**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**:

**Aktueller Stand**:   
Alles fertig implementiert und getestet.

Bestimmte Kombinationen (mit unserem und Professors Code) erstellt und keine Probleme mehr bekannt.

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

**Entwurf**:

**Allgemeine Funktion der Anwendung:**

Die Anwendung realisiert ein verteiltes System. In diesem sollen von beliebig vielen Prozessen, die jeweils ein #Mi-Wert haben, der ggT(größte gemeinsamer Teiler) verteilt berechnet werden. Für die Berechnung wird der Satz des Euklid verwendet.

Zur Realisierung dieser Aufgabe sollen die Prozesse vom Koordinator zufällig zu einem Ring aufgestellt werden, wobei jeder einzelne Prozess jeweils seinen linken und seinen rechten Nachbarn kennt. Dabei muss es mindestens 3 Prozesse geben, da erst ab 3 Prozessen ein Ring mit linken und rechten Nachbarn erzeugt werden kann.   
Dies ist wichtig, da der #Mi-Wert sich nach einer Berechnung verändern kann und die Prozesse teilen dies ihren Nachbarn und dem Koordinator mit, so dass der ggT gefunden wird.

Die Koordination des Systems übernimmt der gleichnamige Koordinator. Mit seiner Hilfe kann man auf das System einwirken (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.

**Wie sieht die Architektur aus?**

Peer-to-Peer Kommunikation:

Unsere Architektur zu unserem Entwurf sieht vor, dass wir 2 Architektur-Arten haben.

Zu einem eine P2P-Architektur, welches unsere Prozesse nutzen um mit ihren Nachbarn direkt zu kommunizieren.

Und eine Client-Server Architektur, welches unteranderem vom Starter, Koordinator, Namensdienst und den ggT-Prozessen genutzt wird, um Informationen mit dem Server, in diesem Fall der Koordinator und der Namensdienst, auszutauschen.

Wie genau starten?

* NS, KO starten
* Starter starten
  + Starter starten ggT-Prozesse, die sich wiederum bei NS und KO melden.
  + Konkrete für anfragender Starter ist: Aktuell gemeldete Starter \* GGTProzessAnzahl \* Quote

Koordinator-Befehle:

* step
  + KO registriert keine ggT-Prozesse mehr
  + KO baut kreis
  + KO ready für Berechnung
* calc
  + KO stößt Berechnung an
* kill
  + Sendet einen kill-Befehl an alle Prozesse

**Koordinator:**

-Allgemein-

Der Koordinator hat die Aufgabe den Startern die steuernden Werte zu übergeben und den Prozessen deren M-Wert zu übergeben.

Hiernach startet der Koordinator die Berechnung.

Der Koordinator kann zudem alle Prozesse beenden.

Der Koordinator wird über manuelle Eingaben gesteuert.

-Schnittstellen-

{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 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.

-Konfigurationsdatei-

koordinator.cfg:

arbeitszeit

* + Nummer
  + Gibt in Sekunden an wie lang der ggT-Prozess bei der Berechnung eines neuen Mis Warten wird.

termzeit

* + Nummer
  + Gibt in Sekunden an wie lange (nach der letzten Nachricht) ein ggT-Prozess wartet bis er eine Abstimmung anfängt.

ggtprozessnummer

* + Nummer
  + Anzahl der zu startenden ggT-Prozesse pro Starter.

nameservicenode

* + Atom
  + Node vom Nameservice

koordinatorname

* + Atom
  + Registrierter Name des Koordinator Prozesses

quote

* + Nummer
  + Gibt in Prozent an, wie viele ggT-Prozesse sich zu einem Vote positiv rückmelden müssen damit der Vote erfolgreich ist. Wird im Koordinator in eine konkrete Zahl umgerechnet, wenn Starter beim Koordinator nach „getsteeringval“ fragt.

korrigieren

* + Boolean
  + Gibt an, ob ein falsches Ergebnis der Berechnung korrigiert werden soll (true) oder nicht (false).

-Benötigte Daten-

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.

-Ring ADT-

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

-Realisierung-

Zu Beginn liest der Koordinator seine spezifische Konfigurationsdatei ein. Hiernach hat der Starter die Möglichkeit die steuernden Werte vom Koordinator zu erfragen und die ggT-Prozesse haben die Möglichkeit sich beim Koordinator zu registrieren. Wird der Koordinator Manuel mit der Eingabe von „step“ auf bereit gesetzt, kann man sich beim Koordinator nicht mehr registrieren und auch keine steuernden Werte mehr erfragen. Zudem werden alle vorhandenen Prozesse werden im Zufall zu einem Ring angeordnet. Um die registrierten Prozesse zu verwalten würde sich eine Liste anbieten, die sich bei jeder Registrierung erweitert. Der Ring wäre auch eine Liste, welche sich zufällig aus der Prozessliste generiert werden würde. Dabei würde man Zufallszahlen generieren welche maximal der Länge der Prozessliste betragen würden. Das Element aus der Prozessliste mit dem generierten Index würde man in die Ringliste übertragen, so dass Prozesse zufällig übertragen werden.

