Entwurf VSP4

Team: 10 (Antoni Romanski, Benjamin Schröder)

Inhaltsverzeichnis

Aufgabenaufteilung:

Der Entwurf wurde gemeinsam erarbeitet und die Implementation wurde wie folgt aufgeteilt:

Datei	Bearbeitet durch
`cbCast.erl`	Benjamin
`vectorC.erl`	Antoni

Quellenangaben:

Folien und Skript, sowie Vorlesungsmitschriften von Prof. Dr. Christoph Klauck.

Bearbeitungszeitraum:

- Gemeinsam: 06.06. (1 Stunde), 20.06. (8 Stunden), 21.06. (6 Stunden), 22.06. (6 Stunden), 23.06 (2 Stunden), 24.06. (4 Stunden)
- Antoni: 08.06 (4 Stunden), 09.06 (3 Stunden), 10.06 (6 Stunden), 15.06 (4 Stunden), 19.06. (4 Stunden)
- Benjamin: 06.06. (1 Stunde), 07.06. (6 Stunden), 08.06. (4 Stunden 30 Minuten), 09.06. (1 Stunde), 10.06. (7 Stunden)

Projektstruktur

Datei	
`cbCast.erl`	Kommunikationseinheiten, welche Nachrichten unter sich verschicken
`vectorC.erl`	Vektoruhr, welche die kausale Rheinfolge sicherstellt
`towerClock.erl`	Zentraler Server der Vektoruhr
`towerCBC.erl`	Ungeordneter Multicast
`testCBC.beam`	führt Tests mit System durch

Dieses Projekt implementiert ein Kommunikationscluster, welches einen kausalen Multicast mittels Vektoruhren durchführen kann.

Ein Cluster besteht aus mehreren Kommunikationseinheiten, welche einander Nachrichten schicken.
Jede Kommunikationseinheit besitzt eine Vektoruhr, welche Ereignisse in einem Vektor speichert. Um alle Vektoruhren im Kommunikationscluster synchron zu halten, gibt es eine Vektoruhrzentrale.

Der Multicast selbst wird nicht von den Kommunikationseinheiten durchgeführt. Diese instrumentieren lediglich einen separaten Prozess (`towerCBC`) eine Nachricht an andere Kommunikationseinheiten im Cluster zu schicken.

Komponenten

Kommunikationseinheit

Die Kommunikationseinheit implementiert die Funktionalität, die für die eigentliche Kommunikation notwendig ist. Hier werden die eigentlichen Nachrichten gesendet und empfangen.

Generelle Anforderungen

1. Die Implementation dieser Komponente muss in einer Datei `cbCast.erl` erfolgen.

2. Kontaktinformationen für den ungeordneten Multicast befinden sich in einer Konfigurationsdatei `towerCBC.cfg`. Parameter in der Datei: `servername`, `servernode`

3. Die zu versendenen Nachrichten sind Zeichenketten.

4. Zum Sicherstellen des kausalen Multicasts gibt es in jeder Kommunikationseinheit sowohl eine Holdbackqueue (HBQ) als auch eine Deliveryqueue (DLQ), durch welche entschieden wird, wann welche Nachrichten ausgeliefert werden dürfen.

5. Aufbau der DLQ (neue Elemente werden hinten eingefügt):
   ```
   [{<Message1>, <MessageVT1>}, ..., {<MessageN-1>, <MessageVTN-1>}, {<MessageN>, <MessageVTN>}]
   ```
   Aufbau der HBQ (neue Elemente werden vorne eingefügt):
   ```
   [{<MessageN>, <MessageVTN>}, ..., {<Message2>, <MessageVT2>}, {<Message1>, <MessageVT1>}]
   ```

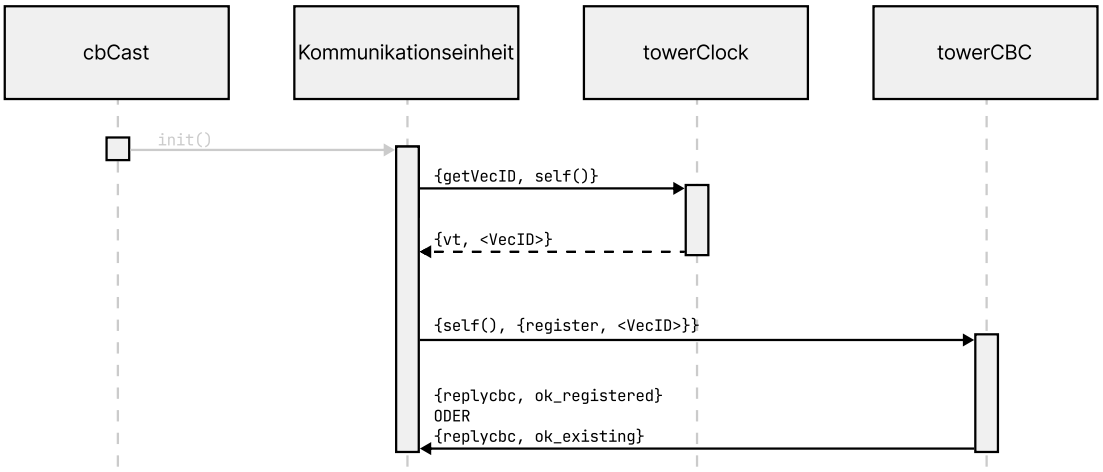
Schnittstellen für den Anwender

6. `init()`
   startet den Prozess für eine neue Kommunikationseinheit

   #### Parameter:
   keine

   #### Rückgabe:
   Prozess-ID (PID) einer Kommunikationseinheit

   #### Ablauf



Implementation:

