BAI5 SoSe2024

Ausarbeitung eines kausalen Multicasts

Verteilte Systeme - gelesen von
Prof. Dr. Christoph Klauck

Kristoffer Schaaf (2588265)

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik
Hochschule für angewandte Wissenschaften Hamburg

Inhaltsverzeichnis

1 Theorie

1.1 CBCast Algorithmus

In einem Netzwerk laufen verschiedene Prozesse auf verschiedenen Knoten und teilen sich keinen Speicherplatz. Die Interaktion zwischen den verschiedenen Prozessen läuft soweit ausschließlich über die Weitergabe von Nachrichten und kein Prozess kennt das Verhalten anderer Prozesse [Bab12]. Der *CBCast* (Chain-Based Broadcast) Algorithmus ist ein Algorithmus der im Bereich der verteilten Systeme zum Einsatz kommt und eine Lösung für genau diese Prozessinteraktion implementiert. Genutzt wie zum Beispiel vom ISIS Projekt [BC91] hat er sich in der Vergangenheit bereits mehrfach renommiert.

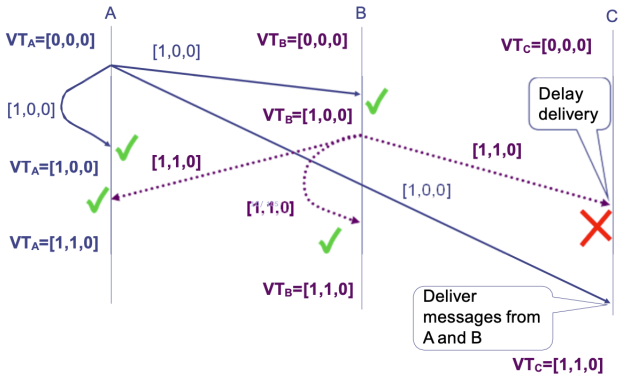

Abbildung 1: CBCAST [Kla24]

In Abb. 1 zu sehen ist ein beispielhafter Ablauf des *CBCASTs* mit drei Prozessen A, B und C. VT zeigt die Vektoruhren der jeweiligen Prozesse. Hier wird der eigene Zeitstempel und der der anderen Prozesse individuell gespeichert. Durch diese Uhr können die Prozesse trotz fehlendem geteilten Speicherplatz erkennen, ob sie mit den anderen Prozessen synchronisiert sind.
Im ersten Schritt verschickt A eine Nachricht an alle Teilnehmer im Netzwerk. Angehängt wird die eigene Vektoruhr - nun mit $A = 1$ erhöht - da dieser Prozess die Nachricht verschickt hat. Wichtig hierbei ist, dass der sendende Prozess die Nachricht immer zusätzlich an sich selber schickt, um eine sichere Synchronisation sicherstellen zu können. In Abb. 1 empfangen Prozess A und B nun die von A gesendete Nachricht. Die Vektoruhren werden verglichen und da jeweils nur ein Zeiger um 1 erhöht wurde, nehmen die Prozesse die gesendete Nachricht an. Nachdem B die Nachricht von A empfangen hat, schickt auch dieser Prozess eine Nachricht an alle Teilnehmer. A und B können diese empfangen. Beim Vergleichen der Vektoruhr von C und der von B mitgeschickten ist aber nun eine zu gder mroße Differenz. Zwei Zeiger sind jeweils um 1 erhöht, da B bereits die Nachricht von A empfangen hat. Bei C fehlt diese noch, deshalb blockiert C. Die Nachricht von A welche daraufhin eintrifft nimmt C dann an.
Die Zeiger in den Vektoruhren sind in vielen Implementierungen Zeitstempel der zuletzt empfangenen Nachricht.

1.2 Kommunikationseinheit

Die *Kommunikationseinheit* ermöglicht es Prozessen, welche über einen *ungeordneten Multicast* (siehe Kapitel 1.3) mit anderen Prozessen kommunizieren, Nachrichten zu schicken und zu empfangen. Das Interface stellt hierbei verschiedene Funktionen zum blockierenden und nicht blockierenden Senden von Nachrichten. Jeder Prozess, welcher als *Kommunikationseinheit* gestartet wird, empfängt bei korrekter Implementierung automatisch Nachrichten.
Desweiteren hat jede *Kommunikationseinheit* eine eigene Vektoruhr. Wird eine Nachricht von der *Kommunikationseinheit* gesendet, wird diese Vektoruhr um 1 erhöht.

Auslieferbarkeit von Nachrichten wird beim Empfangen einer Nachricht im jeweiligen *Kommunikationsprozess* geprüft. Ob eine Nachricht auslieferbar ist, wird durch zwei Bedingungen geprüft. Die notwendige Bedingung ist, dass die Vektoruhr der Nachricht logisch vor oder gleich der Vektoruhr des Prozesses ist. Die hinreichende Bedingung ist, dass die Distanz zwischen den beiden -1 ist, an der Stelle die den Zustand der Vektoruhr der Nachricht zeigt.

1.2.1 Holdback Queue

Ist eine Nachricht nicht auslieferbar wird diese zuerst in eine *Holdback Queue* sortiert. Für die Sortierung gibt es zwei verschiedene Möglichkeiten. Zum einen können neu empfangene Nachrichten direkt an den Anfang der Queue sortiert werden. Die zweite Möglichkeit ist, die *Holdback Queue* als *Priority Queue* umzusetzen. Sortiert wird hierbei anhand der, der Nachricht angehängten, Vektoruhr.
Es gibt vier verschiedene Positionen die Vektoruhren zueinander haben können:

- X before Y: Wenn X mindestens an einer Stelle höher und an allen anderen Stellen höher oder gleich Y ist.
- X after Y: Wenn X mindestens an einer Stelle kleiner und an allen anderen Stellen kleiner oder gleich Y ist.
- X equal Y: Wenn X an allen Stellen gleich Y ist.
- X concurrent Y: Wenn X an mindestens einer Stelle höher und an mindestens einer Stelle kleiner als Y ist.

Vorteilhaft dabei, die *Holdback Queue* nicht als *Priority Queue* umzusetzen ist, dass die Sortierung von *concurrent* Vektoruhren nicht beachtet werden muss. Außerdem ist die Implementierung schneller umzusetzen.
In der Effizienz beider Möglichkeiten ist ein geringer Unterschied zu erkennen. Die *Priority Queue* kann abhängig vom gewählten Sortieralgorithmus (siehe Abb. 2) eine minimale Komplexität von $O(n)$ erreichen. Wenn Nachrichten direkt an den Anfang der Queue einsortiert werden entsteht dadurch eine Komplexität von $O(1)$. Allerdings wird bei der Prüfung auf Auslieferbarkeit nun über die gesamte Queue iteriert, was eine zusätzliche Komplexität von mindestens $O(n)$ zur Folge hat. Beide Möglichkeiten haben also eine

Komplexität von $O(n)$. Abhängig von der Implementierung ist die *Priority Queue* leicht effizienter.

	aufsteigend aufgebaut	**absteigend** aufgebaut	**random** aufgebaut
insertionSort()	O(n) Jedes Element aus list wird nur einmal verglichen und ist dann schon an der richtigen Position.	O(n^2) Jedes Element aus list wird nur einmal verglichen Danach muss aber nochmal die richtige Position gefunden werden.	O (n^2) Der Aufwand ist ähnlich zu dem absteigenden. Es sollte aber etwas schneller gehen, da die richtige Position im Schnitt schneller gefunden wird.
quickSort() erstes Element als Pivotelement	O(n^2) Da das Pivotelement immer das kleinste (oder größte) in der Liste ist, ist die eine Teilliste fast leer und die andere fast voll.	O(n^2) Der Aufwand ist identisch zu dem aufsteigend sortiertem.	O ($n * log(n)$ Die Elemente werden in immer kleiner werdende fast gleich große Teillisten aufgeteilt.
heapSort()	O($n * log(n)$)	O($n * log(n)$)	O ($n * log(n)$)

Abbildung 2: Auswertung Sortieralgorithmen [Sch21]

1.2.2 Delivery Queue

Die *Delivery Queue* ist die zweite Queue eines *Kommunikationsprozesses*. Sie enthält alle auslieferbaren Nachrichten. Wird eine Nachricht ausgeliefert, wird die Vektoruhr der ausgelieferten Nachricht mit der des *Kommunikationsprozesses* synchronisiert.

1.3 Ungeordneter Multicast

Ein *Multicast* verteilt Nachrichten an Teilnehmer in einem Netzwerk. Die Teilnehmer müssen sich hierfür beim *Multicast* registriert haben. Der Unterschied vom *Multicast* zum *Broadcast* besteht darin, dass *Broadcast*-Systeme in der Regel auch die Möglichkeit bieten, ein Paket an alle Ziele zu adressieren, indem ein spezieller Code im Adressfeld verwendet wird. Wenn ein Paket mit diesem Code gesendet wird, wird es von jedem Rechner im Netz empfangen und verarbeitet. Diese Betriebsart wird als *Broadcasting* bezeichnet. *Multicasting* wiederum ist, wenn ein *Broadcast*-System auch die Übertragung an eine Teilmenge von Rechnern unterstützt. [TW11].
Ungeordnet ist ein *Multicast*, wenn die Nachrichten nicht in der Reihenfolge weitergegeben werden, in der sich die jeweiligen Teilnehmer/Prozesse beim *Multicast* registriert haben.

