Freie Universität Berlin
Klaus-Peter Löhr

4 Verteilte Algorithmen

(wird fortgesetzt)

4.1 Notation: Erlang

Verteilter Algorithmus

(distributed algorithm, peer-to-peer algorithm):
Nichtsequentieller Algorithmus, dessen Prozesse ausschließlich über Nachrichten interagieren und zur Erreichung eines gemeinsamen Ziels unter Kenntnis ihrer Partner **kooperieren**.

Kommunikationsoperationen (vgl. 2.4, S. 24):

```
send(message, proc)   oder send(message, proc.port)
recv(mesvar)          oder recv(mesvar, port)
```

Beispiel **Erlang**: *funktionale* Sprache, *dynamisch* getypt (!)

Jeder Prozess hat eine ausgezeichnete, unbeschränkte *mailbox*.

`peer ! expression`
wertet *expression* aus und sendet den Wert an Prozess `peer`

(vereinfacht)

```
receive pattern1 [ when guard1 ] -> expressions1 ;
        pattern2  -> expressions2 ;
        . . . . .
end
```

durchsucht die *mailbox* nach der ersten Nachricht, bei der eine **Musteranpassung** an eines der Muster gelingt *und* der entsprechende *guard* gilt, und wertet dann den zugehörigen Ausdruck aus; *bei Nichtgelingen wird blockiert.*

(vereinfacht)

Variable (variables) beginnen mit Großbuchstaben, z.B. `Message`
Atome (atoms) beginnen mit Kleinbuchstaben, z.B. `urgent`
***Terme** (terms)* sind Atome oder Variable oder:
Tupel (tuples) von Termen, z.B. `{urgent, Message}`
Listen (lists) und weitere .....

Musteranpassung:

Ein *Muster (pattern)* ist ein Term.
Variable in einem Muster werden mit Werten belegt durch

```
pattern = expression
```

oder

```
receive pattern -> ...
```

Bemerkungen:

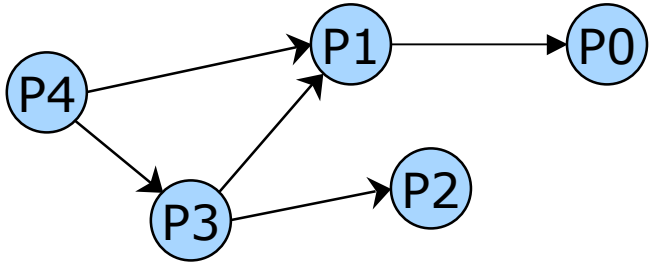
❶ Ein Erlang-Prozess hat zwar nur *eine Mailbox* und *keine Ports*, dies wird aber mehr als aufgewogen durch die Möglichkeiten des `receive` .

❷ *Es gibt „ports"* in Erlang: das sind die aus Kap. 3 bekannten Datenfluss-Ports, die den Anschluss an Pipes des Betriebssystems ermöglichen.

❸ Die Erzeugung von Prozessen in Erlang ist extrem leichtgewichtig (mehr noch als Java Threads).

❹ Das Erlang-System unterstützt in komfortabler Weise auch die *physische Verteilung* eines Programms auf mehrere vernetzte Rechner *(„Distributed Erlang"!).*

Beispiel: **Fluten** eines Prozessnetzes



Sequentiell: Breitendurchlauf eines Graphen

Verteilt:

- Jeder Prozess wartet auf Nachricht;
- wenn ein Prozess eine *neue(!)* Nachricht erhalten hat, leitet er diese an alle ihm bekannten Prozesse weiter,
- verarbeitet sie und wartet auf die nächste Nachricht.
- Der Algorithmus wird gestartet durch Injektion der Nachricht an beliebiger Stelle.

```
pass(Peers) ->   % Funktion beschreibt Prozessverhalten
     receive Info -> propagate(Info, Peers),
                     doWork(Info),
                     pass(Peers)
     after 5000   -> dead
     end .
```

Duplikate werden *nicht* erkannt!

