Programmierparadigmen

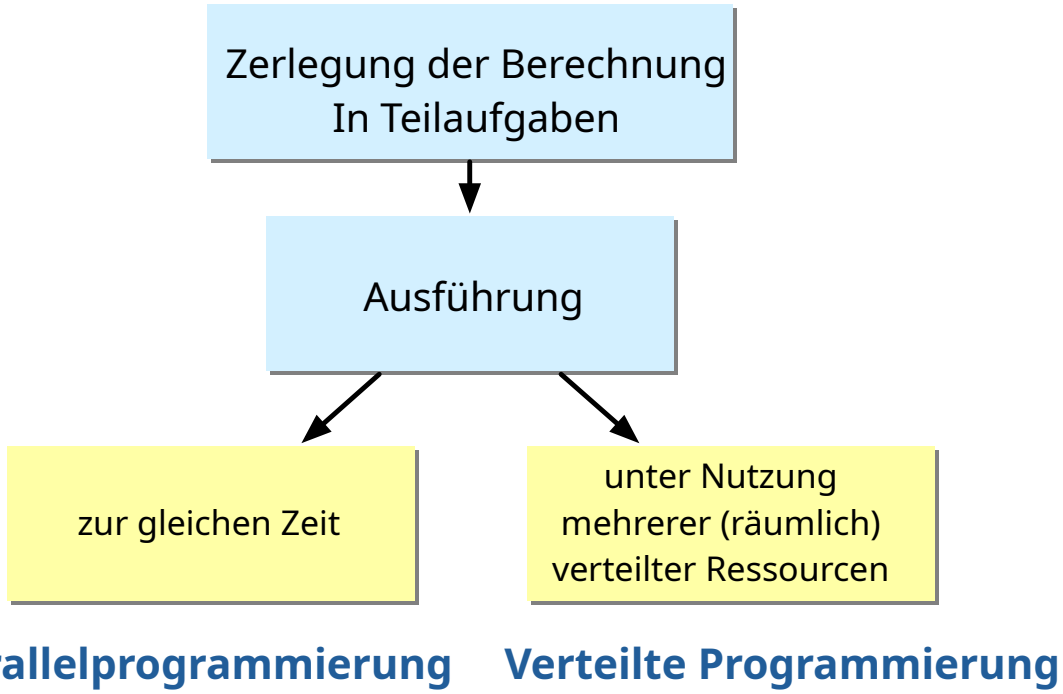
Kapitel 5
Verteilte Programmierung in Erlang

(Diese Folien beruhen z.T. auf einem Foliensatz von Prof. Dr. Kai Uwe Sattler)

Überblick

- Grundlagen verteilter Programmierung
  - Einordnung, Ziele & Motivation
  - Herausforderungen, Anforderungen & Eigenschaften
  - Aktormodell
- Verteilte Programmierung in Erlang
  - Aktormodell in Erlang
  - (Erweiterte) Deterministische Endliche Automaten (DEA)
  - Beispiel 1: Implementierung eines erweiterten DEA in Erlang
  - Beispiel 2: Alternating Bit Protocol
- Zusammenfassung & Ausblick

Einordnung



Ziele

- Bisher:
  - Eine Maschine (evtl. mit mehreren CPU Cores)
  - Prozesse kommunizieren nur innerhalb dieser Maschine
- Jetzt:
  - Mehrere Rechner
  - Prozesse auf verschiedenen Rechnern
- Erfordert:
  - Kommunikation über Knotengrenzen hinweg
  - Behandlung von Knoten- oder Netzwerkausfällen

Motivation (1)

- Viele verschiedene Systeme (Knoten) zur Verfügung
  - PC, Server, virtuelle Maschine
  - Lastverteilung, Spezialisierung auf bestimmte Probleme, ...
- Knoten sind über Netwerke verbunden
  - LAN (Wohnräume, Büros, Uni-Campus): bis zu 10 GBit/s
  - Metropolitan Area Network (MAN; dichtbesiedelte Regionen, Behördennetze): bis zu 10 GBit/s
  - Wide Area Network (WAN; weltweite Vernetzung): hohe Kapazitäten zwischen ISPs

Motivation (2)

Wofür werden verteilte Systeme eingesetzt?