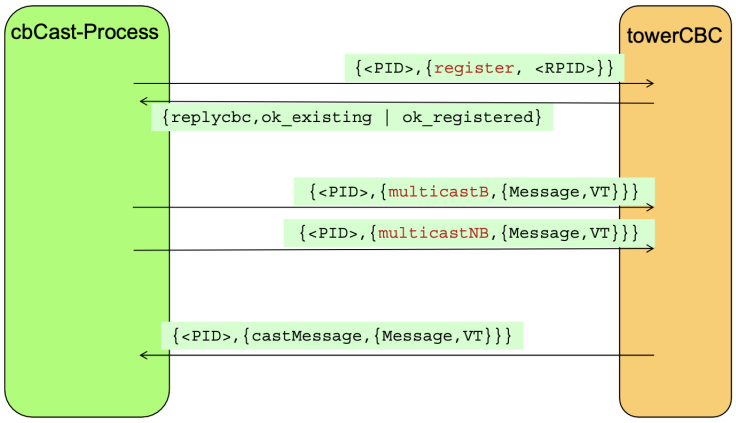

Abbildung 3: ungeordneter Multicast [Kla24]

In Abb. 3 ist eine abstrakte Kommunikation einer *Kommunikationseinheit* mit dem *Multicast towerCBC* dargestellt. Über *register* meldet sich der *cbCast-Prozess* beim *towerCBC* an. Dieser bestätigt die Anmeldung. Über *multicastB (blockierend)* oder *multicastNB (nicht blockierend)* kann der Prozess nun Nachrichten über den Multicast an alle Teilnehmer im Netzwerk schicken. Der Multicast kann diese Nachrichten anschließend mit *castMessage* an die Teilnehmer schicken.

2 Entwurf

2.1 Kommunikationseinheit

2.1.1 init/0

Beim Initialisieren einer Kommunikationseinheit wird ein Prozess gestartet, welcher beim *towerCBC* registriert wird und bei der *towerClock* eine neue Vektoruhr ID anfragt. Wie in Abb. 4 zu sehen, wird der Aufruf an die *towerClock* von dem Prozess gesendet, der auch beim *towerCBC* registriert wird. Grund dafür ist, dass die *towerClock* die Prozess ID des anfragenden Prozesses auf dessen Vektoruhr ID mappt. Würde der Prozess, welcher den *cbCast* Prozess erzeugt diese Anfrage schicken, würde dieses Mapping eine falsche Prozess ID speichern.
Der Verbindungsaufbau oder auch Verbindungstest terminiert das Programm, wenn er fehlschlägt.
Nach Erzeugung des *cbCast* Prozesses ist dieser sequenziell nicht mehr gebunden an den Prozess, der diesen erzeugt hat. Dementsprechend verlaufen die Aufrufe dieser beiden nebenläufig.

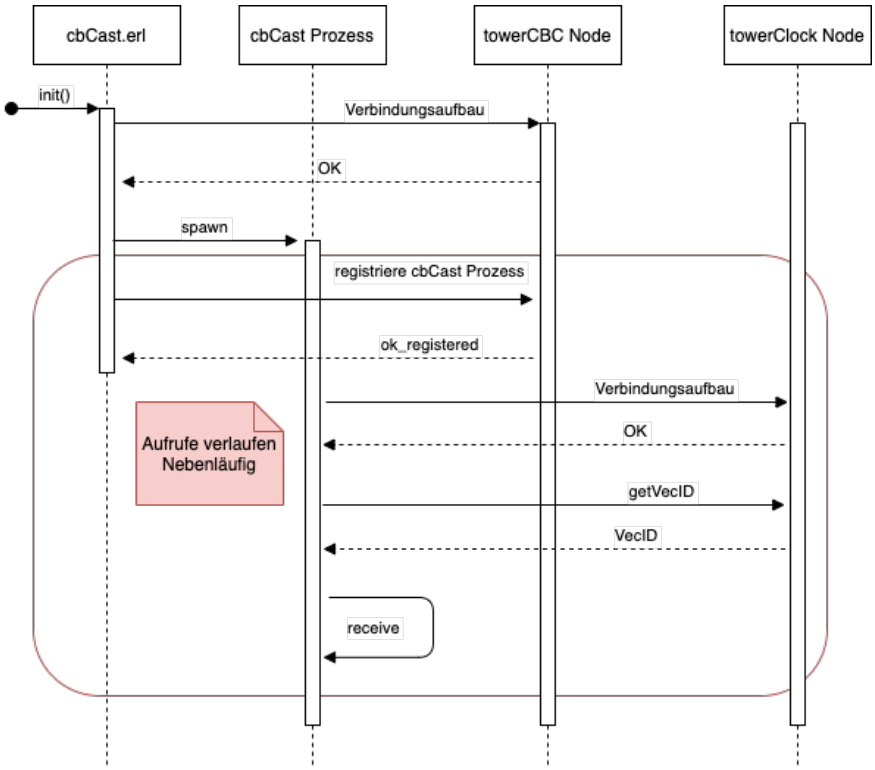

Abbildung 4: Sequenzdiagramm Initialisierung

Alternativ könnte sich der *cbCast.erl* auch erst beim *towerClock* die *VektorID* holen und dann den *cbCast* Prozess erzeugen. Durch den Ablauf in Abb. 4 kann das Holen der *VektorID* aber nebenläufig zur Registrierung beim *towerCBC* passieren. Der ganze Vorgang ist somit schneller.

2.1.2 stop/1

Die Terminierung erfolgt auf zwei verschiedenen Wegen. Zuerst wird der Prozess gestopp. Das bedeutet, dass ein sogenannter *Graceful Shutdown* durchgeführt wird. Hat dieser kein Erfolg, wird der Prozess durch einen *Hard Shutdown* gekillt.
Als Parameter wird der zu terminierende Prozess übergeben.

2.1.3 send/2

Beim Senden (siehe Abb. 5) wird eine Nachricht *(Msg)* an den als Parameter übergebenen *Prozess einer Kommunikationseinheit (cbCast Prozess)* geschickt. Dieser erhöht seine Vektoruhr vor dem Senden um 1 und verschickt die Nachricht an den *Multicast*. Der *Multicast* verteilt die Nachricht an alle Prozesse, inklusive dem Sender.
Die Vektoruhr der versendeten Nachricht hat zu der Vektoruhr der *Kommunikationseinheit* eine Distanz von -1, wodurch diese Nachricht auslieferbar ist und direkt in die *Delivery Queue* einsortiert werden kann.
Aufgrund des manuellen Modus des *Multicasts* muss die Nachricht direkt in die *Delivery Queue* einsortiert werden. Für einen sauberen Ablauf und zum Sicherstellen der kausalen Ordnung wäre es angenehmer die Nachricht nach dem Senden wieder zu empfangen und durch die *Holdback Queue* laufen zu lassen, dies ist aber nicht möglich. Angenommen der *Prozess N einer Kommunikationseinheit* versendet eine Nachricht, welche nicht direkt in der *Delivery Queue* gespeichert wird, dann wird vor dem Versenden die Vektoruhr um 1 erhöht. Beim Empfangen der Nachricht würde die Distanz der Vektoruhr der Nachricht und der Vektoruhr des *Prozesses einer Kommunikationseinheit* eine Distanz von 0 haben (siehe Abb. 6). Um in die *Delivery Queue* sortiert zu werden, muss diese Distanz -1 sein, was durch das weitere Erhöhen der Vektoruhr des Prozesses nicht mehr möglich ist (siehe Abb. 7).

