**Team**: 6, Mert Siginc, Michael Müller

**Aufgabenaufteilung**:

1. <Aufgaben, für die Teammitglied 1 verantwortlich ist>,   
   <Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden>
2. <Aufgaben, für die Teammitglied 2 verantwortlich ist>,   
   <Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden>

**Quellenangaben**:

* <http://erlang.org/doc/apps/stdlib/index.html>
* <http://users.informatik.haw-hamburg.de/~klauck/verteiltesysteme.html>

**Bearbeitungszeitraum**:

|  |  |
| --- | --- |
| Michael Müller | Mert Signic |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Aktueller Stand**: <Welche Teile der Software sind fertig inklusive Tests, welche sind fertig, aber noch nicht getestet, welche müssen noch implementiert werden>

**Änderungen des Entwurfs**: <Vor dem Praktikum auszufüllen: Welche Änderungen sind bzgl. des Vorentwurfs vorgenommen worden.>

Betrifft ein erfolgreicher Vote alle?

Gibt es genau einen Vote oder hat evtl. jeder ggT-Prozess einen eigenen Vote?

**Entwurf**:

**Um was geht es?**

In der Aufgabe geht es darum, einen größten gemeinsamen Teiler (ggt) verteilt zu errechnen.

Diese Arbeit werden N ggT-Prozesse mit je einer Initialzahl erledigen, wobei N >= 3, da erst ab 3 Elementen ein Ring mit linken und rechten Nachbar erzeugt werden kann. Die Koordination des Systems übernimmt der gleichnamige Koordinator. Mit seiner Hilfe, kann man auf das System ein wirken (Starten der Berechnung, stoppen, Neue Berechnung, Herunterfahren, …). Die Starter helfen beim Starten der ggT-Prozesse und erfüllen keinen anderen Zweck. Der Nameservice hilft hierbei (ähnliche wie DNS) von einer Pid zu dem registrierten Namen zu kommen. Und warum ist das wichtig?

Bei allen Schnittstellen sind bei synchroner Kommunikation die Antwortnachrichten bei dem Auftraggeber nicht aufgeführt, so sind z.B. die Antwortnachrichten des Namensdienstes nur beim Namensdienst aufgeführt! From soll eine direkt zum senden verwendbare Prozess-ID sein (self()).

* also die Logs der Antworten?

**Wie sieht die Architektur aus?**

Peer-to-Peer Kommunikation:

ggT-Prozess zu ggT-Prozess

Client-Server (in der Reihenfolge) Kommunikation:

ggT-Prozess zu Koordinator

ggT-Prozess zu Nameservice

Koordinator zu Nameservice

**Wie sehen die Komponenten aus?**

Wie sieht der ggT-Prozess aus?

Wie sieht der Starter aus?

Wie sieht der Koordinator aus?

**Wie sehen die Schnittstellen aus?**

**Nameservice:**

Binden eines Dienstes (einmaliges binden):

register(meindienst,From),

Nameservice ! {From,{bind,meindienst,node()}},

receive ok -> io:format("..bind.done.\n");

in\_use -> io:format("..schon gebunden.\n")

end,

Rebinden eines Dienstes (erstmaliges oder wiederholtes binden):

register(meindienst,From),

Nameservice ! {From,{rebind,meindienst,node()}},

receive ok -> io:format("..rebind.done.\n")

end,

Lookup für einen Dienst:

Nameservice ! {From,{lookup,meindienst}},

receive

not\_found -> io:format("..meindienst..not\_found.\n");

{pin,{Name,Node}} -> io:format("...ok: {~p,~p}.\n",[Name,Node])

end,

Unbind eines Dienstes:

Nameservice ! {From,{unbind,meindienst}},

receive

ok -> io:format("..unbind..done.\n")

end,

unregister(meindienst),

Multicast an alle registrierten Einheiten:

Nameservice ! {From,{multicast,vote,meinname}},

receive

{From,{vote,meinname}} ->...From ! {voteYes,Clientname}...,

end

Reset des Namensdienst:

Nameservice ! {From,reset},

receive

ok -> do\_something\_else,

end

**Koordinator:**

{From,getsteeringval}: PID X Tupel -> void

Die Anfrage nach den steuernden Werten durch den Starter Prozess.