- Gemeinsame Nutzung von Ressourcen:
  - Cloud-Umgebungen
  - Verteilte (Datenbank)-Systeme
  - Ggf. spezialisierte Hardware für bestimmte Aufgaben (z.B. GPU-Server, Storage-Server, etc.)
- Teilaufgaben in großen Anwendungen
  - Parallele Ausführung
  - Getrennte Teilaufgaben (Micro-Services)
- Informationsaustausch
  - E-Mail, Messenger
  - Verteilte Algorithmen

Software-Architekturen

1. **Früher**: Hardware, Betriebssystem, Anwendung
   - Virtualisierung von Prozessor, Speicher, E/A Systemen
   - Interprozesskommunikation (IPC)
2. **Middlewaresysteme**: HW, OS, Middleware, Anwendung
   - Verteilte Dienste
   - Programmierparadigmen: Remote Procedure Call (RPC), Client/Server, ...
   - Java Remote Method Invocation (RMI), CORBA, ...
3. Heute: **Virtualisierung**
   - VM Hypervisor: verstecken Hardware vor Betriebssystem
   - Docker: eine Anwendung pro Container (→ Einsatz von Techniken verteilter Programmierung auch bei lokaler Ausführung)

Herausforderungen

- Viele verschiedene Computer/Server
  - Verschiedene Betriebssysteme
  - Unterschiedliche Leistungsfähigkeit
- Systemkomponenten müssen miteinander kommunizieren
- Verteilte Algorithmen:
  - Nachrichten senden, empfangen, bestätigen,
  - Synchronisation
  - Knotenausfälle behandeln

⇒ **brauchen Modelle zur Beschreibung der Kommunikation.**

Anforderungen & Eigenschaften

- (Last-)Skalierbarkeit (Scale-out)
  - Viele kleine Server – statt eines großen
  - Neue Server nach Bedarf hinzufügen
- Funktionssicherheit (Safety) / IT-Sicherheit (Security)
  - Fehlertoleranz / Verfügbarkeit
  - Ausfälle von einzelnen Knoten kompensieren
  - Redundante Verarbeitung
  - Sicherung gegen Angreifer (Abhören, Manipulation, ...)
- Offenheit / Interoperabilität
  - Neue Knoten und Systeme möglichst einfach integrieren
- Transparenz
  - Verstecke die vielen Server vor Anwendern

→ Diese Dinge geschehen nicht automatisch durch Verteilung!

Kommunikation als Voraussetzung

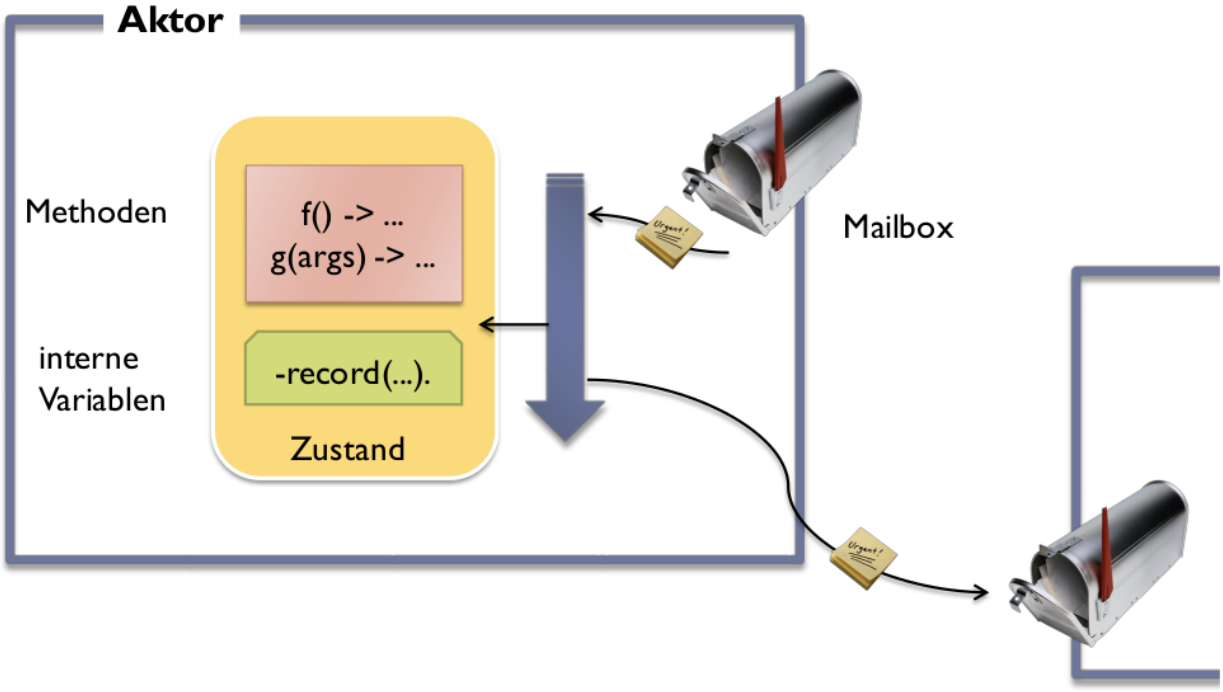
- Verteilte Programmierung: Progamme müssen einen entfernten Computer ansprechen
- Erfordert: Adresse → IP-Adresse
  - Da mehrere Dienste auf demselben Computer laufen, lauscht jeder Dienst auf einem Port (Nummer)
  - Wichtige Ports: 80 (HTTP), 22 (SSH), 25 (SMTP)
- Socket beschreibt einen Endpunkt, d.h. Adresse & Port in einem Netzwerk (TCP oder UDP) :
  - Server-Socket wartet auf Verbindungen
  - Client initiiert Verbindung, ebenfalls über einen (Client-)Socket
- Erlang nimmt Programmierern vieles ab:
  - Einfacher Verbindungsaufbau mittels eines Funktionsaufrufs
  - Verteilte Programmierung nach dem **Aktormodell**