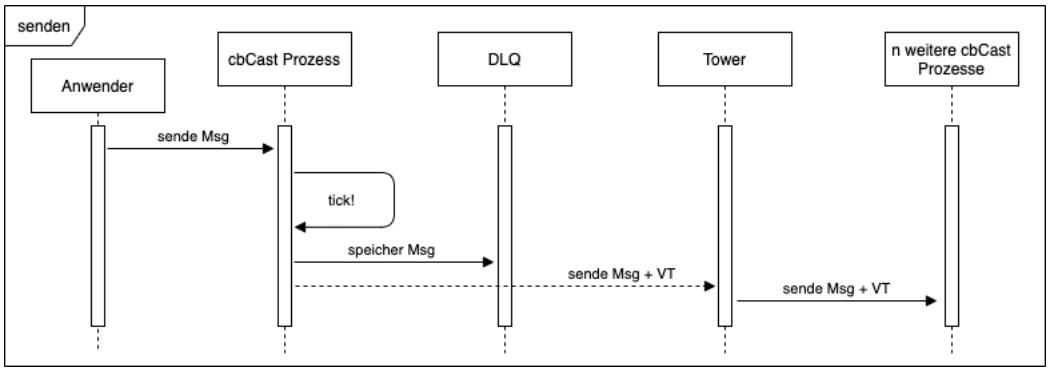

Abbildung 5: Sequenzdiagramm Senden

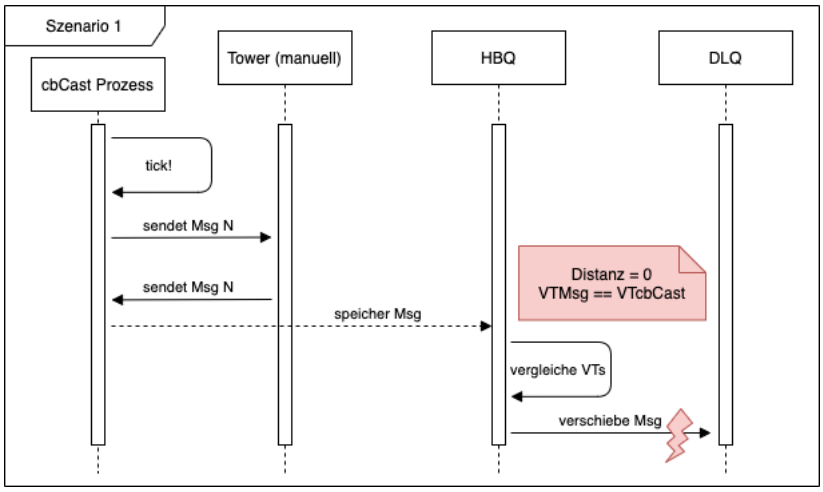

Abbildung 6: Auslieferbarkeit im manuellen Modus - Szenario 1

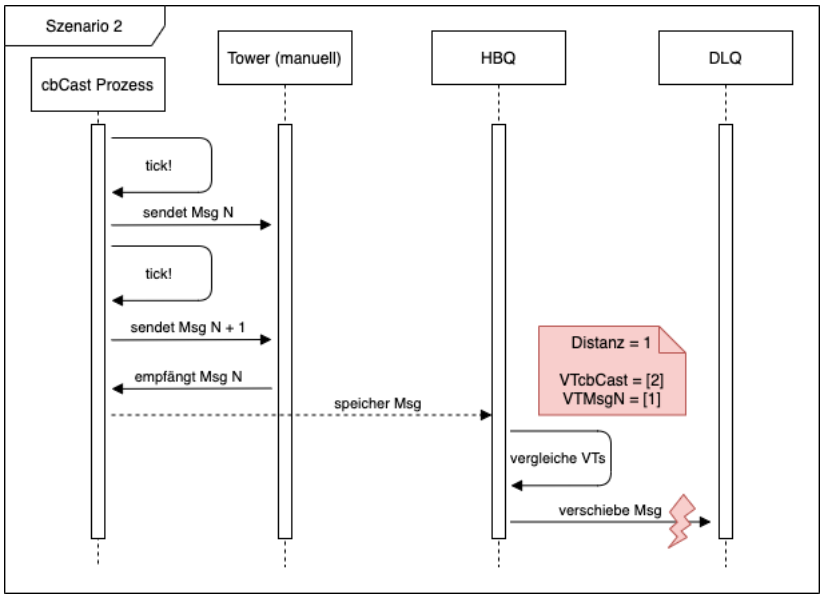

Abbildung 7: Auslieferbarkeit im manuellen Modus - Szenario 2

2.1.4 read/1 & received/1

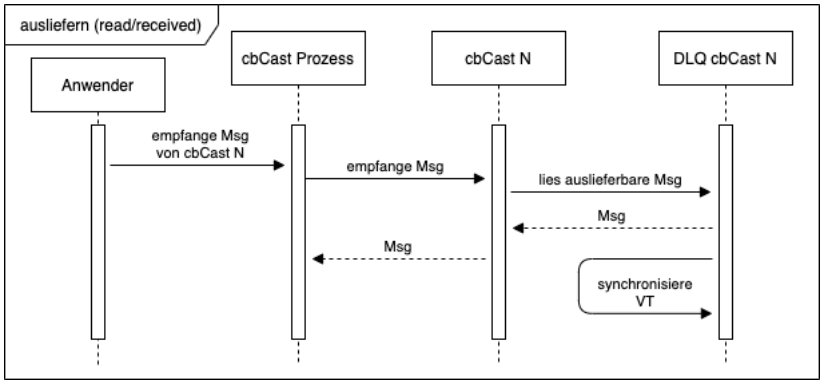

Abbildung 8: Sequenzdiagramm Ausliefern

Das Ausliefern einer Nachrichten (siehe Abb. 8) liest eine Nachricht aus der *Delivery Queue* des *Prozesses einer Kommunikationseinheit (cbCast N)*, welcher als Parameter übergeben wurde. Gelesen wird die der *Delivery Queue* zuerst hinzugefügte Nachricht. Eine Nachricht kann nur einmal gelesen werden und wird dabei aus der Queue entfernt.
Für das Ausliefern gibt es eine Funktion welche blockierend und eine, welche nicht blockierend empfängt.

read/1 (nicht blockierend) Falls der angefragte Prozess keine auslieferbare Nachricht zur Verfügung hat, wird nichts empfangen und der anfragende Prozess läuft normal weiter.

receive/1 (blockierend) Falls der angefragte Prozess keine auslieferbare Nachricht zur Verfügung hat, wartet der anfragende Prozess so lange, bis eine auslieferbare Nachricht empfangen wird.

In beiden Funktionen synchronisiert der angefragte Prozess anschließend seine Vektoruhr mit der der ausgelieferten Nachricht, falls eine Nachricht empfangen wurde.

2.1.5 $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

Wenn der *Multicast (Tower)* eine Nachricht verschickt, wird diese von der Schnittstelle $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$ des *Prozesses einer Kommunikationseinheit* empfangen (siehe Abb. 9). Daraufhin wird die Nachricht in die *Holdback Queue* des Prozesses gepusht. Aus in Kapitel 1.2.1 genannten Gründen, werden neue Nachrichten immer am Anfang der Queue gespeichert. Wenn die empfangene Nachricht von dem Prozess stammt, der sie empfangen hat, wird sie nicht in der *Holdback Queue* gespeichert, sondern verworfen. Das ist möglich durch das Speichern der Nachricht in der *Delivery Queue* beim Versenden (siehe Kapitel 2.1.3).

checkQueues/3 überprüft die *Holdback Queue* auf auslieferbare Nachrichten. Ist eine Nachricht auslieferbar, wird sie an den Anfang der *Delivery Queue* gepusht.

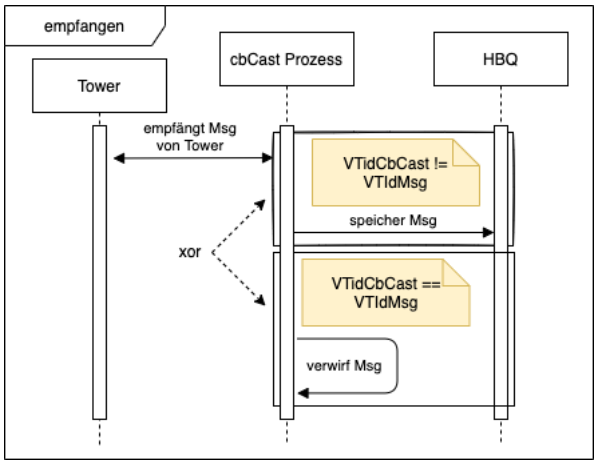

Abbildung 9: Sequenzdiagramm Empfangen

Diese Prüfung muss gemacht werden, wenn eine Nachricht in die *Holdback Queue* hinzugefügt wird und sobald die Vektoruhr des *Prozesses einer Kommunikationseinheit* verändert wird. Dies betrifft also die Schnittstellen **send/2**, **read/1**, **receive/1** und $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$ und wird am Ende der jeweiligen Schnittstelle aufgerufen.

2.2 Vektoruhr-ADT

Die Datenstruktur der Vektoruhr besteht aus einem Tupel mit zwei Elementen. Das erste Element zeigt die Identität der Vektoruhr. Das zweite Element zeigt die eigentliche Vektoruhr, bestehend aus einer Liste aus natürlichen ganzen positiven Zahlen, inklusive der 0.
Die Zahlen in der Liste zeigen den Zeitstempel der verschiedenen *Prozesse der Kommunikationseinheiten*. Die Identität ist der Index in der Liste, welcher den eigenen Zeitstempel zeigt.

2.2.1 initVT/0

Diese Schnittstelle initialisiert eine leere Vektoruhr. Um die richtige Identität zu kennen, wird diese bei der Vektoruhr Zentrale angefragt (siehe Kapitel 2.4 und Abb. 4). Die Identität ist gleichzeitig die Länge der initialen Liste. Alle Zustände sind bei Initialisierung 0. Eine leere Vektoruhr könnte also $\{1, [0]\}$ oder $\{5, [0, 0, 0, 0, 0]\}$ sein.

2.2.2 myVTid/1

MyVTid gibt die Identität der übergebenen Vektoruhr zurück.

2.2.3 myVTvc/1

MyVTvc gibt die eigentliche Vektoruhr als Liste zurück.

2.2.4 myCount/1

MyCount gibt den eigenen Zeitstempel der im Parameter übergebenen Vektoruhr zurück. Dieser ergibt sich anhand der Identität und der eigentlichen Vektoruhr. $\{1, [5]\}$ hat den Zeitstempel 5 und $\{5, [1, 1, 4, 2, 7, 3, 3]\}$ hat den Zeitstempel 7.

2.2.5 foCount/2

Als Parameter werden eine Vektoruhr und ein Index übergeben. Zurückgegeben wird der Zeitstempel der übergebenen Vektoruhr am übergebenen Index. Der Index muss größer als 0 sein.