6.1. Starten eines neuen Prozesses. Dabei die `PID` des erstellten Prozesses merken.

6.2. Ausführen von `start()` in dem erstellten Prozess

6.3. Rückgabe von `PID`

7. `stop(CommPID)`
beendet eine Kommunikationseinheit

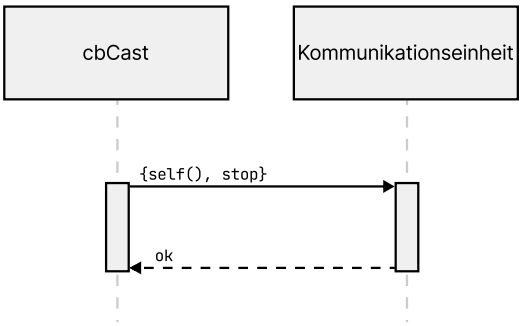
Parameter:

`CommPID`: Prozess-ID der zu beendenden Kommunikationseinheit

Rückgabe:

`done`

Ablauf



Implementation:

7.1. Senden von `{self(), stop}` an `CommPID`

7.2. Warten auf Bestätigung `ok`

7.3. Rückgabe von `done`

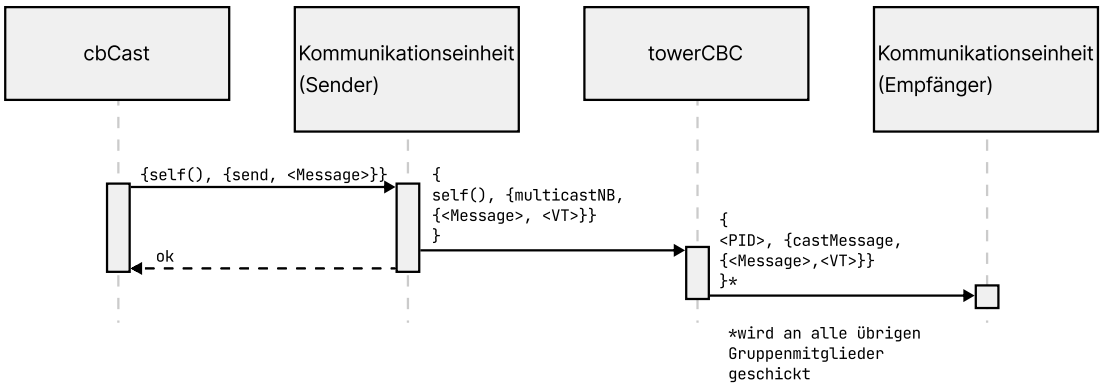
8. `send(CommPID, Message)`
initiiert einen kausalen Multicast an alle Kommunikationseinheiten mit einer angegebenen Nachricht

Parameter:
`CommPID` : Prozess-ID der Kommunikationseinheit, welche den Multicast auslöst

`Message` : die zu versendende Nachricht

Rückgabe:
`ok`

Ablauf



Implementation
8.1. Senden von `{self(), {send, <Message>}}` an `CommPID`

8.2. Warten auf Bestätigung `ok`

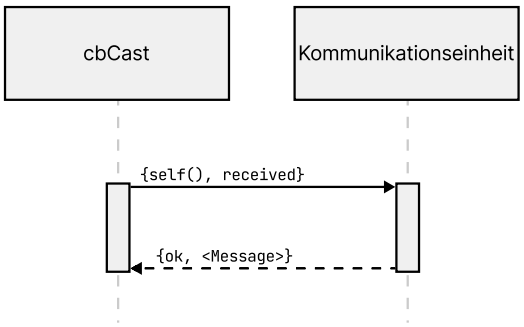
8.3. Rückgabe von `ok`

9. `received(CommPID)`
empfängt eine Nachricht (blockierend)