Aktormodell (1)

- Formales Modell für Nebenläufigkeit und Verteilung
  - C. Hewitt, P. Bishop, R. Steiger: *A Universal Modular Actor Formalism for Artificial Intelligence.* Proc. 3rd International Joint Conference on Artificial intelligence, 1973.
  - Basis für verschiedene Programmiersprachen/Frameworks: Erlang, Akka (Implementierung des Aktor-Modells für Scala und die Java Virtual Machine), Theron++ (Library für C++)
- Prinzipien:
  - Aktor kapselt Zustand und Verhalten
  - Aktoren sind aktiv
  - Aktoren kommunizieren durch Nachrichtenaustausch
    - Nichtblockierendes Senden
    - Blockierendes Empfangen

  → Verklemmungsfrei, wenn Aktoren nur eigenen Speicher nutzen

Aktormodell (2)



- Jeder Aktor nimmt Nachrichten in seiner **Mailbox** entgegen

Aktormodell in Erlang (1)

- Aktormodell in Erlang nativ umgesetzt:
  - Sende- und Empfangsoperationen schon für parallele Programmierung eingesetzt
  - Bisher aber nur auf einem Knoten genutzt
- Programmbestandteile im Aktormodell
  - Verhaltensdefinition: `f() -> ... end.`
  - Erzeugen neuer Aktoren: `Pid = spawn(fun ... ).`
  - Empfangen von Nachrichten: `receive ... end.`
  - Senden: `Pid ! Request.`
  - Ggf. verbinden mit entferntem Knoten: `net_adm:ping('node2@localhost').`
- Kein globaler Zustand!

Aktormodell in Erlang (2)

- Erlang-Knoten starten (sname = short name):
  - ```
    #erl -sname node1 –setcookie 1234
     Eshell V11.0 (abort with ^G)
     (node1@localhost)1>
    ```
- Weiteren Erlang-Knoten starten (selber oder anderer PC):
  - ```
    #erl -sname node2 –setcookie 1234
     Eshell V11.0 (abort with ^G)
     (node2@localhost)1>
    ```
- Liste der verbundenen Knoten:
  - ```
    (node1@localhost)1> nodes().
    []
    ```



    ```

Aktormodell in Erlang (3)

- Verteilte Erlang-Knoten benötigen zur Kommunikation gemeinsames **Magic Cookie** (Passwort)
- Mehrere Varianten:
  - Datei `~/.erlang.cookie`
  - Erlang-Funktion `erlang:set_cookie(node(), Cookie).`
  - Option `erl -setcookie Cookie`

Aktormodell in Erlang (4)

- Verbindungsaufbau mittels `net_adm:ping` Funktion:
  -
```
(node1@localhost) 1> net_adm:ping('node2@localhost').
pong
```
  -
```
(node1@localhost) 2> nodes().
['node2@localhost']
```
  -
```
(node2@localhost) 1> nodes().
['node1@localhost']
```


Aktormodell in Erlang (5)

- Starten eines Prozesses auf entferntem Knoten:

```
1  complicated() ->
2   receive
3    {Sender, I} -> Sender ! I*I
4   end.
5  sender(Num, Pid) ->
6   Pid ! {self(), Num},
7   receive
8    Res -> io:format("Result = ~p~n",[Res])
9   end.
```

```
(node1@localhost) >
 N2Pid = spawn('node2@localhost', fun complicated/0)