2.2.6 isVT/1

IsVT prüft ob die übergebene Vektoruhr syntaktisch korrekt ist.

2.2.7 syncVT/2

Diese Schnittstelle empfängt zwei Vektoruhren. Zuerst werden beide Vektoruhren auf die gleiche Länge mit 0en aufgefüllt. Anschließend werden die normalisierten Vektoren verglichen. Es wird eine neue Vektoruhr mit der Identität des zuerst übergebenen Vektors zurückgegeben. Diese repräsentiert das Maximum aus beiden übergebenen Vektoruhren. Aus $\{3, [1, 4, 3]\}$ und $\{4, [2, 3, 1, 5]\}$ wird also $\{3, [2, 4, 3, 5]\}$.

2.2.8 tickVT/1

textitTickVT erhöht den eigenen Zeitstempel der übergebenen Vektoruhr um 1. Aus $\{2, [2, 1]\}$ wird entsprechend $\{2, [2, 2]\}$.

2.2.9 compVT/2

CompVT vergleicht zwei übergebene Vektoruhren. Hierbei gibt es vier verschiedene Rückgabewerte. Diese Rückgabewerte sind die Positionen die die beiden Vektoruhren zueinander haben (genauer beschrieben in Kapitel 1.2.1).
Nach der Normalisierung der Vektoruhren werden die beiden Vektoren elementweise verglichen. Dabei wird ein Vergleichszustand zurückgegeben, der angibt, ob ein Vektorzeitstempel logisch vor, nach oder gleich der anderen ist. Der Vergleich erfolgt rekursiv, wobei die Vektoren jeweils um das erste Element verkürzt werden und das Ergebnis Schritt für Schritt aktualisiert wird.

- *After* sind zwei Vektoruhren (VT1 *after* VT2), wenn der Vergleichzustand in einem Durchlauf ausschließlich logisch nach oder gleich ist.
- *Before* wenn er ausschließlich logisch vor oder gleich ist.
- *Equal* sind die Vektoren, wenn der Vergleichzustand ausschließlich logisch gleich ist.
- *Concurrent* also nebenläufig sind zwei Vektoruhren zueinander, wenn der Vergleichszustand in einem Durchlauf von logisch vor zu logisch nach oder umgekehrt wechselt.

2.2.10 aftereqVTJ/2

Diese Schnittstelle vergleicht im Sinne des kausalen Multicast die zwei übergebenen Vektoruhren VT und VTR. Dafür wird in beiden Vektoruhren zunächst das Element an der Stelle J entfernt, wobei J die Identität der Vektoruhr VTR darstellt. Die beiden neuen Vektoruhren werden nun über *compVT/2* (siehe Kapitel 2.2.9) miteinander verglichen. Wenn VT logisch nach oder gleich VTR ist, dann wird die Distanz zwischen den beiden entfernten Elementen zueinander zurückgegeben, also VT[J] - VTR[J].

2.3 Ungeordneter Multicast

2.3.1 init/1

Bei der Initialisierung des *Multicasts* (auch *Tower*) gibt es aus Testzwecken zwei verschiedene Modi (siehe Abb. 10). Zum einen kann der *Tower* im Modus *auto* gestartet werden. Dieser Modus ist hauptsächlich für die Schnittstelle $\{\langle PID\rangle, \{multicastB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$ gedacht. Nachrichten, welche vom *Tower* empfangen werden, werden direkt an alle Teilnehmer versendet und nicht gespeichert.
Im Modus *manu* sind die beiden Schnittstellen $\{\langle PID\rangle, \{multicastNB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$ und $\{\langle PID\rangle, \{multicastM, \langle CommNR\rangle, \langle MessageNR\rangle\}\}$ verfügbar. Zusätzlich kann vom Anwender die Schnittstelle *cbcast/2* (siehe Kapitel 2.3.4) aufgerufen werden. Im Gegensatz zum Modus *auto* werden empfangene Nachrichten gespeichert und können mehrfach versendet werden.

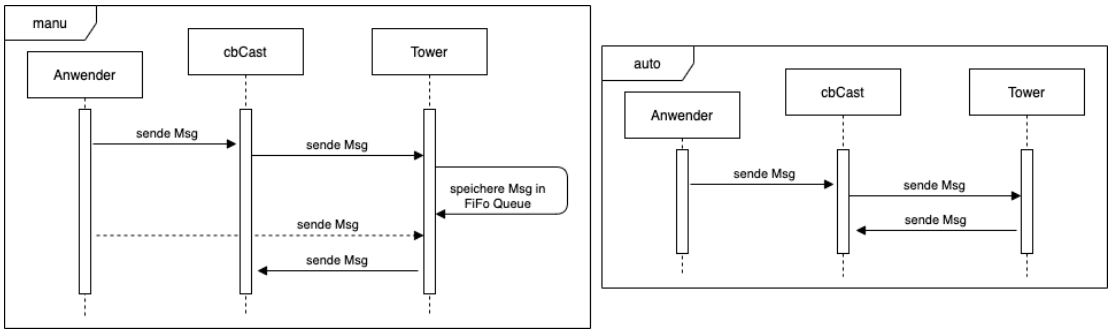

Abbildung 10: *Tower*: Vgl. *manu/auto*

2.3.2 stop/1

Die Schnittstelle *stop* stoppt den entsprechend übergebenen *Tower*. Wie auch bei der Kommunikationseinheit wird die Terminierung auf zwei verschiedenen Wegen durchgeführt (siehe Kapitel 2.1.2).

2.3.3 listall/0

Listall loggt alle, beim *Tower* registrierten, *Prozesse der Kommunikationseinheiten*. Hierbei wird lediglich die Prozess ID geloggt.

2.3.4 cbcast/2

Diese Schnittstelle ermöglicht das manuelle Senden von bestimmten Nachrichten an bestimmte registrierte *Multicast* Teilnehmer - in diesem Fall *Prozesse der Kommunikationseinheiten*. Als Parameter werden zwei ganze natürliche Zahlen größer 0 erwartet. Sowohl die registrierten Teilnehmer als auch die empfangenen Nachrichten werden in zwei separaten FiFo Queues gespeichert. Die beiden übergebenen Zahlen sind die Indizes der beiden Listen.

2.3.5 $\{\langle PID\rangle, \{register, \langle RPID\rangle\}\}$

Beim Senden an diese Schnittstelle wird die mitgesendete *RPID* in einer Liste im *Tower* gespeichert. Sobald die *RPID* in dieser Liste enthalten ist, ist der Prozess hinter dieser ID beim *Tower* registriert und kann somit Nachrichten empfangen.

2.3.6 $\{\langle PID\rangle, \{multicastB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

MulticastB steht in diesem Fall für ein blockierendes *Multicasting*. Genutzt wird die Schnittstelle im Modus *auto* des *Towers*. Versendet ein Prozess eine Nachricht an *multicastB*, wird die Nachricht an alle registrierten Teilnehmer versendet. Blockierend ist der Vorgang, da das Versenden nicht nebenläufig verläuft.

2.3.7 $\{\langle PID\rangle, \{multicastNB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

MulticastNB steht gegensätzlich zum *multicastB* für nicht blockierend. Nachrichten werden von anderen Prozessen im Modus *manu* des *Towers* empfangen und nicht direkt weiterversendet. Stattdessen werden die empfangenen Nachrichten in einer FiFo Queue gespeichert, auf die der *Tower* beim manuellen Senden über die Schnittstelle *cbcast/2* (siehe Kapitel 2.3.4) zugreifen kann.

2.3.8 $\{\langle PID\rangle, \{multicastM, \langle CommNR\rangle, \langle MessageNR\rangle\}\}$

Die letzte Schnittstelle *multicastM* versendet Nachrichten manuell. Während die anderen *Multicast* Schnittstellen vom Anwender oder von anderen Prozessen aufgerufen werden, wird diese Schnittstelle vom *Tower* selbst, bzw. über die Schnittstelle *cbcast/2* (siehe Kapitel 2.3.4) aufgerufen.

2.4 Vektoruhr Zentrale/Tower

Die zentrale Vektoruhr (*Tower*) verwaltet die Identitäten der jeweiligen *Prozesse der Kommunikationseinheiten*. Die Identitäten sind eindeutige IDs aus positiven ganzen Zahlen, beginnend bei 1. Aus Gründen der Erweiterbarkeit, wird die Identität zusammen mit der jeweiligen Prozess ID als Key Value Paar gespeichert.

2.4.1 $\{getVecID, \langle PID \rangle\}$

Beim Senden an diese Schnittstelle wird geprüft ob für den Prozess mit der ID *PID* schon eine Identität angelegt wurde. Falls dies der Fall ist, wird die entsprechende Identität zurückgesendet. Wenn nicht, wird eine neue Identität erzeugt und zurückgeschickt. Ist N die zuletzt erzeugte Identität wird N + 1 die Nächste.

2.4.2 init/0

Init erzeugt eine Vektoruhr Zentrale.

2.4.3 stop/1

Die Schnittstelle *stop* stoppt die entsprechend übergebene Vektoruhr Zentrale. Wie auch bei der Kommunikationseinheit wird die Terminierung auf zwei verschiedenen Wegen durchgeführt (siehe Kapitel 2.1.2).