{hello,Clientname}: Atom X Atom - > void

Ein ggT-Prozess meldet sich beim Koordinator mit Namen Clientname an.

{briefmi,{Clientname,CMi,CZeit}}: Atom X Tupel (Atom X Nummer X Nummer) -> void

Ein ggT-Prozess mit Namen Clientname (keine PID!) informiert über sein neues Mi CMi um CZeit Uhr (CZeit wird mittels util:timeMilliSecond() erstellt).

{From,briefterm,{Clientname,CMi,CZeit}}: PID X Atom X Tupel (Atom X Nummer X Nummer) -> void

Ein ggT-Prozess mit Namen Clientname und Absender From informiert über über die Terminierung der Berechnung mit Ergebnis CMi um CZeit Uhr.

reset: Atom -> void

Der Koordinator sendet allen ggT-Prozessen das kill-Kommando und bringt sich selbst in den initialen Zustand, indem sich Starter wieder melden können.

step: Atom -> void

Der Koordinator beendet die Initialphase und bildet den Ring. Er wartet nun auf den Start einer ggT-Berechnung.

prompt: Atom -> void

Der Koordinator erfragt bei allen ggT-Prozessen per tellmi deren aktuelles Mi ab und zeigt dies im log an.

nudge: Atom -> void

Der Koordinator erfragt bei allen ggT-Prozessen per pingGGT deren Lebenszustand ab und zeigt dies im log an.

toggle: Atom -> void

Der Koordinator verändert den Flag zur Korrektur bei falschen Terminierungsmeldungen.

{calc,WggT}: Tupel (Atom, Nummer) -> void

Der Koordinator startet eine neue ggT-Berechnung mit Wunsch-ggT WggT.

kill: Atom -> void

Der Koordinator wird beendet und sendet allen ggT-Prozessen das kill-Kommando.

**Starter:**

{steeringval,ArbeitsZeit,TermZeit,Quota,GGTProzessnummer}:

Tupel (Atom X Nummer X Nummer X Nummer X Nummer) -> void

die steuernden Werte für die ggT-Prozesse werden im Starter Prozess gesetzt; Arbeitszeit ist die simulierte Verzögerungszeit zur Berechnung in Sekunden, TermZeit ist die Wartezeit in Sekunden, bis eine Wahl für eine Terminierung initiiert wird, Quota ist die konkrete Anzahl an notwendigen Zustimmungen zu einer Terminierungsabstimmung und GGTProzessnummer ist die Anzahl der zu startenden ggT-Prozesse.

**ggT-Prozess:**

{setneighbors,LeftN,RightN}: Tupel (Atom X Atom X Atom) -> void

die Namen des linken und rechten Nachbarn werden gesetzt.

{setpm,MiNeu}: Tupel (Atom X Nummer) -> void

die von diesem Prozess zu bearbeitenden Zahl für eine neue Berechnung wird gesetzt.

{sendy,Y}: Tupel (Atom X Nummer) -> void

der rekursive Aufruf der ggT Berechnung.

{From,{vote,Initiator}}: Tupel (PID X Tupel (Atom X Atom)) -> void

Wahlnachricht für die Terminierung der aktuellen Berechnung; Initiator ist der Initiator dieser Wahl und From ist sein Absender.

{voteYes,Name}: Tupel (Atom X Atom) -> void

erhaltenes Abstimmungsergebnis, wobei Name der Name des Absenders ist.

{From,tellmi}: Tupel (PID X Atom) -> From ! {mi,Mi}

Sendet das aktuelle Mi an From. Wird vom Koordinator z.B. genutzt, um bei einem Berechnungsstillstand die Mi-Situation im Ring anzuzeigen.