Zudem werden zu allen Prozessen die jeweiligen Nachbarn zugeordnet und diese über die Schnittstellen der Prozesse übermittelt. Dabei wird in der Ring-Liste nach den jeweiligen Nachbarn geschaut. Dabei ist zu beachten, dass das erste Element unteranderem das letzte Element in der Liste als Nachbarn hat.

Wird dann der Manuele Befehl „calc“ eingeben, werden allen Prozessen ein spezifischen Mi-Wert übermittelt. Dieser Mi-Wert wird über die gegebene vsutil erzeugt.

Hiernach werden bei 20% jedoch min. 2 Prozesse, über die Schnittstelle(sendy),   
die Berechnung angestoßen (Wie? Das war ja unser Gespräch: entweder senden die Ihr Mi an die Nachbarn oder der Koordinator hat ne seperate Zahl…).

Jederzeit könnte ein ggt-Prozess den Koordinator über seine Terminierung informieren. Ist dies der Fall, so wird überprüft, ob die erhaltenen Information, welches den Mi-Wert des Prozesses beinhaltet, größer ist als die

bereits bekannte Zahl (Und wie ist die bekannt? idk Glaube ist der gewünschte GGT).

Ist dies der Fall und in den Informationen ist ein spezielles Flag gesetzt, so wird die kleinste Zahl an diesen Prozess per „sendy“ übergeben.

Durch die Manuele Eingabe von „kill“ wechselt der Koordinator in den Zustand beenden, wobei er dann allen Prozesses ein Kill-Kommando sendet. (Findest du das so besser?)

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.

**Starter:**

-Allgemein-

Hilft dem System bzw. dem Koordinator eine Vielzahl an ggT-Prozessen zu starten. Dabei ist definiert (siehe Konfigurationsdatei) wie viele ggT-Prozesse pro Starter gestartet werden.

-Schnittstellen-

{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.

-Konfigurationsdatei-

Siehe Konfigurationsdatei des ggT-Prozesses.

**ggT-Prozess:**

-Allgemein-

Alle ggT-Prozesse durchlaufen den gleichen Code und verrichten die eigentliche Arbeit in dem sie verteilt (mit asynchronen Nachrichten) den ggT berechnen. Haben alle zu einem definierten Ende der Berechnung gefunden, werden die einzelnen ggTs der einzelnen Prozesse ggf. korrigiert und warten dann auf eine neue Berechnung. Bis sie „gekillt“ werden.

-Schnittstellen-

{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.

-Konfigurationsdatei-

ggt.cfg

praktikumsgruppe

* + Nummer
  + Gibt an in welcher Praktikumsgruppe sich das Team befindet.

teamnummer

* + Nummer
  + Gibt an in welchem Team die Studenten (oben genannt) zu finden sind.

nameservicenode

* + atom
  + Node vom Nameservice

koordinatorname

* + Registrierter Name des Koordinator Prozesses

-Benötigte Daten-

Der aktuelle Mi (Nummer).

Der Clientname (Atom), besteht aus:

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

Anzahl der aktuell bekommenen „voteYes“ Nachrichten.

-Realisierung-

Der ggt-Prozess wird vom Starter erzeugt. Nach der Erzeugung durch den Starter, welches jedem Prozess einen eindeutigen Namen zuweist, meldet sich der ggt-Prozess beim Koordinator und beim Namendienst und teilt ihm seinen Namen mit. Zuzüglich registriert er sich lokal mit seinem Namen als Erlang-Node.

Hiernach erwartet er vom Koordinator seine Nachbarn und danach seine Mi-Zahl und ist ab dann bereit eine Berechnung zu starten. Die Nachbarn werden über die Schnittstelle {setneighbors,LeftN,RightN} übergeben. Die Mi-Zahl über die Schnittstelle {setpm,MiNeu}.

Die Berechnung wird über den Koordinator oder über andere ggT-Prozesse angestoßen.   
Über die Schnittstelle {sendy,Y} wird die Berechnung angestoßen. (siehe unten „Berechnet wird wie folgt“)

Kommt es zu einer Berechnung ändert sich der Mi-Wert des Prozesses, welches der Prozess dann an den Koordinator und an seine Nachbarn über deren Schnittstellen mitteilt.