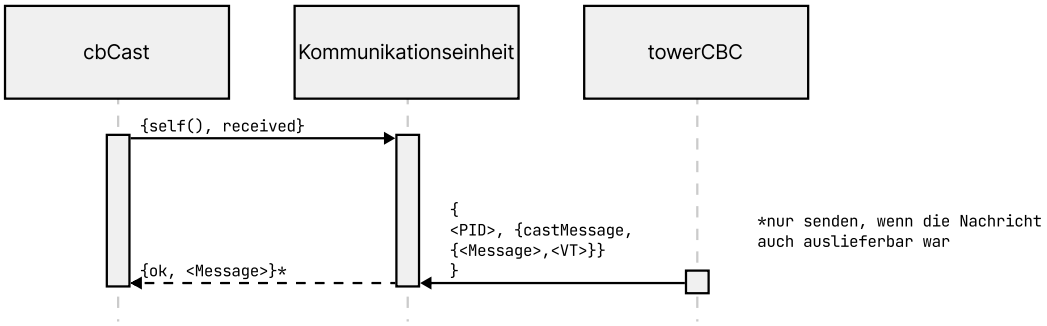
Parameter:
`CommPID` : Prozess-ID der Kommunikationseinheit, von welcher die nächste Nachricht empfangen werden soll

Rückgabe:
empfangene Nachricht als Zeichenkette

Ablauf
Bei `received` können zwei Szenarien eintreten.



Szenario 1: Es ist bereits eine auslieferbare Nachricht vorhanden



Szenario 2: Es ist keine auslieferbare Nachricht vorhanden und der Prozess blockiert, bis eine solche eingetroffen ist

Implementation

9.1. Senden von `{self(), received}` an `CommPID`

9.2. Warten auf Bestätigung `{ok, <Message>}`

9.3. Rückgabe von `Message`

10. `read(CommPID)`
empfängt eine Nachricht (nicht blockierend)

Parameter:

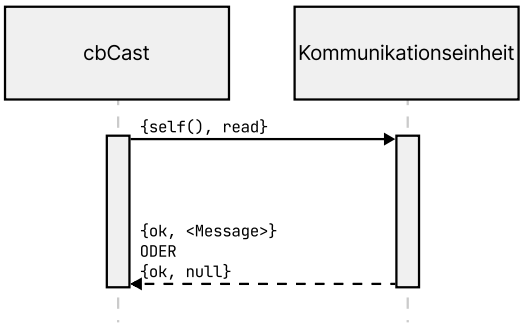
`CommPID`: Prozess-ID der Kommunikationseinheit, von welcher die nächste Nachricht empfangen werden soll

Rückgabe:

empfangene Nachricht als Zeichenkette

`null`, wenn keine Nachricht vorhanden war

Ablauf



Implementation

10.1. Senden von `{self(), read}` an `CommPID`

10.2. Warten auf Bestätigung `{ok, <Message>}`

10.3. Rückgabe von `Message` (kann hier sowohl eine Zeichenkette als auch `null` sein)

Hilfsfunktionen

11. `start()`
initialisiert eine Kommunikationseinheit

Parameter:

keine

Rückgabe:

`ok`

Implementation:

11.1. Auslesen der Kontaktinformationen `TowerCBC` für die `towerCBC`-Komponente aus `towerCBC.cfg`

11.2. Erstellen einer eigenen Vektoruhr `MyVT = {<VecID>, <Vektor>}` mithilfe von `initVT()`

11.3. Registrieren des eigenen Prozesses beim `towerCBC`, indem die Nachricht `{<PID>, {register, <VecID>}}` an diesen gesendet wird

11.4. Warten auf Bestätigung `{replycbc, ok_registered}` oder `{replycbc, ok_existing}`

11.5. Initialisieren der eigenen `DLQ` und `HBQ` mit den Hilfsfunktionen `initHBQ()` und `initDLQ()`

11.6. Aufrufen von `loop(MyVT, DLQ, HBQ, TowerCBC)`

12. `loop(VT, DLQ, HBQ, TowerCBC)`
    hält eine Kommunikationseinheit am Leben und empfängt dabei Nachrichten, um die internen Schnittstellen anbieten zu können.

    ### Parameter:
    `VT`: Vektorzeitstempel der aktuellen Kommunikationseinheit

    `DLQ`: DLQ der aktuellen Kommunikationseinheit

    `HBQ`: HBQ der aktuellen Kommunikationseinheit

    `TowerCBC`: Kontaktinformationen für den ungeordneten Multicast

    ### Rückgabe:
    `ok`

    ### Implementation:
    12.1. Empfangen von Nachrichten in einer Endlosschleife, solange bis die Nachricht `stop` erhalten wird

    12.2. Je nach empfangener Nachricht die Implementierung der jeweiligen internen Schnittstelle aufrufen. Sollte eine Nachricht empfangen werden, zu der es keine interne Schnittstelle gibt, soll eine entsprechende Fehlermeldung ausgegeben werden.

Schnittstellen der eigentlichen Kommunikationseinheit
Die folgenden Schnittstellen befinden sich alle in der Schleife `(VT, DLQ, HBQ, TowerCBC)` und haben somit Zugriff auf die dort durchgereichten Parameter.