{From,pingGGT}: Tupel (PID X Atom) -> From ! {pongGGT,GGTname}

Sendet ein pongGGT an From. Wird vom Koordinator z.B. genutzt, um auf manuelle Anforderung hin die Lebendigkeit des Rings zu prüfen.

kill: Atom -> void

der ggT-Prozess wird beendet.

**Welche Daten müssen gespeichert werden?**

Koordinator:

Flag (Boolean) das angibt ob ein (evtl. falsches) Ergebnis der Berechnung am Ende korrigiert werden soll.

Aktuell minimalster ggT (Nummer) in der Berechnung.

Ring aus allen bekannten ggT-Prozessnamen (siehe ADT).

Die Steeringvalues für die Starter

ggT-Prozess:

Der aktuelle Mi (Nummer).

Der Clientname (Atom), besteht aus:

1. Praktikumsgruppennummer
2. Teamnummer
3. Nummer des ggT-Prozesses (aus der Sicht des Starters, nicht global!)
4. Nummer des Starters

Anzahl der aktuell bekommenen „voteYes“ Nachrichten.

**Wie sehen die Abstrakten Datentypen (ADT) aus?**

Der Ring im Koordinator:

Ist eine unsortierte Liste und hält als Elemente die bekannten ggT-Prozessnamen. Am Anfang wird der Ring einmal gebaut (den ggT-Prozessen die Nachbarn mitgeteilt) und danach nicht mehr verändert.

**Wie sieht der Ablauf aus?**

Der Ring wird wie folgt gebaut:

1. Alle bekannten ggT-Prozessnamen (die sich zurückgemeldet haben) sind in einer unsortierten Liste, kein Name ist doppelt. Die Nachbarn pro ggT-Prozessname werden wie folgt festgelegt und dem ggT-Prozess mitgeteilt:
   1. Die Nachbarn des ersten Elements sind:
      1. Das letzte Element (Linker Nachbar)
      2. Das 2. Element (Rechter Nachbar)
   2. Die Nachbarn des letzten Elements sind:
      1. Das erste Element (Rechter Nachbar)
      2. Das vorletzte Element (Linker Nachbar)
   3. Für die restlichen Elemente (2. Element bis vorletztes Element) sind die Nachbarn:
      1. Element davor (Linker Nachbar)
      2. Element dahinter (Rechter Nachbar)
2. Festgelegt werden die Nachbarn mit Hilfe der „setneighbors“ Schnittstelle des jeweiligen ggT-Prozesses. Hierzu wird mit Hilfe des Nameservices („lookup“) die ggT-Prozess ID zum bekannten ggT-Prozessnamen herausgefunden. Damit kann dann die ggT-Prozess Schnittstelle angesprochen werden. Die Nachbarn werden mit ihrem ggT-Prozessnamen übergeben.
3. Sind die Nachbarn erst einmal bekannt ist der Ring gebaut und Bereit für die Berechnung. Da wir nicht davon ausgehen, dass Nodes in unserem Beispiel abstürzen ist es nicht nötig den Ring ggf. noch einmal zu bauen.

Berechnet wird wie gefolgt:

1. Alle ggT-Prozessen wird ein neues Mi mitgeteilt („setpm“).
2. Ein ggT-Prozess bekommt ein neues Y mitgeteilt („sendy“).
   1. Zuerst wird eine gewisse Zeit gewartet (siehe steeringvalues, Verzögerungszeit).
   2. Darauhin berechnet der ggT-Prozess ein neues Mi anhand folgenden Pseudocodes:

if (y < Mi) then

NeuesMi = mod(Mi – 1, y) + 1

1. Ist ein neues Mi entstanden wird es an die Nachbarn gesendet („sendy“), informiert den Koordinator („briefmi“) und ersetzt das alte Mi.
2. Ist kein neues Mi entstanden so wird nichts getan.

Wie genau starten?

Welche Phasen gibt es mit welchen Eigenschaften (nicht in der Reihenfolge)?

* Initialisierung
* Berechnungsbereit
* Berechnung
* Ergebnis
* Herunterfahren