2.5 Generelle Designentscheidungen

2.5.1 Logging

Pro Node wird eine generische .log Datei erstellt. Beispielweise gibt es für den Node *'cbCast1@MacBook'* die Datei *'cbCast1@MacBook'.log*. Dies bringt den Vorteil, dass die verschiedenen Kommunikationseinheiten - welche über die gleiche .beam Datei ausgeführt werden, aber auf verschiedenen Nodes laufen - separat voneinander geloggt werden.
Zum Debuggen war ein Gedanke, zusätzlich eine Logging Datei zu erstellen, in welcher alle Prozesse loggen. Hierdurch kann sequenziell nachverfolgt werden, ob die Reihenfolge der Aufrufe korrekt verläuft. Im Verlauf der Implementierung hat sich herausgestellt, dass diese Datei wenig Mehrwert bringt. Deswegen fehlt diese in der finalen Implementierung.

3 Realisierung

3.1 Allgemeines

3.1.1 Timeouts

In der Implementierung ist, bis auf Ausnahmen, für jeden *receive* Block ein *Timeout* eingebaut. Dieser beträgt 1000ms. Wird der Timeout ausgelöst, wird eine für jeden Fall individuelle Fehlermeldung geloggt.

3.1.2 Responses

Um die jeweiligen Nachrichten und vor allem die Responses korrekt einordnen zu können, sind diese wie folgt aufgebaut:
Wird die Nachricht $\{self(), \{getMessage\}\}$ an den *Prozess der Kommunikationseinheit* geschickt, dann ist die Antwort: $\{replycbcast, ok\_getMessage, \{...\}\}$. Anhand folgender Tabelle 1 kann eine Antwort-Nachricht eindeutig zugeordnet werden.

TowerClock	replyclock
TowerCBC	replycbc
Prozess der Kommunikationseinheit	replycbcast
Prozess der Queues	replyqueues

Tabelle 1: Namenszuordnung

3.2 Kommunikationseinheit

3.2.1 init/0

Die Initialisierung der *Kommunikationseinheit* erzeugt einen Prozess, dessen Prozess ID zurückgegeben wird. Wie in Abb. 9 zu sehen muss erst eine Verbindung zum *Tower* hergestellt werden. Über die *erlang* Funktion *net_adm:ping/1* wird angefragt, ob die *Node* des *Towers* erreichbar ist. Anschließend wird der eigentlich Prozess gestartet, diesem werden zuerst die Datei, in welcher geloggt wird und die Adresse des *Towers* mitgebenen.
Aus dem neuen Prozess heraus, werden jetzt zwei wichtige Schritte getriggert:

Die Initialisierung der Vektoruhr und die anschließenende Erzeugung eines neuen Prozesses für die *Holdback* und *Delivery Queue*, im Folgenden bezeichnet als der *Prozess für die Queues*. Folgende Funktion zeigt die gespeicherten Zustände dieses Prozesses und die verfügbaren Schnittstellen.

```
1 loopQueues(Datei, VT, HBQ, DLQ) ->
2     receive
3         {From, {pushHBQ, {Message, NewVT}}} ->
4             ...
5         {From, {pushDLQ, {Message, NewVT}}} ->
6             ...
7         {From, {popDLQ}} ->
8             ...
```

```
        {From, {checkQueues}} ->
            ...
        {From, {syncVT, {AsyncVT}}} ->
            ...
        {From, {tickVT}} ->
            ...
        {From, {getVTid}} ->
            ...
        {From, {listQueues}} ->
            ...
        Any ->
            ...
    end.
```

Der Prozess speichert die Datei, in welcher geloggt wird als Zeichenkette, den Vektorzeitstempel in der in Kapitel 3.3 festgelegten Datenstruktur und die *Holdback* und *Delivery* Queue jeweils als Liste. Während *listQueues* nur eine convenience Schnittstelle ist, welche ausschließlich zum Debuggen genutzt wird, haben alle anderen Schnittstellen eine wichtige Funktionalität:

- *pushHBQ* und *pushDLQ* fügen Nachrichten der jeweiligen *Queue* hinzu
- *popDLQ* entfernt eine Nachricht nach dem FiFo Prinzip aus der *Delivery Queue*
- *checkQueues* (siehe Abb. 11) prüft ob in der *Holdback Queue* auslieferbare Nachrichten enthalten sind und sortiert diese entsprechend in die *Delivery Queue*
- *syncVT* synchronisiert die im Prozess gespeicherte Vektoruhr mit einer als Parameter mitgelieferten Vektoruhr
- *tickVT* erhöht die im Prozess gespeicherte Vektoruhr um 1
- *getVTid* gibt die eigene Identität der im Prozess gespeicherten Vektoruhr zurück.

Das Empfangen von Nachrichten der Kommunikationseinheit was das Senden und das blockierende und nicht blockierende Empfangen von Nachrichten vom und an den *Tower* ermöglicht. Dieser Prozess speichert die in die zu loggende Datei, die Adresse des *Towers* und die Adresse des Prozesses in welchem die *Queues* gespeichert sind. Im Folgenden ist dies der *Prozess der Kommunikationseinheit*.

```
loop(Datei, TowerCBC, Queues) ->
    receive
        {_From, {castMessage, {Message, NewVT}}} ->
            ...
            loop(Datei, TowerCBC, Queues);
        ...
    end.
```

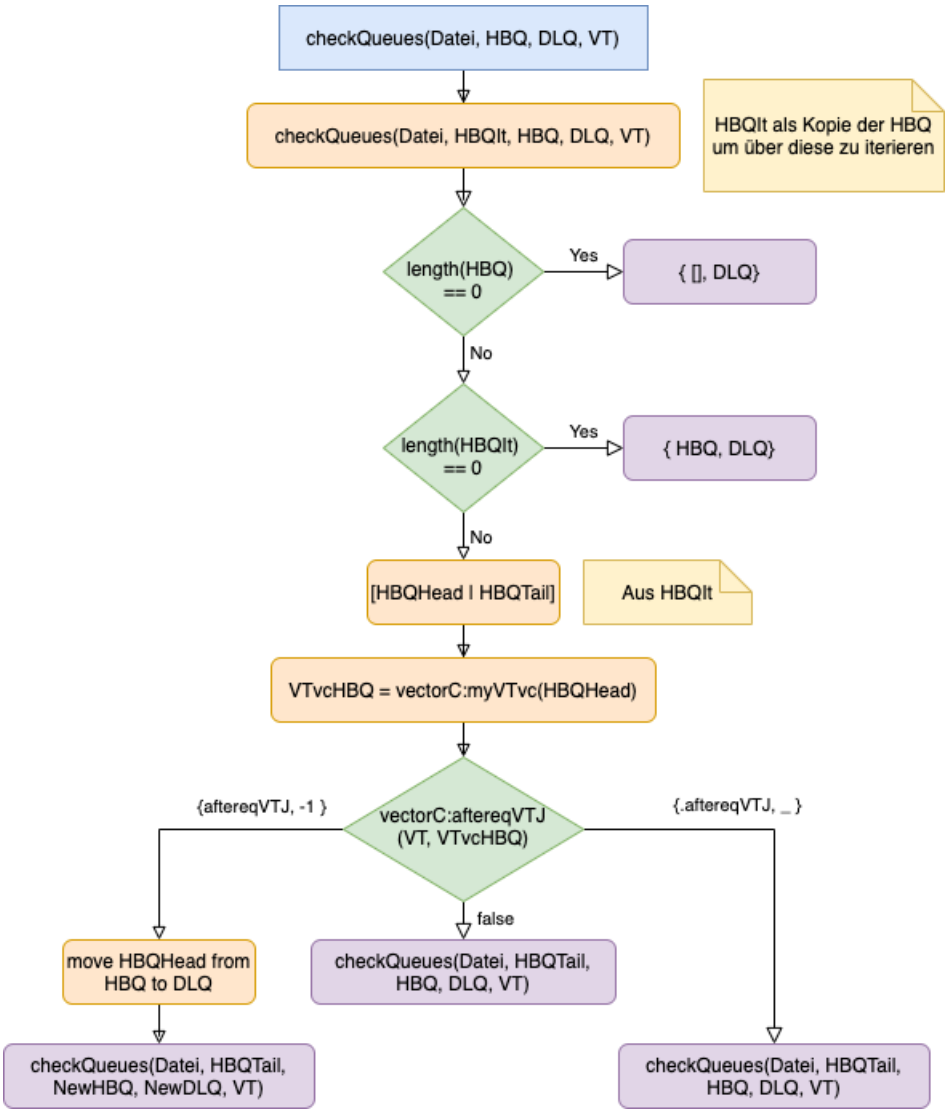

Abbildung 11: checkQueues

3.2.2 stop/1

Wie bereits in Kapitel 2.1.2 beschrieben, wird das Stoppen zuerst durch einen *Graceful Shutdown* versucht. Dieser schickt eine Nachricht an den Prozess nach dessen Bearbeitung dieser sich selbst nicht wieder aufruft und somit terminiert. Schlägt dies fehl, wird nach einem Timeout die *erlang* Funktion *exit/2* aufgerufen und ein *Hard Shutdown* durchgeführt.