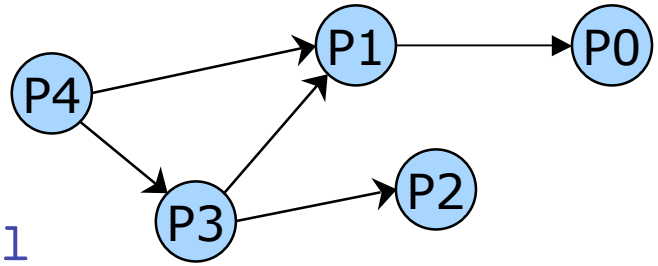
```
propagate(X, [First|Rest]) ->
             First ! X,
             propagate(X, Rest);
propagate(X, []) -> X .

dowork(X) -> io:format("~w in ~w~n", [X, self()]).
 % will write, e.g., "info in <0.37.0>"
```

... und weniger trivial, mit Duplikat-Erkennung:

```
node(Peers, State) ->
     receive {info, Info} -> if State/=Info ->
                                 prop({info,Info}, Peers),
                                 display(Info);
                               true ->
                                 skip
                            end,
                            node(Peers, Info);
             work        -> doWork(State),
                            node(Peers, State);
             shutdown    -> propagate(shutdown, Peers),
                            dead
     end .
```

Verteiltes Programm entsteht durch Vereinbarung der Prozesse in einer Funktion, die dann aufgerufen werden kann (vgl. 2.4, S. 25):




```
-module(pro). % in file pro.erl
-export([node/2, start/1]).
.............. % Code für die Funktionen node etc.
start(Message) ->
     P0 = spawn(pro, node, [[],      noInfo]),
     P1 = spawn(pro, node, [[P0],    noInfo]),
     P2 = spawn(pro, node, [[],      noInfo]),
     P3 = spawn(pro, node, [[P1,P2], noInfo]),
     P4 = spawn(pro, node, [[P1,P3], noInfo]),
     P4 ! Message,
     done .
```

Interaktives Testen:

```
$ erl
Erlang R15B02 (erts-5.9.2) [source] [async-threads:

Eshell V5.9.2  (abort with ^G)
1> c(pro).
{ok,pro}
2> pro:start({info,message}).
info message arrived in <0.42.0>
info message arrived in <0.39.0>
info message arrived in <0.41.0>
info message arrived in <0.38.0>
info message arrived in <0.40.0>
done
3>
```

Übersetzen

Beenden mit ^C a

Bemerkungen:

- Der Algorithmus funktioniert auch für *zyklische* Netze.
  (In Erlang allerdings etwas umständlicher zu formulieren.)
- Es handelt sich tatsächlich um einen *verteilten Algorithmus:*
- die Prozesse „kennen" ihre Partner nicht nur namentlich,
  sie kennen auch deren Verhalten.
- Dieses Verhalten ist *hier identisch mit dem eigenen*
  - nicht ungewöhnlich für verteilte Algorithmen.
- Das Beispiel ist relativ einfach. Typische verteilte
  Algorithmen sind wesentlich kniffliger.

4.2 Beispiel: Verteilter Ausschluss

Verteilter wechselseitiger Ausschluss:

- Prozesse haben kritische Abschnitte,
  z.B. wegen Zugriffs auf gemeinsame Ressourcen
  wie Peripheriegeräte, Funkverbindungen

- Ausschluss verklemmungsfrei und fair

- verschiedene Ansätze:
  - zentraler Koordinator-Prozess
  - Berechtigungs-*Token* wird herumgereicht
  - Prozesse einigen sich

Voraussetzungen:

- Wir betrachten kommunizierende **Stationen**
- ... bestehend aus *Anwendungsprozess* und *Begleitprozess*
- *... mit gemeinsamen Daten*
- Anwendungsprozess befolgt **Protokoll** für Ausschluss
- Begleitprozess erledigt Teil des Protokolls
- jede Station hat eine *Mailbox*

4.2.1 Algorithmus von Ricart/Agrawala

Lösungsidee: Stationen kennen sich und einigen sich über Eintritt

❶ eintrittswillige Station A sendet *Anfrage mit Zeitstempel an alle Partner* und wartet auf Zustimmung aller Partner.

❷ Ein Partner B *stimmt zu*, wenn er selbst nicht an einem kritischen Abschnitt interessiert ist.

❸ Wenn B selbst im kritischen Abschnitt ist, *merkt er sich die Anfrage* und beantwortet sie nach Verlassen des kritischen Abschnitts.

