme aller bereits getätigten Berechnungen wird vom Server intern weitergeführt. Der Servercode ist in einem Modul *server_module* untergebracht. Erlang Programme werden in Module organisiert. Eine *export*-Anweisung macht dabei Funktionen eines Moduls für die Außenwelt sichtbar.

```
-module(server_module).
-export([start/0, loop/1]).

square(X) ->
    X*X.
rectangle(X, Y) ->
    X*Y.

start() ->
    spawn(server_module, loop, [0]).

loop(Total) ->
    receive
        {Pid, {square, X}} ->
            Result = square(X),
            Pid ! Result,
            loop (Total + Result);
        {Pid, {rectangle, [X,Y]}} ->
            Result = rectangle(X,Y),
            Pid ! Result,
            loop(Total + Result);
        {Pid, total_area} ->
            Pid ! Total,
            loop (Total)
    end.
```

Wie ist das Programm zu verstehen? Das hier vorliegende Erlangmodul besteht aus einer Moduldeklaration und einem Modulkörper. Der Deklarationsteil

```
-module(server_module).
-export([start/0, loop/1]).
```

definiert den Namen des Moduls *server_module* und die Funktionen (*start()* und *loop()*), die vom Modulnutzer aufgerufen werden können. Der Modulkörper enthält die Funktionsdefinitionen. Ein externer Aufruf von *server_module:start()* erzeugt mit der Funktion *spawn()* einen neuen Erlang-Prozess. Dieser führt die *loop()*-Funktion mit einer initialen Gesamtfläche 0 aus. *Loop()* legt das Verhalten des Servers fest. Die Syntax zur Prozesserzeugung ist *spawn(ModulName, FunctionName, ArgumentList)*. *Spawn* erzeugt als Rückgabewert einen Prozessidentifizier (*Pid*).

An *loop()* fällt umgehend die Struktur

```
receive
    Message1 -> Action1;
    Message2 -> Action2;
    ...
end
```

auf. Es handelt sich bei *receive..end* um eine musterbasierte Operation, mit der Nachrichten empfangen werden können. Nachrichten sind in diesem Beispiel Objekte, die auf strukturelle

Vorgaben (Muster) von *Message1*, *Message2*, ... passen können. In unserem Beispiel handelt es sich bei den Vorgaben um einfache Tupel, die Werte mit sich tragen. Die *receive..end* - Funktion wartet somit auf Objekte der beschriebenen Art und führt, falls sich ein solches in der Mailbox befindet, die eine Aktion (*Action1*, *Action2*, ...) aus. Die Auswahl der Aktion geschieht über Pattern-Matching. Durch das Pattern-Matching werden alle Variablen aus dem Muster mit den Werten aus der Nachricht gebunden. Diese Variablen können dann in der Aktion verwendet werden.

Unser Flächenserver kann u.a. Nachrichten der Form *{Pid, {square, X}}* empfangen, wie im Message/Action-Paar

```
...
{Pid, {square, X}} ->
    Result = square(X),
    Pid ! Result,
    loop (Total + Result);
...
```

festgelegt. *square* ist ein Atom, *Pid* und *X* sind Variablen. In Erlang werden Variablen stets groß und Atome klein geschrieben, Tupel haben die Form *{X1, X2, X3, ...}* und werden zur Speicherung einer festen Liste von Elementen benutzt, während Listen durch *[X1, X2, X3 ...]* dargestellt werden und von ihrer Kapazität her variabel sind. Variablen können nur einmal mit einem Wert belegt werden. Die Variable *Pid* ist in unserem Beispiel mit dem Prozessidentifizier des sendenden Prozesses assoziiert. Diesen Identifizier brauchen wir als Rücksendeadresse für das Ergebnis der Flächenberechnung *square* von *X*.