13. `{<PID>, {castMessage, {<Message>, <MessageVT>}}}`
    empfängt eine Nachricht einer Kommunikationseinheit

    ### Parameter:
    `PID`: Sender der empfangenen Nachricht (nicht benötigt)

    `Message`: Nachricht als Zeichenkette

    `MessageVT`: Vektorzeitstempel der Nachricht

    ### Implementation:
    13.1. Überprüfung, ob die neue Nachricht auslieferbar ist (`checkDeliverable(VT, MessageVT)`)

    13.2. Wenn `true`: Einfügen von `{<Message>, <VTMessage>}` in `DLQ`

    13.3. Wenn `false`: Einfügen von `{<Message>, <VTMessage>}` in `HBQ`

14. `{<From>, stop}`
    beendet die `loop(VT, DLQ, HBQ, TowerCBC)`-Schleife der aktuellen Kommunikationseinheit

Parameter:

`From`: Prozess-ID des Anwenderprozesses

Implementation:

14.1. Beenden der Schleife

14.2. Senden einer Bestätigung `ok` an den Prozess `From`

15. `{<From>, {send, <Message>}}`
sendet eine Nachricht als kausaler Multicast an alle anderen Kommunikationseinheiten

Parameter:

`From`: Prozess-ID des Anwenderprozesses

`Message`: die zu versendende Nachricht

Implementation:

15.1. Hochzählen des eigenen Ereigniszählers (`VT = tickVT(VT)`)

15.2. `Msg = {Message, VT}`

15.3. Einfügen von `Msg` in die DLQ (`addToDLQ(DLQ, Msg)`)

15.4. Senden von `{self(), {multicastNB, <Msg>}}` an `TowerCBC`. Dieser kümmert sich um die Ausführung des Multicasts

15.5. Senden einer Bestätigung `ok` an den Prozess `From`

16. `{<From>, received}`
liest die erste auslieferbare Nachricht aus der DLQ (blockierend)

Parameter:

`From`: Prozess-ID des Anwenderprozesses

Implementation:

16.1. `Msg = getMessage(DLQ})`

16.2. Wenn `Msg == null`: Warten und Blockieren des Prozesses, solange bis eine neue auslieferbare Nachricht eintrifft und diese dann unter `Msg` speichern

16.3. `Msg = {Message, MessageVT}`

16.4. `{ok, <Message>}` an `From` senden

16.5. Anschließendes Synchronisieren der lokalen Vektoruhr `VT` mit `VTMessage` (`syncVT(VT, MessageVT)`)

16.6. Überprüfung, ob neue Nachrichten auslieferbar sind und Verschieben dieser in die DLQ (`moveDeliverable(HBQ, DLQ, VT)`)

17. `{<From>, read}`
liest die erste auslieferbare Nachricht aus der DLQ (nicht blockierend)

Parameter:
`From`: Prozess-ID des Anwenderprozesses

Implementation:
17.1. `Msg = getMessage(DLQ)`

17.2. Wenn `Msg == null`: Senden von `{ok, null}` an `From` und Beenden dieser Anfrage

17.3. Ansonsten: `{Message, MessageVT} = Msg`

17.4. `{ok, <Message>}` an `From` senden

17.5. Anschließendes Synchronisieren der lokalen Vektoruhr `VT` mit `VTMessage` (`syncVT(VT, VTMessage)`)

17.6. Überprüfung, ob neue Nachrichten auslieferbar sind und Verschieben dieser in die DLQ (`moveDeliverable(HBQ, DLQ, VT)`)

Schnittstellen der HBQ
18. `initHBQ()`
initialisiert die HBQ

Parameter:
keine

Rückgabe:
eine leere HBQ

Implementation:
18.1 Rückgabe einer leeren Liste

19. `addToHBQ(HBQ, {Message, MessageVT})`
fügt eine Nachricht in die HBQ ein

Parameter:
`HBQ`: HBQ der aktuellen Kommunikationseinheit

`Message`: erhaltene Nachricht

`MessageVT`: Zeitstempel der Nachricht

Rückgabe:
neue Version der HBQ, inklusive der angegebenen Nachricht

Implementation:

19.1. Einfügen von `{<Message>, <MessageVT>}` in `HBQ`

19.2. Rückgabe von `HBQ`

20. `checkDeliverable(VT, MessageVT)`
prüft eine Nachricht auf Auslieferbarkeit (durch Überprüfen des dazugehörigen Zeitstempels)

Parameter:

`VT`: momentaner Zeitstempel der aktuellen Kommunikationseinheit

`MessageVT`: Zeitstempel der zu überprüfenden Nachricht

Rückgabe:

`true`, wenn die Nachricht auslieferbar ist

`false`, wenn die Nachricht nicht auslieferbar ist

Implementation:

20.1. `Return = aftereqVTJ(VT, MessageVT)`

20.2. Rückgabe von `true`, wenn `Return == {aftereqVTJ, -1}`

20.3. Ansonsten Rückgabe von `false`

21. `moveDeliverable(HBQ, DLQ, VT)`
verschiebt alle auslieferbaren Nachrichten von der HBQ in die DLQ