(node1@localhost) > sender(25, N2Pid).
 Result = 625
 ok
```


Aktormodell in Erlang (6)

- **Verteilte Programmierung** vs. **parallele Programmierung** mit Erlang:
  - Außer dem zusätzlichen Parameter für den Rechnernamen bei dem Aufruf von `spawn(...)` wenig Änderungen erforderlich → bitte Beispiele aus Kapitel 4.b jeweils verteilt ausprobieren
  - Erfordert zusätzliches Abfangen neuer Fehlersituationen
  - Einige unterstützende Erlang-“Features“:
    - Port Mapper: Abbildung Knotenname → (Adresse, Port)
    - Modul für globale Namensregistrierung
    - Funktionen für Verteilen/Laden von Erlang Modulen
    - Nützliche Build-in Functions (BIFs):
      - Abfragen lokaler Magic Cookies
      - Prozesserzeugung
      - Knoten-Monitoring (→ Überwachung und ggf. Neustart von Prozessen für Fehlertoleranz)

Hintergrund: Endliche Automaten (1)

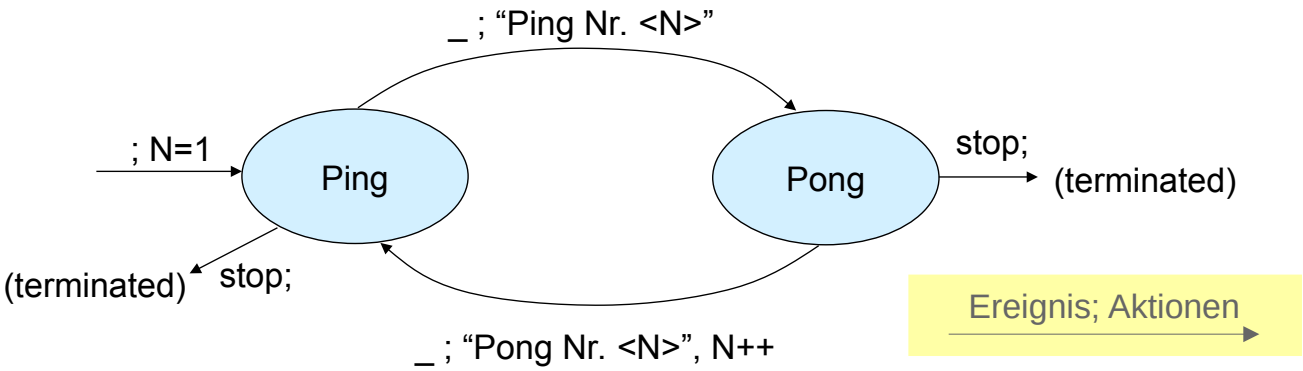
- Ein deterministischer endlicher Automat (DEA) ist ein Quintupel $A = (\Sigma; Q; s; F; \delta)$ mit:
  - Σ ist ein Alphabet
  - Q ist eine endliche Menge, deren Elemente wir Zustände nennen
  - $s \in Q$ ist der sogenannte Startzustand
  - $F \subseteq Q$ ist die Menge der sogenannten Endzustände, auch akzeptierende Zustände genannt
  - $\delta : Q \times \Sigma \rightarrow Q$ ist die sogenannte Übergangsfunktion
- Erläuterung:
  - Der Automat erhält als Eingabe ein Wort $w \in \Sigma^*$
  - Zu Beginn befindet er sich in einem Startzustand
  - Er liest w zeichenweise von links nach rechts, wobei jedes Zeichen einen Zustandsübergang gemäß Funktion δ bewirkt
  - Befindet er sich nach Abarbeitung von w in einem Endzustand, so wird w akzeptiert, anderenfalls wird w abgelehnt

Hintergrund: Endliche Automaten (2)

- Endliche Automaten (Finite State Machines, FSM) werden in der Informatik zur Lösung vielfältiger Aufgaben eingesetzt – zwei wichtige Anwendungsgebiete sind:
  - Compilerbau: Prüfen, ob ein Wort Element einer Sprache ist (siehe vorige Folie)
  - Telekommunikation: Spezifikation von Kommunikationsprotokollen
- Bei der Spezifikation von Kommunikationsprotokollen wird oft mit einer Variante der FSM, den erweiterten endlichen Automaten (Extended FSM, EFSM) gearbeitet:
  - EFSM verfügen zusätzlich über Variablen, die Werte aus einem endlichen Definitionsbereich aufnehmen können; hierdurch werden genau genommen sogenannte „Nebenzustände“ eingeführt
  - Weiterhin können für einen Zustandsübergang Bedingungen über Variablenbelegungen gefordert werden und bei einem Übergang können Nachrichten gesendet sowie Variablen manipuliert werden

Beispiel 1: Ein einfacher Ping-Pong-Server (1)

- Im folgenden Beispiel wollen wir die Implementierung von erweiterten endlichen Zustandsautomaten in Erlang demonstrieren



- Der Automat startet nach Erzeugung im Zustand Ping, die interne Variable N wird auf 1 initialisiert
- Bei Erhalt der Nachricht `stop` hält er an; bei Erhalt beliebiger anderer Nachrichten (`_`) gibt er “Ping Nr. <N>” bzw. “Pong Nr. <N>” aus

Beispiel 1: Ein einfacher Ping-Pong-Server (2)

- Implementierung eines EFSM in Erlang:
- Für jeden Zustand wird eine eigene Funktion implementiert, diese Funktion führt ein `receive` aus und ruft je nach Eingabe und ggf. Bedingungen an Variablen (implementiert durch Guards) die Funktion des entsprechenden Folgezustands auf
- Die aktuelle Variablenbelegung des Automaten wird den Funktionen jeweils als Parameter übergeben
- Es ist darauf zu achten, dass nach dem Aufruf des Folgezustandes keine Berechnung mehr erfolgt (wie bei endrekursiven Funktionen), da andernfalls der Aufrufstack des Prozesses beliebig anwachsen kann
- Zusätzlich wird eine Funktion zur Erzeugung/Initialisierung des Automaten implementiert, die ein `spawn` mit der Funktion des Startzustandes ausführt, welcher initiale Werte für die Variablen des Automaten übergeben werden

Beispiel 1: Ein einfacher Ping-Pong-Server (3)