Für eine *square* Nachricht, wird zunächst die Funktion *square(X)* ausgewertet und das Ergebnis an den Anfrager mit dem Prozessidentifizier *Pid* gesendet. Anschließend wird *loop()* rekursiv mit dem Ergebnis *Total+Result* unserer Gesamtflächenberechnung aufgerufen. Um den Prozessidentifizier des Flächenservers bei allen Klienten auf unserem Knoten bekannt zu machen, können wir diesen als Service unter einem Namen auf unserem mit *register()* registrieren (*area_server*) und ihn direkt über diesen ansprechen.

```
start() ->
    Pid = spawn(server_module, loop, [0]),
    register(area_server, Pid).
```

Wie sendet nun ein Klient dem Server eine Nachricht? Der Programmtext eines möglichen Klienten sieht für den Kommunikationsanteil so aus:

```
area_server ! {self(), {square, 2}},
receive
    Area -> ...
...
```

Die Funktion *self()* liefert den Prozessidentifizier des die Funktion aufrufenden Prozesses. Dieser wird in unserem Beispiel als Rücksendeadresse mit an den Flächenberechnungsserver gesendet.

Der Klient wartet nach dem Absenden der Anfrage *{self(), {square, 2}}* auf das Berechnungsergebnis, das, wenn es in der Mailbox eintrifft, an die Variable *Area* gebunden wird und ausgewertet wird.

Der Vollständigkeit halber erläutern wir den kompletten Aufbau der *receive..end*-Funktion, da diese noch zwei wichtige Konzepte beinhaltet.

```

receive
    Message1 [when Guard] ->
        Action1;
    Message2 [when Guard] ->
        Action2;
    ...
    [after TimeExp ->
        ActionT]
end

```

Die Verarbeitung von Nachrichten kann durch spezielle Bewacher (Guards) gesteuert werden. Ein Guard ist ein Boolescher Ausdruck. Ist der Wert eines Guards *true* und befindet sich zeitgleich eine auf ein Receive „matchende“ Nachricht in der Mailbox, wird diese verarbeitet. Interessant ist auch das *after*-Statement mit dem sich leicht Timeouts realisieren lassen: Wird innerhalb der *TimeExp* - ein als Millisekunden interpretierter Integerwert - keine Nachricht aus der Mailbox abgearbeitet, wird die Aktion *ActionT* ausgewertet. Mit der *after*-Anweisung kann schnell ein Timer gebaut werden, der einem Prozess nach verstrichener Zeit eine Nachricht zukommen lässt. Etwa so:

```

-module(timer).
-export([timeout/2, cancel/1, timer/3]).

timeout(Time, Alarm) ->
    spawn(timer, timer, [self(), Time, Alarm]).

cancel(Timer) ->
    Timer ! {self(), cancel}.

timer(Pid, Time, Alarm) ->
    receive
        {Pid, cancel} ->
            true
    after Time ->
        Pid ! Alarm
    end.

```

Durch den Aufruf von *timer:timeout(Time, Alarm)* wird ein Timer-Prozess erzeugt. Die Wartezeit *Time* gibt an, nach Ablauf welcher Zeit der den Timer erzeugende Prozess eine Alarm-Nachricht zugesendet werden soll. Mit *timer:cancel(Timer)* kann der Timer gestoppt werden.

In eingebetteten Anwendungen gibt es typische, immer wiederkehrende Standardaufgaben. Dazu gehören z.B. das Protokollieren (logging) und Überwachen (monitoring) bestimmter Systemfunktionen und Systemzustände. In dem folgenden stark vereinfachten Beispiel sollen etwa Zustandsinformationen einer Batterie eines fiktiven Satelliten überwacht werden.

```

-module(monitor_battery).
-export([start/0, init/0]).

start() ->
    spawn(monitor_battery, init, []).

init() ->
    data_management:subscribe(battery_info, [temperature, voltage]),
    server().

server() ->
    receive
        {data_management, battery_info, [Temp, Voltage]} ->
            check(Temp, Voltage),

```

```

server()
end.

check(Temp, Voltage) when Temp < 10.0 ->
    alarm ! {self(), {alarm, {battery_to_cold, Temp}}};
check(Temp, Voltage) when Temp > 60.0 ->
    alarm ! {self(), {alarm, {battery_to_hot, Temp}}};
check(Temp, Voltage) when Voltage < 2.3 ->
    alarm ! {self(), {alarm, {battery_voltage_to_low, Voltage}}};
...

```

Der Prozess `monitor_battery` meldet sich durch den Funktionsaufruf `data_management:subscribe(battery_info, [temperature, voltage])` bei einer hier nicht näher beschriebenen Komponente Data-Management für die Temperatur -und Spannungswerte der Batterie an. Das Data-Management sendet die Werte anschließend in regelmäßigen Abständen als Nachricht `{data_management, battery_info, [Temp, Voltage]}` an den Monitor. Die Funktion `check()` überprüft die Werte und sendet im Fehlerfall eine Statusmeldung an einen Prozess namens `alarm`.