3.2.3 send/2

Versendet wird eine Nachricht indem diese an den *Prozess der Kommunikationseinheit* (*cbCast Prozess*) geschickt wird (siehe Abb. 12).

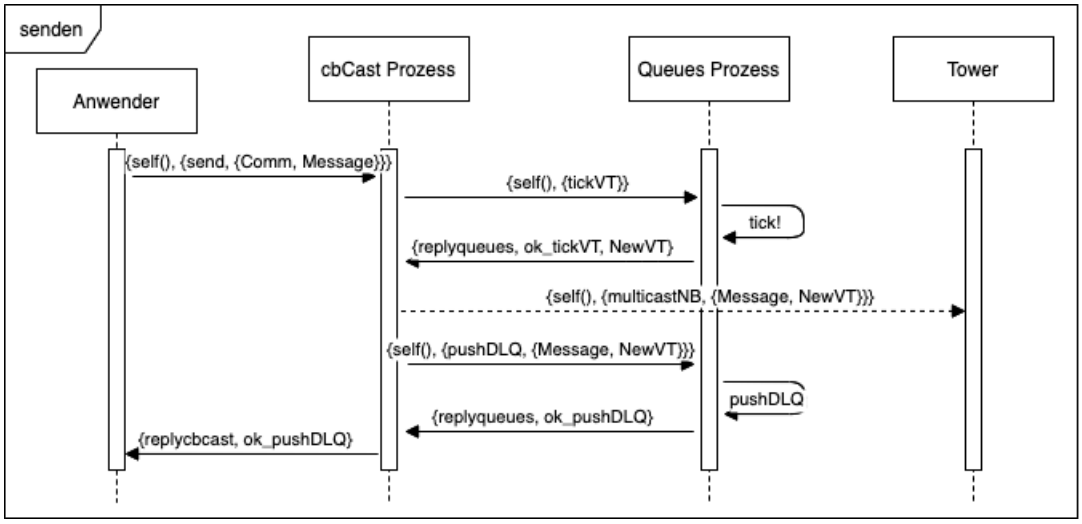

Abbildung 12: send/2

3.2.4 read/1 & received/1

Um eine Nachricht auszuliefern wird auch hier der *Prozess der Kommunikationseinheit* (*cbCast Prozess*) angesprochen. Beide Arten der Auslieferung schicken an die gleiche Schnittstelle. Diese fragt dann über den mitgeschickten Parameter *Blocking* ab ob blockierend oder nicht blockierend ausgeliefert werden soll.

read/1 (nicht blockierend) Das nicht blockierende Ausliefern schickt die Nachricht $\{self(), \{getMessage, false\}\}$ (siehe Abb. 13).

received/1 (blockierend) Das blockierende Ausliefern schickt die Nachricht $\{self(), \{getMessage, true\}\}$ (siehe Abb. 14).

3.2.5 $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

Die Schnittstelle *castMessage* (siehe Abb. 15) überprüft nach dem Empfangen einer Nachricht ob die Identität der Vektoruhr der Nachricht, die gleiche Identität hat, wie der *Prozess der Kommunikationseinheit*. Wenn dies der Fall ist, wird die Nachricht verworfen, da die Nachricht bereits beim Versenden in der *Delivery Queue* gespeichert wurde.