Parameter:

`HBQ`: HBQ der aktuellen Kommunikationseinheit

`DLQ`: DLQ der aktuellen Kommunikationseinheit

`VT`: momentaner Zeitstempel der aktuellen Kommunikationseinheit

Rückgabe:

neue Versionen der HBQ und DLQ: `{HBQ, DLQ}`

Implementation:

21.1. Schleife durch alle Nachrichten `{Message, MessageVT}` in `HBQ`:

21.2. Überprüfung der Auslieferbarkeit mithilfe von `checkDeliverable(VT, MessageVT)`

21.3. Wenn `true`, dann Entfernen von `{Message, MessageVT}` aus `HBQ` und Einfügen in `DLQ` mithilfe von `addToDLQ(DLQ, {Message, MessageVT})`

21.4. Wenn `false`, dann wird die Nachricht in der HBQ gelassen

Schnittstellen der DLQ

22. `initDLQ()`
initialisiert die DLQ

Parameter:
keine

Rückgabe:
eine leere DLQ

Implementation:
22.1 Rückgabe einer leeren Liste

23. `addToDLQ(DLQ, {Message, MessageVT})`
fügt eine Nachricht in die DLQ ein

Parameter:
`DLQ`: DLQ der aktuellen Kommunikationseinheit

`Message`: erhaltene Nachricht

`MessageVT`: Zeitstempel der Nachricht

Rückgabe:
neue Version der DLQ, inklusive der angegebenen Nachricht

Implementation:
23.1. Einfügen von `{<Message>, <MessageVT>}` in `DLQ`

23.2. Rückgabe von `DLQ`

24. `getMessage(DLQ)`
ermittelt die erste auslieferbare Nachricht aus der DLQ

Parameter:
`DLQ`: DLQ der aktuellen Kommunikationseinheit

Rückgabe:
`{<Nachricht>, <NeueDLQ>}`, wenn die DLQ nicht leer war

`{null, []}`, wenn die DLQ leer war

Implementation:
24.1. Rückgabe von `{null, []}`, wenn die DLQ eine leere Liste ist

24.2. Ansonsten Rückgabe des ersten Elementes `Elem` in `DLQ` und der Restliste `Rest` der DLQ: `{Elem, Rest}`

Vektoruhr-ADT

Die Vektoruhr wird genutzt, um den globalen Ablauf des Systems in die richtige Reihenfolge bringen zu können. Wie der Name schon sagt, wird diese als Vektor dargestellt, wobei jeder Skalar den Ereigniszähler eines Prozesses im System darstellt. Über den Index eines Prozesses kann sein Ereigniszähler in der Vektoruhr ausgelesen werden. In der nachfolgend definierten Komponente wird die Funktionalität einer solchen Uhr implementiert.

Generelle Anforderungen

1. Die Implementation dieser Komponente muss in einer Datei `vectorC.erl` erfolgen.

2. Kontaktinformationen für die Vektoruhrzentrale befinden sich in einer Konfigurationsdatei `towerClock.cfg`. Parameter in der Datei: `servername`, `servernode`

3. Die erstellten Vektorzeitstempel haben den folgenden Aufbau:
   `{<ID>, [<Ereigniszähler 1>, ..., <Ereigniszähler N-1>, <Ereigniszähler N>]}`

Schnittstellen

4. `initVT()`
   erstellt einen initialen Vektorzeitstempel

   ### Parameter:
   keine

   ### Rückgabe:
   initialer Vektorzeitstempel

   ### Implementation:
   4.1. Auslesen von `servername` und `servernode` aus der Konfigurationsdatei

   4.2. Kontaktaufbau zur `servernode` der `towerClock`-Komponente

   4.3. Anfrage einer eindeutigen `ID` für den neuen Zeitstempel bei Prozess `servername`

   4.4. Erstellen der Datenstruktur aus erhaltener `ID` und einem Nullvektor der Länge `ID`

   4.5. Rückgabe der erstellten Datenstruktur

5. `myVTid(VT)`
   ermittelt die eindeutige ID der Kommunikationseinheit aus einem Vektorzeitstempel

   ### Parameter:
   `VT`: Vektorzeitstempel

   ### Rückgabe:
   Prozess-ID als Integer

Implementation:

5.1. Rückgabe des ersten Wertes in `VT`

6. `myVTvc(VT)`
ermittelt den Vektor eines Vektorzeitstempels

Parameter:

`VT`: Vektorzeitstempel

Rückgabe:

Vektor als Liste von Integer

Implementation:

6.1. Rückgabe des zweiten Wertes in `VT`

7. `myCount(VT)`
ermittelt den Zählerwert zur eigenen ID aus einem Vektorzeitstempel

Parameter:

`VT`: Vektorzeitstempel

Rückgabe:

Zählerwert als Integer

Implementation:

7.1. `ID` und `Vektor` aus `VT` ermitteln

7.2. Zählerwert an Index `ID` aus `Vektor` auslesen

7.3. ausgelesenen Zählerwert zurückgeben

8. `foCount(J, VT)`
ermittelt den Zählerwert für einen bestimmten Index ein einem Vektorzeitstempel

Parameter:

`J`: Index des gewünschten Ereigniszählers

`VT`: Vektorzeitstempel

Rückgabe:

Zählerwert als Integer

Implementation:

8.1. `Vektor` aus `VT` ermitteln

8.2. Zählerwert an Index `J` aus `Vektor` auslesen

8.3. ausgelesenen Zählerwert zurückgeben

9. `isVT(VT)`
   überprüft einen Vektorzeitstempel auf Gültigkeit

   ### Parameter:
   `VT` : Vektorzeitstempel

   ### Rückgabe:
   `true` , wenn `VT` ein gültiger Zeitstempel ist

   `false` , wenn `VT` kein gültiger Zeitstempel ist

   ### Implementation:
   9.1. Überprüfung, ob `VT` eine Liste mit Länge `2` ist (nein: Rückgabe von `false`)

   9.2. `ID` und `Vektor` aus `VT` ermitteln

   9.3. Überprüfung, ob `ID` ein Integer ist (nein: Rückgabe von `false`)

   9.4. Überprüfung, ob `Vektor` eine Liste mit Länge `ID` ist (nein: Rückgabe von `false`)

   9.5. Überprüfung, ob die Elemente in `Vektor` Integer sind (nein: Rückgabe von `false`)

   9.6. Rückgabe von `true` , wenn alle Überprüfungen erfolgreich waren

10. `syncVT(VT1, VT2)`
    synchronisiert zwei Vektorzeitstempel

    ### Parameter:
    `VT1` : eigener Vektorzeitstempel
    `VT2` : Vektorzeitstempel, mit dem synchronisiert werden soll

    ### Rückgabe:
    neuer, synchronisierter Vektorzeitstempel

    ### Implementation:
    10.1. `ID` und `Vektor1` aus `VT1` , und `Vektor2` aus `VT2` ermitteln

    10.2. `{Vektor1Ext, Vektor2Ext} = extendVector(Vektor1, Vektor2)`

    10.3. Eine neue Liste `VektorNeu` erzeugen

    10.4. Durch `Vektor1Ext` und `Vektor2Ext` iterieren und für die Elemente beider Vektoren jeweils die Zählerwerte vergleichen. Der größere von beiden Werte wird jeweils in `VektorNeu` eingefügt (selbe Reihenfolge).

    10.5. Eine neue Liste `VTOut` erzeugen und dort `ID` und `VektorNeu` einfügen

10.6. Rückgabe von `VTOut`

11. `tickVT(VT)`
erhöht den Ereigniszähler der aktuellen Kommunikationseinheit um 1

Parameter:
`VT` : Vektorzeitstempel

Rückgabe:
neuer, hochgezählter Vektorzeitstempel

Implementation:
11.1. `ID` und `Vektor` aus `VT` ermitteln

11.2. Zählerwert `Value` an Index `ID` aus `Vektor` auslesen

11.3. Erzeugen einer Kopie `VektorNeu` von `Vektor`, wobei der Wert am Index `ID` mit `Value + 1` ersetzt wird

11.4. Eine neue Liste `VTOut` erzeugen und dort `ID` und `VektorNeu` einfügen

11.5. Rückgabe von `VTOut`

12. `compVT(VT1, VT2)`
vergleicht zwei Vektorzeitstempel

Parameter:
`VT1` : Vektorzeitstempel 1

`VT2` : Vektorzeitstempel 2

Rückgabe:
`afterVT`, wenn `VT1 > VT2`

`beforeVT`, wenn `VT1 < VT2`

`equalVT`, wenn `VT1 == VT2`

`concurrentVT`, wenn keine zeitliche Reihenfolge existiert

Implementation:
12.1 `Vektor1` aus `VT1`, und `Vektor2` aus `VT2` ermitteln

12.2 `{Vektor1Ext, Vektor2Ext} = extendVector(Vektor1, Vektor2)`

12.1. Rückgabe des Funktionsaufrufs `compareVector(Vektor1Ext, Vektor2Ext)` zurückgeben

13. `aftereqVTJ(VT, VTR)`
vergleicht zwei Vektorzeitstempel im Sinne des kausalen Multicasts

Parameter:

`VT` : eigener Vektorzeitstempel

`VTR` : Vektorzeitstempel einer erhaltenen Nachricht

Rückgabe:

`{aftereqVTJ, <Distanz der beiden Zeitstempel an Stelle J>}` , wenn `VT >= VTR`

`false` , wenn `VT < VTR`

Implementation:

13.1. `Vektor1` aus `VT` , sowie ID `J` und `Vektor2` aus `VTR` ermitteln

13.2. `{Vektor1Ext, Vektor2Ext} = extendVector(Vektor1, Vektor2)`

13.3. Elemente an Stelle `J` aus `Vektor1Ext` und `Vektor2Ext` entfernen, und als `DistVT` und `DistVTR` merken

13.4. Funktionsaufruf `compareVector(Vektor1Ext, Vektor2Ext)` . Speichern der Rückgabe in der Variable `Result`

13.5. Rückgabe von `false` , wenn `Result` den Wert `beforeVT` oder `concurrentVT` hat

13.6. Ansonsten Rückgabe von `{aftereqVTJ, DistVT - DistVTR}`

Hilfsfunktionen

14. `extendVector(Vektor1, Vektor2)`
verlängert einen kleineren Vektor, sodass beide übergebenen Vektoren die gleiche Länge haben

Parameter:

`Vektor1` : erster Vektor

`Vektor2` : zweiter Vektor

Rückgabe:

zwei Vektoren mit gleicher Länge: `{<Vektor>, <Vektor>}`

Implementation:

14.1. `Diff = length(Vektor1) - length(Vektor2)`

14.2. `Diff > 0` ⇒ kürzerer Vektor = `Vektor2`
`Diff < 0` ⇒ kürzerer Vektor = `Vektor1`
`Diff == 0` ⇒ beiden Vektoren sind gleich lang, Überspringen der Schleife in 14.3

14.3. Schleife mit `|Diff|` Durchläufen: Einfügen von `0` in den kürzeren Vektor

14.4. Rückgabe von `{Vektor1, Vektor2}`

15. `compareVector(Vektor1, Vektor2)`
vergleicht zwei Vektoren

Parameter:

`Vektor1` : erster Vektor

`Vektor2` : zweiter Vektor

Rückgabe:

`afterVT` , wenn `Vektor1 > Vektor2`

`beforeVT` , wenn `Vektor1 < Vektor2`

`equalVT` , wenn `Vektor1 == Vektor2`

`concurrentVT` , wenn keine zeitliche Reihenfolge existiert

Implementation:

15.1. Durch `Vektor1` und `Vektor2` iterieren und für die Elemente (`Elem1` und `Elem2`) beider Vektoren jeweils die Zählerwerte vergleichen. Folgendes wird zurückgegeben, wenn die vorangehende Bedingung für alle verglichenen Element-Paare wahr ist:
`Elem1 <= Elem2` ⇒ `beforeVT`
`Elem1 == Elem2` ⇒ `equalVT`
`Elem1 => Elem2` ⇒ `afterVT`

15.2. Sollte keine dieser Bedingungen für alle Element-Paare eintreffen, wird `concurrentVT` zurückgegeben

Tower / Vektoruhr Zentrale

Der Tower stellt das zentrale Gegenstück der Vektoruhr-ADT dar und verwaltet die Prozessnummern, welche in den Vektoren die Prozesse darstellen.

Schnittstellen

1. `{getVecID, <PID>}`

Gibt eine eindeutige Prozess-ID der towerClock zurück.

Parameter

`PID` : die PID, an welche die Antwort geschickt werden soll.

Rückgabe

`{vt, <Prozess-ID>}` : eine eindeutige Prozess-ID welche den Prozess in anderen Vektoruhren identifiziert.

2. `init()`

   Startet die towerClock.

   ### Parameter
   `init` hat zwar keine direkten Parameter, liest jedoch folgende Einstellungen aus der Datei `towerCLock.cfg` aus.

   `servername`: der Name der `towerClock`

   `servernode`: die Erlang Node, auf welcher die towerClock gestartet werden soll

   ### Rückgabe
   `PID` des Tower

3. `stop(<PID>)`
   Beendet die towerClock.

   ### Parameter
   `PID`: ProzessID der towerClock

   ### Rückgabe
   `true`, wenn der Tower erfolgreich beendet wurde.

Tower / Ungeordneter Multicast
Zum Testen der Kommunikationseinheiten bietet der Tower einen ungeordneten Multicast. Er leitet also innerhalb einer Gruppe von Kommunikationseinheiten die Nachrichten weiter und verhindert, dass ein geordneter Multicast entsteht.

Der towerCBC kann entweder in einem **manuellen** Modus oder einem **automatischen** Modus gestartet werden.
Im **manuellen** Modus können Nachrichten einzeln an die Kommunikationseinheiten verschickt werden, wobei hier die Einheiten `multicastNB` benutzen müssen. Dieser Modus ist primär zum Testen des Systems gedacht. Im **automatischen** Modus sendet der Tower direkt die Nachrichten an alle im Cluster registrierten Kommunikationseinheiten.

Schnittstellen
1.
```
init() -> PID
init(auto|manu) -> PID
```

   Startet den Multicast.