❹ Wenn B aber selbst bereits Anfragen verschickt hat, *entscheiden die Zeitstempel*, ob ❷ oder ❸ erfolgt !

Animation: http://cs.gmu.edu/cne/workbenches/ricart/ricart.html

Protokoll in Pseudocode:

recv(xyz, ...) bedeutet wie in Erlang „entnimm der Mailbox die nächste mit xyz beginnende Nachricht, sobald vorhanden"

```
Clock                    % actual time
Time = . . .             % time of request
N = . . .                % number of stations
Peers = . . .            % set of all stations
Deferred = emptyset      % set of waiting stations
Critical = false         % entering or inside critical section

enter: . . . . .         % prologue of critical section

leave: . . . . .         % epilogue of critical section

helper: . . . . .        % loop body of helper process
```

```
enter:  < Critical = true
          Time = Clock >
          for P ∈ Peers - Me do send(request, Me, Time, P)
          for I ∈ [1..N-1]     do recv(reply)

leave:  < Critical = false >
          for P ∈ Deferred do send(reply, P)
          Deferred = emptyset

helper:  repeat
               recv(request, Requester, T)
               < if Critical and Time<T  then
                     Deferred += Requester
                 else send(reply, Requester)  >
```

[Ricart/Agrawala 1981]

Schönheitsfehler:

Verteilte Systeme kennen *keine globale Zeit*

➔ Problem der *Uhrensynchronisation*

➔ Problem Time==T (evtl. Stationsnummer anhängen!)

➔ Lösung: *virtuelle Zeit* verwenden

Analyse

Aufwand: für jeden kritischen Abschnitt:

2(n-1) Nachrichten

Korrektheit:

❶ Ausschluss garantiert ?

❷ kein Verhungern ?

❶ Ausschluss garantiert ?

Fall 1: A sendet in seinem Prolog eine Anfrage mit Zeit T_A,
B antwortet und tritt dann auch in seinen Prolog ein.

Folgerung: B sendet Anfrage mit $T_B > T_A$, die deshalb
von A erst im Epilog beantwortet wird ✓

Fall 2: A und B sind beide in ihre Prologe eingetreten,
bevor einer der beiden eine Anfrage des anderen
beantwortet hat.

Folgerung: Im Fall $T_B > T_A$ wird B antworten,
im Fall $T_B < T_A$ wird A antworten ✓

❷ kein Verhungern ?

Unter den Prozessen, die ihren Prolog noch nicht abgeschlossen haben, gibt es einen Prozess P mit kleinstem Zeitstempel.

Dessen Anfrage wird daher von allen anderen beantwortet - irgendwann auch vom eventuellen kritischen Prozess.
P kann dann selbst in seinen kritischen Abschnitt eintreten.

Das Gleiche gilt jeweils für den Prozess mit dem nächsten Zeitstempel. ✓

Bemerkung: Die jeweils in ihrem Prolog hängenden Prozesse bilden eine *virtuelle Warteschlange* gemäß den Zeitstempeln.

4.2.2 Einfacher Token Ring

Lösungsidee: Stationen bilden einen Ring, auf dem eine spezielle Marke *(token)* permanent herumgereicht wird. Wer in seinen kritischen Abschnitt eintreten will, greift sich die Marke; beim Austritt schickt er sie wieder auf die Reise (wie beim LAN *Token Ring).*

Kommunikationsoperationen:

send() schickt Marke an den Nachfolger im Ring
recv() empfängt Marke vom Vorgänger

Critical = false
Entry = false
Exit = false

```
enter:  Crtical = true
        P(Entry)

leave:  Critical = false
        V(Exit)

helper: recv()
        if Critical then
            V(Entry)
            P(Exit)
        send()
```

Einer der helper-Prozesse muss hier starten !

Bemerkungen:

- *Ausschluss* ist gesichert, da nur eine Station im Besitz der Marke sein kann.
- *Verhungern* ist ausgeschlossen, da die Marke nach jedem kritischen Abschnitt weitergereicht wird.
- *Fairness:* höchstens n-1 Stationen können sich vordrängeln
- Die *heftige Kommunikation* kann dadurch abgemildert werden, dass die helper-Prozesse Pausen einlegen

[Kapitel 4 wird fortgesetzt]