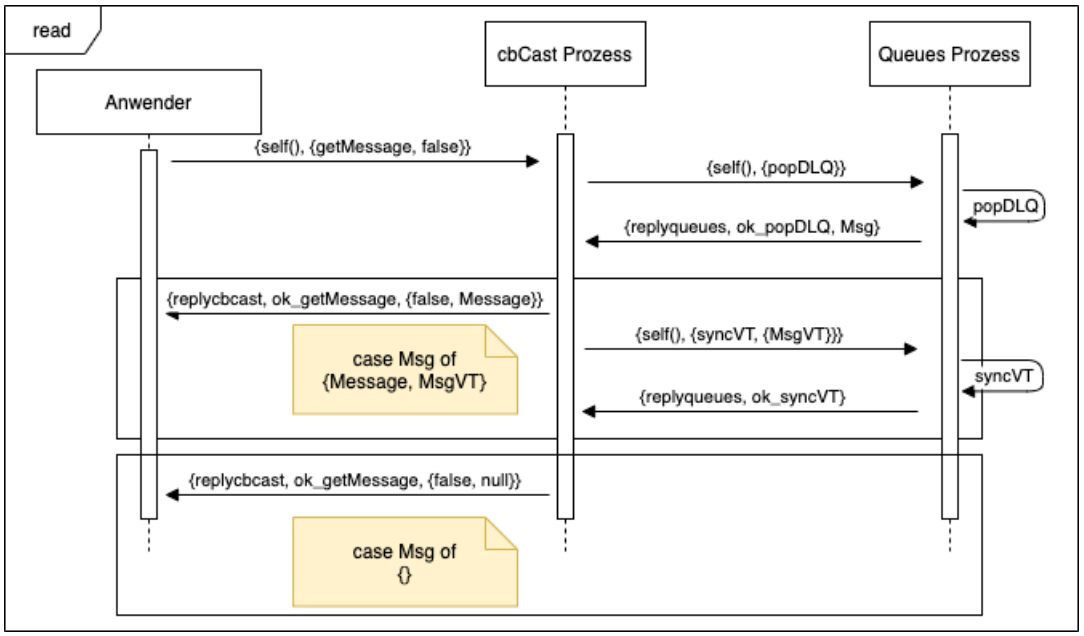

Abbildung 13: read/2

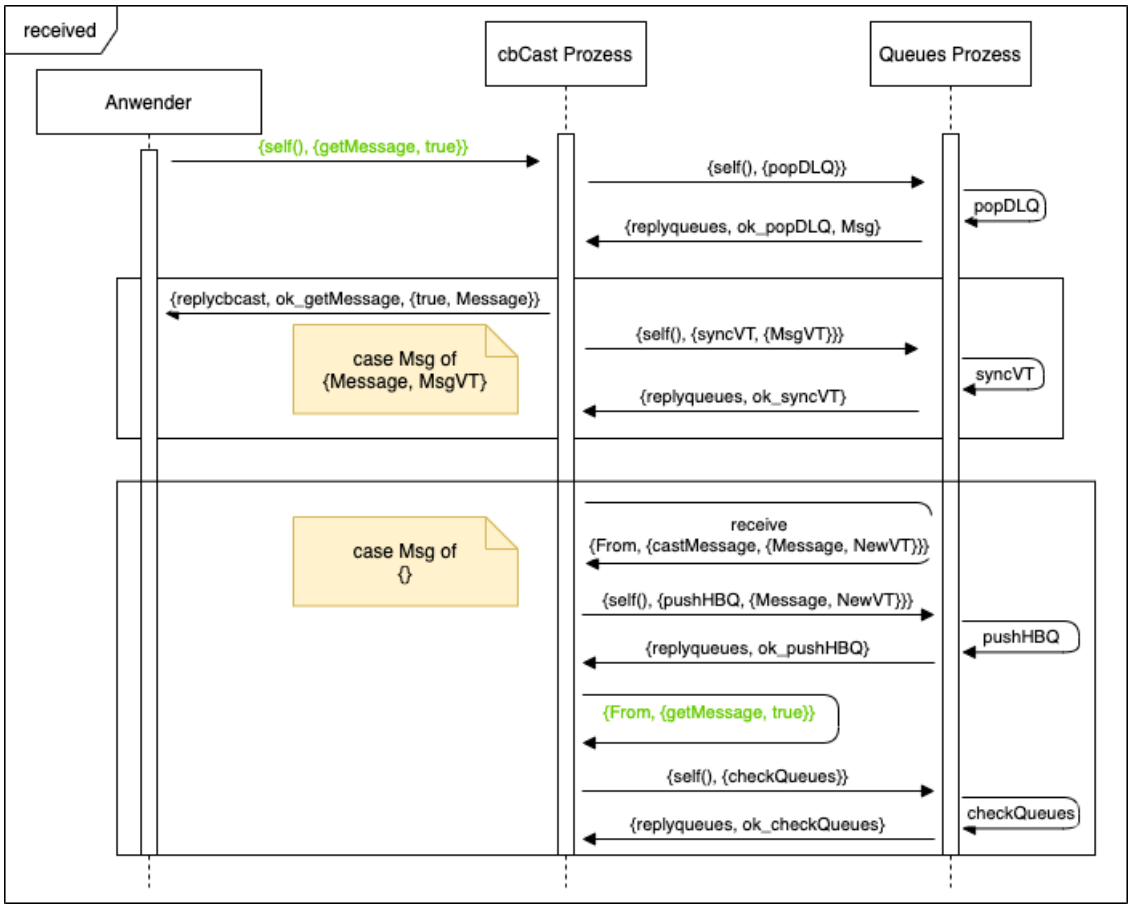

Abbildung 14: received/2

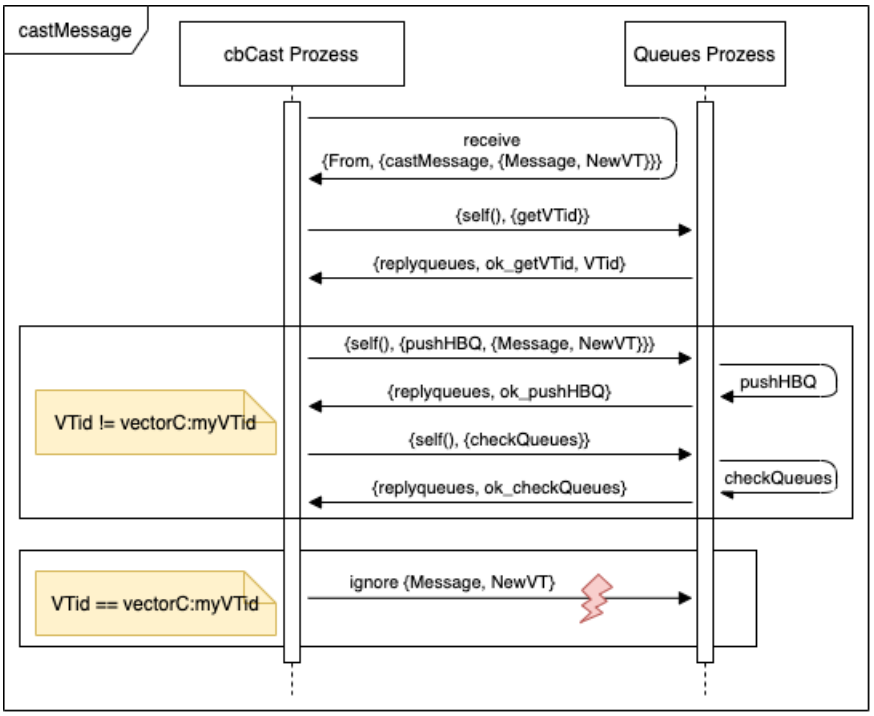

Abbildung 15: $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

3.3 Vektoruhr-ADT

Die bereits in Kapitel 2.2 beschriebene Datenstruktur ist ein *erlang* Tupel bestehend aus der Vektoruhr Identität als Integer und der Vektoruhr als *erlang* Liste. In der Vektoruhr sind alle Zeitstempel der verschieden *Prozesse der Kommunikationseinheiten*, inklusive dem Eigenen, gespeichert. Als Beispiel: $\{2, [1, 3, 4, 2]\}$. Die Vektoruhr Identität ist 2 und die Vektoruhr ist $[1, 3, 4, 2]$. Die Identitäten beginnen bei 1. Somit ist der eigene Zeitstempel dieser ADT 3.

```
1 VT = {2, [1,3,4,2]}.
2 vectorC:myCount(VT).
3 3
```

3.3.1 initVT/0

Diese Funktion stellt im ersten Schritt eine Verbindung zur *TowerClock* über die *erlang* Funktion *net_adm:ping/1* her. Ist der Verbindungsaufbau erfolgreich, wird am *TowerClock* die Identität angefragt. Zurückgegeben wird diese Identität *VecID* als Tupel {*VecID, zeros(VecID)*}. *zeros/1* erstellt eine Liste, welche genau so viele Nullen enthält, wie es im Parameter übergeben wurde.

3.3.2 myCount/1

Über die Hilfsfunktion *getElementByIndex(VT, VTID)* kann der entsprechende Ereigniszähler, bzw. der eigene Zeitstempel gefunden werden.

3.3.3 foCount/2

foCount/2 wirft eine Exception, wenn die übergebene Position J kleiner als 0 oder größer als die Länge der verfügbaren Zeitstempel ist.

```
1 foCount(J, {_VTID, VT}) when J > 0 andalso J =< length(VT)->
      getElementByIndex(VT, J);
2 foCount(_, _) -> throw({error, "Invalid index. Index must be greater than 0 and
      smaller than the size of available timestamps."}).
```

3.3.4 isVT/1

Zuerst wird geprüft ob das Tupel genau zwei Elemente enthält. Trifft dies zu, wird geprüft, dass

1. das erste Element ein Integer ist,
2. das zweite Element eine Liste ist,
3. jedes Element der Liste ein Integer ist.

3.3.5 syncVT/2

Im ersten Schritt dieser Funktion wird zuerst die Länge der beiden übergebenen Vektoruhren anhand der Hilfsfunktion *padWithZeros/2* normalisiert:

```
1 NormalizedVT1 = padWithZeros(VT1, length(VT2)),
2 NormalizedVT2 = padWithZeros(VT2, length(VT1)),
```

Diese beiden Vektoruhren werden dann Index für Index miteinander verglichen. Es wird jeweils das Maximum der beiden Werte als neue Liste gespeichert und zusammen mit der Identität der ersten Vektoruhr zurückgegeben.

3.3.6 tickVT/1

tickVT/1 nutzt die Hilfsfunktion *incrementElementAtIndex/3*. Übergeben werden dieser Funktion die Vektoruhr an sich, die Identität und ein Counter.

3.3.7 compVT/2

Auch in *compVT/2* werden zuerst die beiden Vektoruhren normalisiert (siehe Kapitel 3.3.5). Anschließend wird die Hilfsfunktion *compareLists/3* (siehe Abb. 16) aufgerufen.

3.3.8 aftereqVTJ/2

Die Funktion *aftereqVTJ/2* (siehe Abb. 17) nutzt zwei weitere Funktionen. Zuerst die *removeJ/2* Hilfsfunktion, welche in beiden Vektoruhren ein Element entfernt und dann die beiden neuen Vektoruhren vergleicht (siehe Kapitel 2.2.10).