```
1   -module(pingpong).
2   -export([go/0]).
3
4   go() -> spawn(fun() -> ping(1) end).
5
6   ping(N) ->
7    receive
8     {From, stop} -> true;
9     {From, Msg}  -> io:format("Ping Nr. ~.10B ~n", [N]),
10                    pong(N)
11   end.
12
13  pong(N) ->
14   receive
15    {From, stop} -> true;
16    {From, Msg}  -> io:format("Pong Nr. ~.10B ~n",[N]),
17                    ping(N+1)
18   end.
```


Beispiel 1: Ein einfacher Ping-Pong-Server (4)

- Test in der Erlang-Shell:
  - `> c(pingpong).`
  - `> Pid = pingpong:go().`
    `<0.66.0>`
  - `> Pid!{self(), bla}.`
    `Ping Nr. 1`
    `{<0.47.0>,bla}`
  - `> Pid!{self(), bla}.`
    `Pong Nr. 1`
    `{<0.47.0>,bla}`
  - `> Pid!{self(), bla}.`
    `Ping Nr. 2`
    …
  - `> Pid!{self(), stop}.`
    `{<0.47.0>,stop}`

Empfangen mit Timeouts

- In der Praxis kommt es oft vor, dass eine erwartete Nachricht nicht ankommt, z.B. weil der Kommunikationskanal diese verloren hat
- Damit ein Ablauf in diesem Fall nicht blockiert, sollte ein Wecker gestellt werden, der einen auf die Fehlersituation hinweisen kann
- Klingelt der Wecker (Timeout), dann kann eine entsprechende Aktion (z.B. Neuübertragung der letzten Nachricht) ausgelöst werden
- In Erlang:

```
1  receive
2    Pattern1 [when Guard1] -> Expressions1;
3    Pattern2 [when Guard2] -> Expressions2;
4    ...
5    after Time -> Expressions
6  end.
```

- Nach `Time` Millisekunden wird der entsprechende Ausdruck ausgewertet

Registrieren von Prozessen

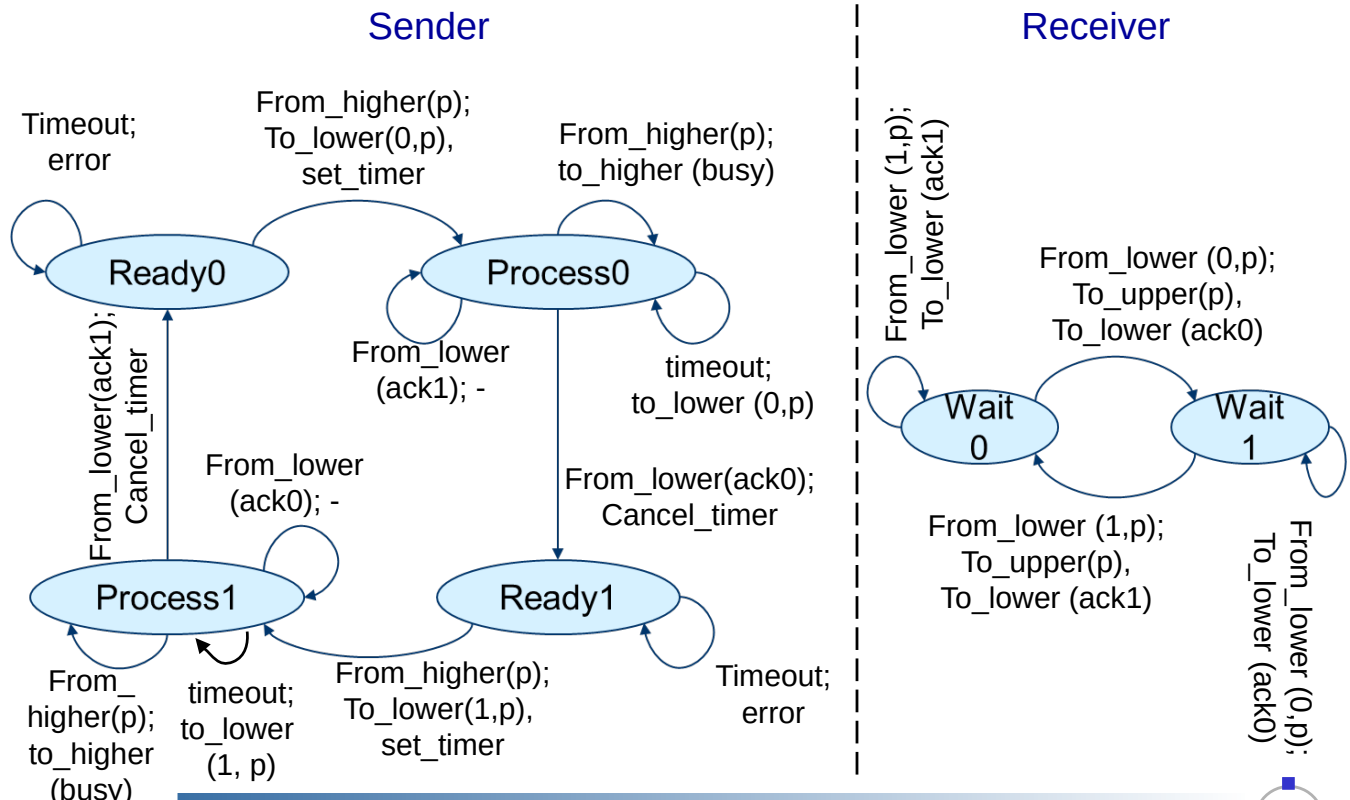
Prozesse können unter einer eindeutigen Kennung im Erlang-System registriert werden:

- `register(anAtom, Pid)`
  - Registriert den Prozess `Pid` mit dem Namen `anAtom`
- `unregister(anAtom)`
  - Entfernt eine evtl. vorhandene Registrierung von `anAtom`
  - Bei Beendung werden Prozesse automatisch deregistriert
- `whereis(anAtom) -> Pid | undefined`
  - Ermittelt die `Pid` des unter `anAtom` registrierten Prozesses
- `registered() -> [anAtom::atom()]`
  - Gibt eine Liste sämtlicher registrierter Atome zurück
- `anAtom!Msg`
  - Sendet Nachricht `Msg` an registrierten Prozess `anAtom`

Beispiel 2: Das Alternating-Bit-Protokoll (1)

- Das Alternating-Bit-Protokoll ermöglicht es, Nachrichten über einen verlustbehafteten Kommunikationskanal vollständig zu übertragen, sofern Verluste nur gelegentlich auftreten (transiente Fehler)
- Hierzu muss der Empfänger für jedes erhaltene Paket eine Bestätigungsnachricht (Ackknowledgement, kurz: Ack) schicken
- Achtung: Der Kanal kann Nachrichten und Acks verlieren!
- Um zwischen beiden Fällen korrekt zu unterscheiden, werden je zwei unterschiedliche Sequenznummern und Acks benötigt
- Siehe hierzu auch Vorlesung Telematik 1, Kapitel 4
- Der Empfänger liefert eine Nachricht nur dann aus, wenn er sie das erste Mal erhält (keine Duplikate)
- Erhält der Sender innerhalb eines Timeout-Intervalls nicht das erwartete Ack0 bzw. Ack1, sendet er die Nachricht erneut
- Bei Erhalt eines unerwarteten Ack, wird die aktuelle Nachricht erneut gesendet

Beispiel 2: Das Alternating-Bit-Protokoll (2)

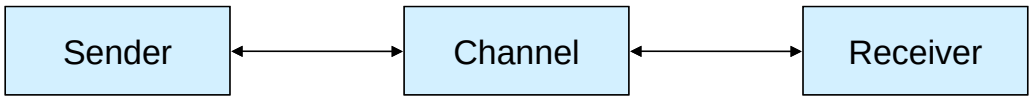


Beispiel 2: Das Alternating-Bit-Protokoll (3)

- Der Protokollautomat auf der vorigen Seite zeigt auch die Nachrichten von bzw. zu der Instanz der höheren Schicht; diese Nachrichten werden wir in der folgenden Implementierung ignorieren
- Wir implementieren stattdessen eine Variante, bei welcher:
  - der Sender zu Beginn eine Liste mit sämtlichen zu sendenden Nachrichten erhält und
  - der Empfänger die erstmals empfangenen Nachrichten einfach auf den Bildschirm ausgibt
- Weiterhin geben wir bei allen Aktionen Statusmeldungen aus
- Für die Modellierung der Verluste im Kommunikationskanal benötigen wir einen Zufallszahlengenerator:
- ```
  -import(rand, [seed/3, uniform/0]).
     rand:seed(exs1024, {23, 13, 97}) % init generator
     RN = uniform()  % returns float in [0.0, 1.0]
  ```
- Bemerkung: In Erlang müssen Timeouts nicht explizit gelöscht werden

Beispiel 2: Das Alternating-Bit-Protokoll (4)

- Systemarchitektur:



- Die drei Prozesse des Systems werden von der Funktion `initialize(ErrorRate, NumberOfMessages)` gestartet
  - Der Sender hat vier Zustandsfunktionen (s.o.); zu Beginn wird die Funktion `senderReady0(List)` gestartet, wobei List eine Liste mit den Zahlen `{1, …, NumberOfMessages}` ist
  - Der Kanal hat nur einen Zustand; er zieht jeweils eine Zufallszahl und verwirft die aktuell empfangene Nachricht, sofern die `Zufallszahl ≤ ErrorRate` ist; andernfalls leitet er sie weiter
  - Der Empfänger hat zwei Zustandsfunktionen; zu Beginn wird `receiverWait0` gestartet
  - Die Funktion initialize wartet nach Prozesserzeugung auf eine `ready`-Nachricht; nach Erhalt sendet sie `stop`-Nachrichten an alle Prozesse