... und verteilten Programmierung mit Erlang

Nachdem wir uns nun relativ lange mit der Programmierung auf einem einzelnen System beschäftigt haben, möchten wir einen anderen sehr attraktiven Aspekt Erlangs behandeln, die Programmierung verteilter Anwendungen.

In einem verteilten Programm sollen die einzelnen Prozesse auf unterschiedlichen Rechnern laufen und im Zusammenspiel eine komplexere Aufgabe lösen. Für eine solche Zerlegung spielen nicht nur Performanzaspekte eine Rolle, oftmals ist die Verteilung aufgrund der Problemstellung nahe liegend. Denken sie an ein Buchungssystem, bei dem die Terminals quer über das Land verstreut stehen können. Mit Erlang lassen sich solche verteilten Anwendungen besonders einfach realisieren. Sehen sie selbst.

Voraussetzung für eine Verteilung der Anwendung ist, dass es auf jeder Maschine ein eigenes Erlang-Laufzeitsystem gibt, auf dem die Erlangprogramme ausgeführt werden. Die Rechner müssen natürlich einem Netzwerk angehören. Ein solches Laufzeitsystem wird im Sinne der verteilten Programmierung dann als Knoten bezeichnet, der einen eindeutigen Namen in der Form von `name@host` besitzt. Um einen Prozess auf einem entfernten Knoten zu starten, reicht weiterhin ein `spawn`-Aufruf, dem in diesem Falle allerdings ebenfalls der Zielknoten mitgeteilt wird. Die Kommunikation zwischen Prozessen auf unterschiedlichen Rechnern geschieht nach wie vor mittels Prozessidentifizier und wird durch die Anweisung

```
{Name, Node} ! Message
```

eingeleitet. Die Variable `Node` ist der Hostbezeichner mit einem vorangestellten Knotennamen (`node = node@host`). `Name` ist ein auf dem Knoten registrierter Prozess. Stellen wir uns vor, unser Flächenberechnungsserver braucht für seine Berechnung spezielle Hardware, die auf dem Rechner `erlang.org` installiert ist. Die Klienten laufen nun auf irgendwelchen anderen Rechnern. Um nun den Server anzusprechen ruft ein Klient schlicht

```
{area_server, 'server@erlang.org'} ! {self(), {square,2} }
```

auf.

Gibt es einen Prozess *area_server* auf dem Knoten *server* des Rechners *erlang.org*, wird die Nachricht an diesen zugestellt, andernfalls geht sie verloren. Der Verbindungsaufbau zu einem entfernten Knoten wird im Übrigen von der Erlang-Laufzeitumgebung automatisch übernommen, wenn dieser mit einem solchen Knoten in Kontakt treten möchte. Hier braucht der Programmierer keine Hand anlegen. Das hört man doch gern. Mit der Funktion *node()* erhält man den eigenen Knotennamen, mit *nodes()* die Namen aller dem Knoten bisher bekannten Knoten.

Es gibt viele Gründe, Geschwindigkeitszuwachs die für oder der eine verteilte Anwendung sprechen. Neben der natürlichen Abbildung der Problemstellung, lässt sich durch eine Verteilung auch die Robustheit/Fehlertoleranz einer Anwendung erhöhen. Hierzu bietet Erlang auch einiges.

Fehlertolerante und robuste Programme

Fehlerfreie Programme sind noch immer der Wunschtraum eines jeden Entwicklers und da die Diskussion um Fehlerfreiheit eine rein philosophische ist, müssen Programme auf bestimmte Ausnahmesituationen hin vorbereitet werden. Nur so können sich Programme kontrolliert, robust und fehlertolerant verhalten. Dazu bietet Erlang im Wesentlichen die folgenden Mechanismen:

- die Überwachung der Auswertung eines Ausdrucks,
- das Abfangen der Auswertung undefinierter Funktionen
- die Überwachung des Verhaltens ganzer Prozesse.