Bei jeder Berechnung wird eine gewisse Zeit gewartet um eine längere Berechnung zu Simulieren. Wird eine gewisse Zeit keine Berechnung mehr angestoßen, so wird vom Prozess ein Voting gestartet indem er die Schnittstelle des Namendienstes anspricht. Wird eine gewissen Quote an Zustimmungen erhalten, informiert der Prozess den Koordinator über die Terminierung der Berechnung. (Müssen Fragen was genau hier passieren soll)

**Wie sieht der Ablauf aus?**

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.

Ein Vote (Aktiv) läuft wie folgt ab:

1. Ein ggT-Prozess hat seit „termzeit“ keine Nachricht mehr bekommen.
2. Daraufhin stößt er ein Vote an (multicast, vote, an Nameservice).
3. Er wartet nun auf genügend „voteyes“ antworten anderer ggT-Prozesse
   1. Sind genügend eingetroffen:
      1. Meldet er dem Koordinator „briefterm“.
      2. Loggt die Anzahl der eingegangenen voteYes Meldungen (Also nicht einfach abbrechen nachdem quote erreicht? Dann müsste man ja eine gewisse Zeit warten, wie lang Sinnvoll? Idk fragen wir in der VL)
      3. Behält jedoch vorerst noch sein Mi, da das Ergebnis ggf. noch korrigiert werden muss.
      4. Ist aber auch bereit für eine komplett neue ggT Berechnung
   2. Sind nicht genügend eingetroffen:
      1. So geht er wieder in den normalen Berechnungszustand über.

Ein Vote (Passiv) läuft wie folgt an:

1. Ein anderer ggT-Prozess schickte per „multicast, vote“ über den Nameservice an alle ggT-Prozesse die Abstimmungsaufforderung.
2. Hat dieser ggT-Prozess seit „termzeit“ / 2 keine Nachrichten mehr bekommen antwortet er mit „voteYes“.
3. Danach geht der wieder in den normalen Berechnungszustand über.

**Wohin mit den Fragen & deren Antworten?**

* Betrifft ein erfolgreicher Vote alle?

Ich denke nur den Initiator , ist eine gewisse quote erreicht, werden alle kommenden votes ab dann positiv, die Iniatoren davor müssen zum nächsten Vote warten. So denke ich.

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

Oben eingegangen, jeder hat einen eigenen, da wenn der vote positiv verläuft nur der Koordinator davon weiß. Ohne Gewähr.

* Wenn wir immer nur aktuelles Minimum im System zwischen speichern kann das am Ende dennoch größer als der WggT sein, soll das so?

Kommt drauf an, was ist der Initial-Wert hier? 0 geht nicht, da nach kleiner geschaut wird und zb. 99999 macht auch wenig Sinn, d.h. man müsste den gewünschten GGT als initialen-Wert nehmen und dann kann der nie größer sein, sondern nur kleiner.

* Wenn wir zu jeder Zeit eine neue Kalkulation anstoßen können, aber gerade noch sendy-Nachrichten mindestens eines Nachbarn unterwegs sind (weil der neues Mis errechnet hat) verfälscht dies das Ergebnis! Wie damit umgehen?
* Ich glaube dafür ist das reset, welches alle laufenden prozesse beendn soll und dann kann man mit toggle eine neue Berechnung anstoßen.

Wann ist das System wirklich mit der Berechnung fertig? Wenn alle ggT-Prozesse den richtigen ggT errechnet haben (mit oder ohne Korrigieren) und jeder ggT-Prozess einen eigenen erfolgreichen Vote angestoßen hat?

Würde sagen garnicht, mann soll mit nudge erragen ob sie noch rechnen

Nicht genutze elemente

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?

**Gliederung: (wird in den Überschrifften nicht übernommen & nicht so wichtig)**

1. Um was geht es?
2. Wie sieht die Architektur aus?
3. Wie sehen die Komponenten aus?
   1. Nameservice
      1. Allgemein
      2. Schnittstellen
   2. Koordinator
      1. Allgemein
      2. Schnittstellen
      3. Konfigurationsdatei
      4. Benötigte Daten
      5. Ring ADT
   3. Starter
      1. Allgemein
      2. Schnittstellen
      3. Konfigurationsdatei
   4. ggT-Prozess
      1. Allgemein
      2. Schnittstellen
      3. Konfigurationsdatei
      4. Benötigte Daten
4. Wie sieht der Ablauf aus?
   1. Erstellung des Rings
   2. Berechnung
   3. Vote (Aktiv)
   4. Vote (Passiv)

**Wie sehen die Komponenten aus?**

**Nameservice: (Brauchen wir nicht, meiner Meinung nach reicht es wenn wir sagen wir nutzen die schnittstellen)**

-Allgemein-

Der gegebene Nameservice wird dem System helfen, eine gegebene Prozess ID zu seinem registrierten Namen aufzulösen. Da vom Professor gegeben wird es nicht weiter spezifiziert, weil wir die „Innereien“ und die konkreten Abläufe nicht wissen müssen.

-Schnittstellen-

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