  ```

Beispiel 2: Das Alternating-Bit-Protokoll (5)

```
1  -module(altbit).
2  -export([initialize/2]).
3  -import(rand, [seed/3, uniform/0]).
4
5  for(Max, Max, F) -> [F(Max)]; % for convenience
6  for(I, Max, F) -> [F(I)|for(I0+1, Max, F)].
```


Beispiel 2: Das Alternating-Bit-Protokoll (6)

```
1  initialize(ErrorRate, NumberOfMessages) ->
2    rand:seed({23, 13, 97}), % initialize RNG
3    SendList = for(1, NumberOfMessages, fun(I) -> I end),
4    register(initializer, self()), % others may send us “ready”
5    Receiver = spawn(fun() -> receiverWait0() end),
6    register(receiver, Receiver),
7    Channel  = spawn(fun() -> channelIdle(ErrorRate) end),
8    register(channel, Channel),
9    Sender  = spawn(fun() -> senderReady0(SendList) end),
10   register(sender, Sender),
11   io:format("Started Alt.-Bit Protocol with ~.10B Messages and
12     Error-Rate ~f ~n", [NumberOfMessages, ErrorRate]),
13   receive % wait for Signal that all Messages went ok
14     ready -> io:format("All ~.10B Messages successfully
15                         transmitted. ~n",[NumberOfMessages])
16   end, % Now clean up everything:
17   Sender!stop, Receiver!stop, Channel!stop,
18   unregister(initializer).
```


Beispiel 2: Das Alternating-Bit-Protokoll (7)

```
1  senderReady0([]) -> initializer!ready;
2
3  senderReady0([M|MS]) ->
4    channel!{receiver, {seq0, M}},
5    io:format("Sender: sends Message ~.10B. ~n",[M]),
6    senderProcess0([M|MS]).
```

- Die Implementierung der Funktionen `senderReady1` und `senderProcess1` bleibt dem Leser als Übungsaufgabe überlassen :o)