Die Auswertung einer Division durch 0, einer Zuweisung wie $2 = 3$ oder aber das Aufrufen einer undefinierten Funktion führt in Erlang zu einer Exception und damit möglicherweise zum Abbruch des entsprechenden Prozesses. Zum Überwachen von Ausdrücken und zum Abfangen solcher Fehler gibt es die auch in anderen Sprachen bekannten *catch()* und *throw()* Funktionen. In den *catch*-Blöcken werden durch die eigenen Fehlerbehandlungen realisiert. Darüber hinausgehend können die Default-ErrorHandler Erlangs umgeschrieben und an die Wünsche des Programmierers angepasst werden.

Auf höherer Ebene macht Erlang die Überwachung ganzer Prozesse möglich. Mit der Funktion *link()* kann ein Prozess sich mit einem anderen Prozess verbinden (linked processes). Stürzt danach einer der beiden Prozesse ab, wird üblicherweise auch der andere Prozess über ein EXIT-Signal beendet. Ein solches Signal ist ein Tupel {EXIT, ExitingPid, Reason}.

Fehlertoleranz bringt erst die *trap-exit()*-Funktion, mit der solche EXIT-Signale als normale Nachrichten beim empfangenen Prozess behandelt werden können. Der Empfängerprozess ist dann in der Lage, dem Benutzer eines Systems über das Ableben eines anderen Prozesses zu informieren oder aber gar in eigener Regie den vorher abgestürzten Prozess wieder, eventuell auf einem anderen Knoten, neu zu starten. Mit Hilfe dieses Mechanismus kann ein nebenläufiges System als hierarchischer Baum (Supervisor Tree) von Überwacherprozessen geschaffen werden und damit ein hoher Grad an Robustheit gegenüber Fehlern erreicht werden.

Die Erstellung derartiger Supervisor Trees wird in Erlang durch ein entsprechendes Behaviour (siehe Artikel Erlang/OTP) unterstützt. Dabei werden dem Entwickler im Wesentlichen zwei verschiedene Supervisor-Verhaltensmuster angeboten. Bei One-For-One

wird jeder gestorbene Kindprozeß vom Supervisor wieder zu neuem Leben erweckt, während bei One-For-All alle Kinderprozesse auf gleicher Ebene ebenfalls durch den Supervisor getötet werden.

Hot Code Replacement

Für viele hochverfügbare Anwendungen ist ein Rebooten des Systems, etwa um fehlerhafte Software zu beheben, nicht akzeptabel. Erlang ist speziell für solche "non-stop"-Systeme, das sind Systeme mit z.B. 99.999 Verfügbarkeit (5 Minuten downtime im Jahr), entwickelt und bietet daher einfache Möglichkeiten, fehlerhafte Programme und Daten im Betrieb zu ersetzen. So können Module mit neuen Funktionalitäten in ein laufendes System geladen werden. Es können sogar mehrere Versionen eines Moduls gleichzeitig existieren. Spezielle Funktionen wie z.B. `code:load_file(Module)`, `code:delete_module(Module)`, `code:purge_module(Module)` machen dieses möglich. Das folgende Beispiel soll zeigen, wie bei einem Server eine einzelne Funktion zur Laufzeit ausgetauscht werden kann. Wir nehmen an, dass die Funktion `New_Fun` aus einem bereits neu hin zu geladenen Modul stammt.

```
loop(State, Fun) ->
    receive
        {request, Pid, Input} ->
            {Result, New_State} = Fun(Input, State),
            Pid ! Result,
            loop (New_State, Fun);
        {change_code, New_Fun} ->
            loop(State, New_Fun)
    end.
```

State beschreibt den lokalen Zustand des Servers. *Fun* ist eine Funktion, die das Verhalten des Servers festlegt. Die aktuellen Ergebnisse von *Fun* sind von dem derzeitigen Systemzustand State und dem übermittelten Wert Input abhängig. Um die Implementierung von *Fun* zur Laufzeit auszutauschen, senden wir dem Server eine Nachricht `{change_code, New_Fun}`. In dieser Nachricht ist `New_Fun` der Namen der nun zu verwendenden "fehlerfreien" Funktion. `New_Fun` wird durch den rekursiven `loop`-Aufruf an die Variable *Fun* gebunden, die alte Implementierung wird durch den Erlang-Garbage Collector verworfen.