   ### Parameter
   `auto`: Startet den Multicast im automatischen Modus

   `manu`: Startet den Multicast im manuellen Modus

   ### Rückgabe
   `PID`: ProzessID des Towers

2. `stop(<PID>)`
   Beendet towerCBC

   ### Parameter
   `PID` : ProzessID des towerCBC

   ### Rückgabe
   `true` , wenn der Tower erfolgreich beendet wurde.

3. `reset(<PID>)`
   Versetzt den towerCBC in den Initialzustand.

   ### Parameter
   `PID` : PrzessID des towerCBC

   ### Rückgabe
   `true` wenn die Operation erfolgreich war

4. `listall()`
   Listet alle registrierten Kommunikationseinheiten auf.

   ### Anforderungen
   Muss auf der Node des towerCBC ausgeführt werden.

   ### Rückgabe
   `true` wenn die Operation erfolgreich war, ansonsten `false`

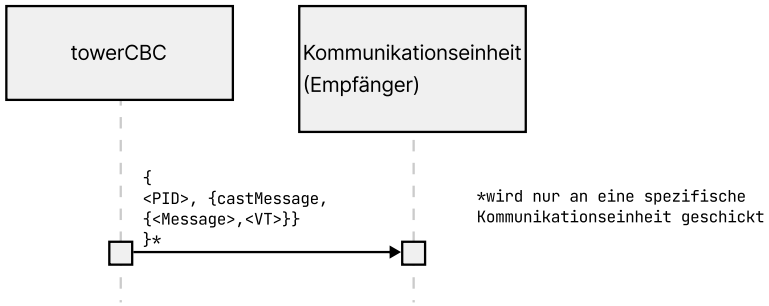
5. `cbcast(<Receiver>,<MessageNumber>)`
   Schickt die Nachricht mit der Nummer `MessageNumber` an `Receiver`.

   ### Parameter
   `Reveiver`: ProzessID des Empfängers der Nachricht

   `MessageNumber`: Nummer der Nachricht, welche verschickt werden soll

   ### Rückgabe
   `true` wenn die Operation erfolgreich war, ansonsten `false`

   ### Anforderungen
   Muss auf der Node des towerCBC ausgeführt werden und towerCBC muss sich im manuellen Modus befinden.

   ### Ablauf
   

6. `{<PID>,{register,<RPID>}}`
   Registriert eine Kommunikationseinheit beim towerCBC.

   ### Parameter
   `PID`: ProzessID des Senders der Nachricht

   `RPID`: ProzessID des zu regestrierenden Prozesses

   ### Rückgabe
   Schickt `{replycbc,ok_existing}` an `PID` zurück, wenn der `RPID` bereits registriert war und `{replycbc,ok_registered}` wenn der Prozess erfolgreich zum ersten Mal regestriert wurde.

7. `{<PID>, {multicastB, {<Message>,<VT>}}`
   Startet einen blockierenden (zu dieser Zeit ist der towerCBC nicht verfügbar) Multicast mit der Nachricht `Message`.

   ### Parameter
   `PID`: die ProzessID des Senders, welche im Log vermerkt wird

   `Message`: die Nachricht welche verschickt werden soll

   `VT`: der Vektorzeitstempel, in welchem die kausale Rheinfolge festgehalten wird.

8. `{<PID>,{multicastNB, {<Message>,<VT>}}}` Startet einen nicht blockierenden Multicast mit der Nachricht `Message`.

   ### Parameter

   `PID`: die ProzessID des Senders, welche im Log vermerkt wird

   `Message`: die Nachricht welche verschickt werden soll

   `VT`: der Vektorzeitstempel, in welchem die kausale Rheinfolge festgehalten wird.

Analyse

Anwendungszenario: ein Online Strategie Videospiel

Bei solch einem Anwendungsfall wäre die konsistente Reihenfolge von Aktionen wichtig und könnte dabei helfen, dass der Zustand des Spiels nach einiger Zeit immer für alle Spieler nachvollziehbar ist.
Außerdem müsste so nicht jeder Client immer auf die Aktion eines anderen warten, sondern könnte parallel Ereignisse im Spiel abarbeiten.

Ein Nachteil dieser Implementierung bei solch einer Anwendung wäre, dass das Auflösen der verschiedenen Spielzustände komplexer werden könnte, da es weit mehr Ereignisse geben würde als bei unserem vergleichbar simplen Projekt.

Ein weiterer Vorteil bei der Verwendung eines kausalen Multicasts bei Online Videospielen, wäre die Möglichkeit Spiele ohne einen zentralen Server stattfinden zu lassen. So müsste der Betreiber des Spiels nicht teuere Infrastrukturkosten bezahlen. Unsere Implementation hat hier jedoch eine entscheidende Schwäche.

Bei unserer Implementierung des kausalen Multicast gibt es die zwei zentralen Einheiten `towerCBC` und `towerClock`. Der kausale Multicast ermöglicht ja eigentlich eine serverlose dezentrale Architektur. Durch die Abhängigkeit von einer solchen zentralen Einheit gibt es jedoch wieder einen zentralen Auslastungspunkt, welcher bei vielen Clients alle Anfragen behandeln muss.