Beispiel 2: Das Alternating-Bit-Protokoll (8)

```
1  senderProcess0([]) -> initializer!ready; % to be safe
2
3  senderProcess0([M|MS]) ->
4  receive
5    ack0 -> io:format("Sender: received expected Ack for
6                       Message ~.10B. ~n",[M]),
7            senderReady1(MS);
8    ack1 -> io:format("Sender: received unexpected Ack;
9                       Send Again Message ~.10B. ~n",[M]),
10           channel!{receiver, {seq0, M}},
11           senderProcess0([M|MS]);
12   stop -> true
13   after 1000 -> io:format("Sender: Timeout! Send Again
14                            Message ~.10B. ~n",[M]),
15                 channel!{receiver, {seq0, M}},
16                 senderProcess0([M|MS])
17 end.
```

Beispiel 2: Das Alternating-Bit-Protokoll (9)

```
1  channelIdle(ErrorRate) ->
2    RN = uniform(), % determine if msg to be dropped
3    receive
4     {receiver, {Seq, M}} when RN =< ErrorRate -> % drop
5       io:format("Channel: drops Message ~.10B. ~n",[M]),
6       channelIdle(ErrorRate);
7     {receiver, {Seq, M}} when RN > ErrorRate ->  % deliver
8       receiver!{Seq, M},
9       channelIdle(ErrorRate);
10    {sender, M} when RN =< ErrorRate ->          % drop
11      io:format("Channel: drops  ~w ~n",[M]),
12      channelIdle(ErrorRate);
13    {sender, M} when RN > ErrorRate ->           % deliver
14      sender!M,
15      channelIdle(ErrorRate);
16     stop -> true
17   end.
```


Beispiel 2: Das Alternating-Bit-Protokoll (10)

```
1  receiverWait0() ->
2    receive
3      {seq0, M} -> io:format("Receiver: received and
4                              delivers Message ~.10B. ~n",[M]),
5                   channel!{sender, ack0},
6                   receiverWait1();
7      {seq1, M} -> io:format("Receiver: ignores
8                              unexpected Message ~.10B. ~n",[M]),
9                   channel!{sender, ack1},
10                  receiverWait0();
11     stop -> true
12   end.
```

- Die Implementierung der Funktion `receiverWait1` bleibt dem Leser als Übungsaufgabe überlassen :o)

Beispiel 2: Das Alternating-Bit-Protokoll (11)

```
> altbit:initialize(0.45, 3).
Started Alternating Bit Protocol with 3 Messages
and Error-Rate 0.450000
Sender: sends Message 1.
Channel: drops Message 1.
Sender: Timeout! Send Again Message 1.
Channel: drops Message 1.
Sender: Timeout! Send Again Message 1.
Receiver: received and delivers Message 1.
Sender: received expected Ack for Message 1.
Sender: sends Message 2.
Receiver: received and delivers Message 2.
Channel: drops  ack1
Sender: Timeout! Send Again Message 2.
Receiver: ignores unexpected Message 2.
Sender: received expected Ack for Message 2.
Sender: sends Message 3.
Receiver: received and delivers Message 3.
Sender: received expected Ack for Message 3.
All 3 Messages successfully transmitted.
true
```


Fazit zu verteilter Programmierung in Erlang

- Erlang stellt grundlegende Funktionen und Sprachelemente für die Erzeugung von und Kommunikation zwischen Prozessen bereit
  - Dabei wird auch Kommunikation über Rechnergrenzen hinweg unterstützt
  - Somit erlaubt Erlang auf einfache Weise, nebenläufige Programme und Automaten für Kommunikationsprotokolle zu implementieren
- Bei der Implementierung von erweiterten endlichen Automaten in Erlang werden:
  - die Zustände auf separate Funktionen und
  - die Variablen des Automaten auf Parameter der Zustandsfunktionen abgebildet

    → anders kann man sich in Erlang keine Werte merken, die man ggf. auch ändern können muss

Zusammenfassung & Ausblick

- Verteilte Programmierung als wichtige Technik zur Lösung von Aufgaben,
  - bei denen Daten ohnehin verteilt anfallen und auch verteilt bearbeitet werden sollen, oder
  - die so groß sind, dass sie nicht mehr auf einzelnen Rechnern gelöst werden können.
- Es gibt vielfältige Architekturen und Programmiermodelle:
  - Nachrichtenaustausch
  - Verteilte Prozeduraufrufe
  - Publish-Subscribe & Blackboard-Architekturen
  - ...
- In dieser Vorlesung wurde nur der nachrichten-orientierte Ansatz am Beispiel der Programmiersprache Erlang demonstriert.