Real-Time und der Interpreteransatz

Erlang aus der Suche nach einer kompakten, leicht erlernbaren und portablen Programmiersprache für Telekommunikationsanwendungen entstanden. Der Interpreteransatz, der dazu auch bei Java verfolgt wurde, gewährleistet zwar auf der einen Seite Portabilität und Sicherheit in punkto Speicherzugriffe und Robustheit gegenüber einzeln abstürzenden Prozessen, er wirft aber natürlich auch die Frage nach dessen Echtzeittauglichkeit auf. Erlang ist mit Sicherheit nicht geeignet für kritische Echtzeitsysteme und systemnaher Software. Aber doch immerhin zur sicheren Steuerung und Überwachung solcher Systeme. Dazu muss jeder Interpreter in Bezug auf die Ausführung der Erlang-Prozesse folgende Vorgaben erfüllen: Das Scheduling ist fair in dem Sinne, dass alle ablaufbereiten Prozesse auch ausgeführt werden; wenn möglich in der Reihenfolge, in der sie lauffähig geworden sind. Kein Prozess soll den ganzen Rechner blockieren. Aus diesem Grund wird der aktuell laufende Prozess nach einer bestimmten Zeit (time-slice) vom Prozessor genommen und ein nächster kann fortfahren. Eine time-slice endet nach ca. 500 Reduktionen/Funktionsaufrufen. Mit diesen Vorgaben lassen sich Echtzeitsysteme mit weichen Zeitanforderungen und Antworten im Bereich 5-20ms durchaus realisieren.

Sehr leistungsfähig ist das leichtgewichtige Prozessmodell-Erlangs, Anwendungen mit mehreren tausend Prozessen sind hier kein Problem. Die Prozesserzeugungszeiten bleiben auch für große Prozesszahlen konstant. Mit Interpreter ist der Erlang Emulator, eine Virtuelle Maschine, gemeint. Es ist klar, dass durch diesen plattformunabhängigen Ansatz, die Laufzeit der Anwendungen sich gegenüber Maschinencode erhöhen. Ähnlich wie bei den JVM sind jedoch unterschiedliche Optimierungen der Erlang VM möglich und bereits integriert (siehe HiPE [4]).

Fazit

Erlang ist nicht die "eierlegende Wollmilchsau" unter den Programmiersprachen! Wie aber hoffentlich durch diesen Artikel sichtbar wurde, bietet Erlang sehr gute Instrumente, um in kurzer Zeit verteilte und fehlertolerante Systeme zu entwickeln. Neben der leichten Erlernbarkeit der Sprache sorgt sein deklarativer Charakter für eine messbare Produktivitäts- und Qualitätssteigerung um mehr als das 4-fache gegenüber imperativen Sprachen wie C/C++ und Java. Projekte, wie die AXD 301 ATM Switch von Ericsson [??], dokumentieren dieses eindrucksvoll. Diese Switch hat immerhin ein Codevolumen (reiner Anwendungscode, ohne OTP) von über eine Million Zeilen Erlang, 400.000 Zeilen C/C++ (Treibersoftware) und 13.000 Zeilen Java (GUI). Mit der Sprache können schnell ablauffähige Programme erzeugt werden, die im Sinne eines Prototyps zur Überprüfung von Systemanforderungen dienen können. Erlang unterstützt somit auch moderne leichtgewichtige Entwurfsprozesse (Extreme Programming) spielend.

Zum Studium weiterer Erlang-Mechanismen etwa zum Thema "Sicherheitskonzepte" sei das Buch [??] wärmstens empfohlen. Erlang hat seine Praxistauglichkeit bereits in vielen Telekommunikationsanwendungen unter Beweis gestellt und besitzt durchaus das Rüstzeug zu mehr. Und wenn es auch schon wegen des Interpreteransatzes noch nicht für Systeme mit kritischen Echtzeitbedingungen geeignet ist, erscheint es aber doch für Monitoring und Kommandierungssysteme und andere verteilte, fehlertolerante Anwendungen (Web-Services, Spiele-Server, ...) eine sehr interessante Alternative zu Java oder C++ zu sein. Mit der Open Source Initiative wurde schon ein sehr großer Schritt zur Verbreitung der Programmiersprache Erlang gemacht, wir wollen mit diesem Artikel unseren Beitrag dazu leisten. Leisten sie doch auch einen.