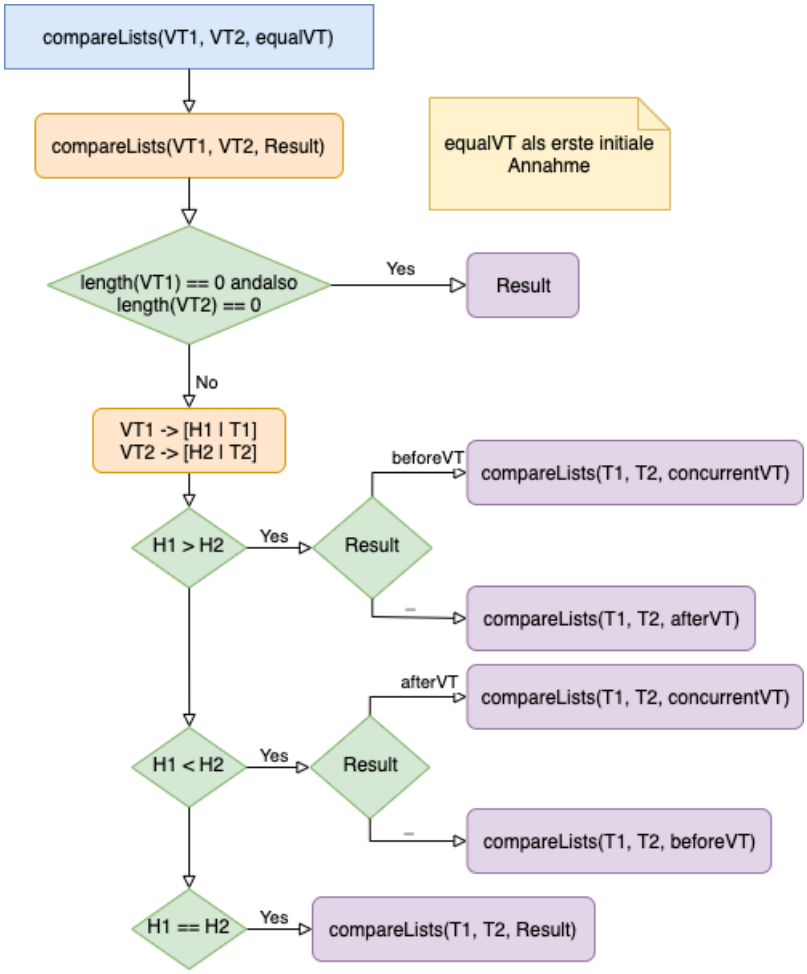

Abbildung 16: compVT

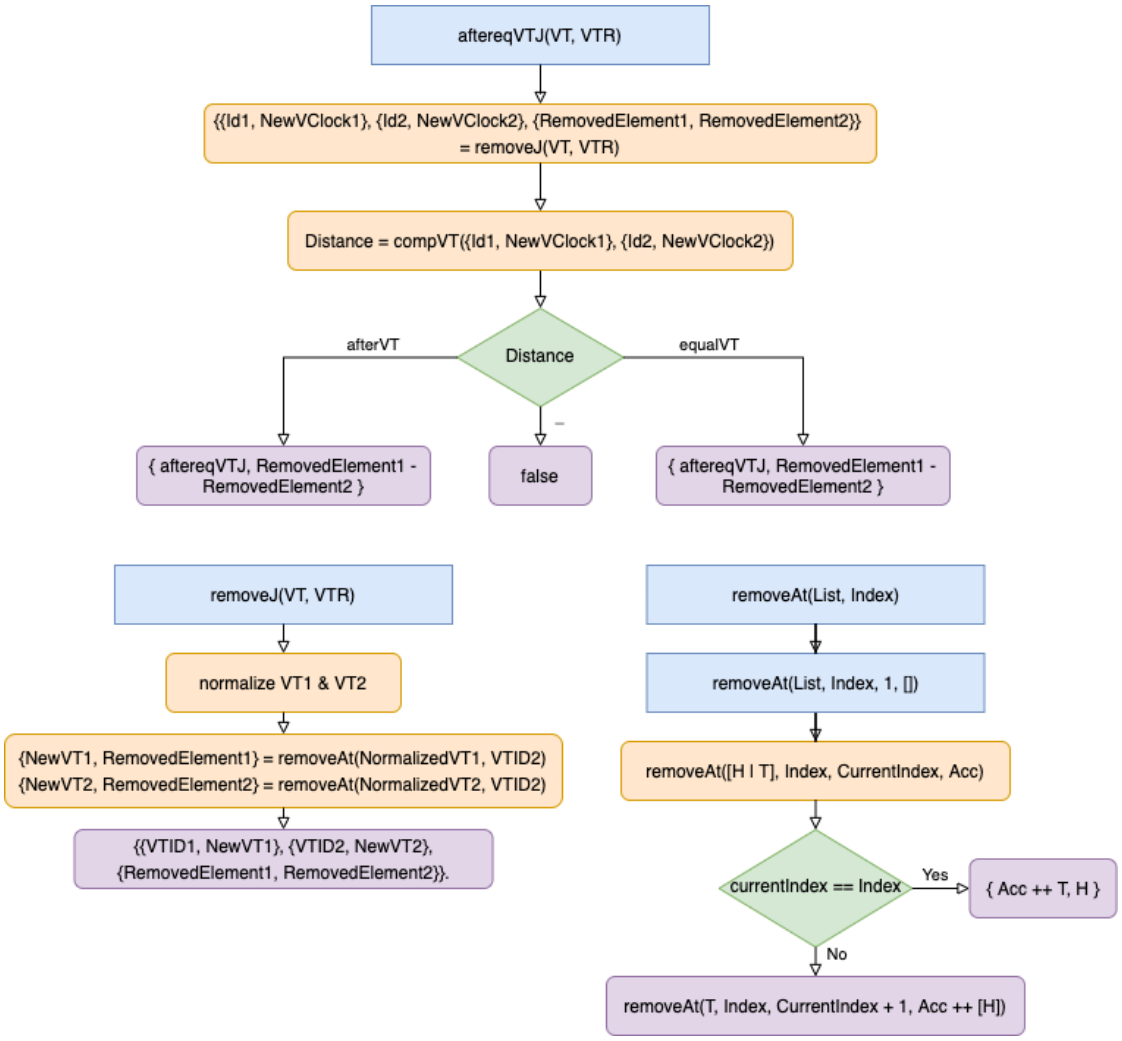

Abbildung 17: aftereqVTJ/2

3.4 Ungeordneter Multicast

3.4.1 init/1

Wie bereits in Kapitel 2.3.1 beschrieben, kann der *Tower* in zwei verschiedenen Modi gestartet werden. Diese werden dem Prozess in der Booleschen Variable *Auto* übergeben. Der Prozess wird im folgenden als der *Prozess des TowerCBCs* bezeichnet. Wird als Parameter *auto* übergeben, wird die Variable *Auto* auf true gesetzt. Für den Parameter *manu* wird *Auto* als false gespeichert. In der Variable *Registered* werden alle registrierten *Prozesse der Kommunikationseinheiten* gespeichert. Die Variable ist eine Liste nach dem FiFo Prinzip. Dies wird in Kapitel 3.4.5 genauer beschrieben. Auch die letzte Variable *Buffer* ist eine Liste nach dem FiFo Prinzip. In dieser Liste werden alle Nachrichten welche über die Schnittstelle $\{\langle PID\rangle, \{multicastNB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$, beschrieben in Kapitel 3.4.7, empfangen werden, gespeichert.

```
1 loop(Datei, Registered, Auto, Buffer) ->
2     receive
3         ...
4     end.
```

3.4.2 stop/1

Das Stoppen dieses Prozesses ist identisch zu Kapitel 3.2.2 umgesetzt.

3.4.3 listall/0

Diese Funktion schickt eine Nachricht an den *Prozess des TowerCBCs* welcher daraufhin alle, in der Variable *Registered*, gespeicherten Prozess Ids in die zu loggende Datei schreibt.

3.4.4 cbcast/2

Als Parameter empfängt diese Funktion zwei Integer, welche zuerst auf den richtigen Datentypen geprüft werden. Die Parameter sind Indexe für die beiden Listen, welche im *Prozess des TowerCBCs* gespeichert sind. Um sicherzustellen, dass diese Indexe mindestens größer als 0 sind ist diese Bedingung als Guard in der Funktion integriert. Anschließend wird eine Nachricht an den *Prozess des TowerCBCs* geschickt. Diese ist in Kapitel 3.4.8 aufgeführt.

3.4.5 $\{\langle PID\rangle, \{register, \langle RPID\rangle\}\}$

Beim Senden an diese Schnittstelle wird über die Hilfsfunktion *isListMember/2* geprüft ob die mitgeschickte Prozess ID *RPID* bereits in der Liste *Registered* enthalten ist. Ist dies der Fall, ist die Antwort an den Sender der Nachricht $\{replycbc, ok\_existing\}$. Wenn nicht, dann wird die Prozess ID der Liste hinzugefügt und die Antwort ist $\{replycbc, ok\_registered\}$.

3.4.6 $\{\langle PID\rangle, \{multicastB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

Auch bei dieser Schnittstelle wird die Hilfsfunktion *isListMember/2* genutzt um sicherzustellen, dass keine, nicht im *Prozess des TowerCBCs* registrierten Prozesse, Nachrichten via *Multicast* verschicken können. Ist die Registrierung bestätigt wird die, in der Nachricht mitgeschickte, Vektoruhr über die Funktion *vectorC:isVT/1* auf syntaktische Echtheit geprüft. Ist auch dieser Test bestanden, wird die Nachricht an alle *Prozesse der Kommunikationseinheiten* geschickt, welche in der Liste *Registered* enthalten sind.

3.4.7 $\{\langle PID\rangle, \{multicastNB, \{\langle Message\rangle, \langle VT\rangle\}\}\}$

Wie bereits in Kapitel 2.3.1 beschrieben, ist diese Schnittstelle nur im *manu* Modus möglich. Die Variable *Auto* muss also auf *false* gesetzt sein. Ähnlich zur vorherigen Schnittstelle (siehe Kapitel 3.4.6) wird in dieser zuerst der Sender der Nachricht überprüft. Ist dieser Registriert und ist die mitgeschickte Vektoruhr validiert wird die Nachricht an die im *Prozess des TowerCBCs* gespeicherte Liste *Buffer* angehängt.

3.4.8 $\{\langle PID\rangle, \{multicastM, \langle CommNR\rangle, \langle MessageNR\rangle\}\}$

Die letzte Schnittstelle wird über *cbcast/2* aufgerufen. Diese Funktion überprüft bereits ob die beiden Indizes *CommNR* und *MessageNR* größer 0 sind. Um Angriffe von außen zu vermeiden, wird diese Prüfung erneut vorgenommen. Zusätzlich kann nun geprüft werden, dass die beiden Indizes nicht größer als die Länge der jeweilig zugehörigen Liste sind. Über die Hilfsfunktion *getElementByIndex/2* werden der entsprechende *Prozess der Kommunikationseinheit* und die entsprechende Nachricht ermittelt. Über einen Guard und Pattern Matching kann in einer Zeile überprüft werden, dass die gefundenen Elemente dem korrekten Datentypen entsprechen (siehe Listing 1 Zeile 4). Ist der Test erfolgreich wird die Nachricht an den Empfänger geschickt und mit $\{replycbc, ok\_send\}$ geantwortet. Schlägt der Test fehlt wird mit $\{replycbc, error\_send\}$ geantwortet.

```
1 Comm = getElementByIndex(Registered, CommNR - 1), % Receiver Index beginnt bei 0
2 Result = getElementByIndex(Buffer, MessageNR - 1), % Buffer Index beginnt bei 0
3 case Result of
4     {Message, VT} when is_pid(Comm) ->
5         ...
6         From ! {replycbc, ok_send};
7     _ ->
8         From ! {replycbc, error_send}
9 end,
```

Listing 1: Codeausschnitt multicastM

3.5 Vektoruhr Zentrale/Tower

Aufgabe der *Vektoruhr Zentrale* ist, allen *Prozessen der Kommunikationseinheiten* eine eindeutige Identität zu geben. Diese ist vom Typ Integer und startet bei 1.

3.5.1 init/0

Diese Funktion erzeugt einen Prozess, welcher unter der Prozess ID *vtKLCclockC* registriert wird. Der erzeugte Prozess sieht wie folgt aus:

```
1  loop(Datei, Map) ->
2      receive
3          {getVecID, PID} ->
4              ...
5              loop(...
6          {From, {stop}} when is_pid(From)->
7              ...
8          Any ->
9              util:logging(Datei, "Unknown message: "++util:to_String(Any)++"\n"),
10             loop(Datei, Map)
11     end.
```

Gespeichert wird einmal die in die zu loggende Datei und eine Map. Die Map ist als Liste implementiert, welche Objekte aus Key Value Paaren enthalten. Die Keys sind Prozess IDs, die Values sind Integer.

3.5.2 stop/1

Das Stoppen des Prozesses funktioniert wie in Kapitel 3.2.2.

3.5.3 $\{getVecID, \langle PID \rangle\}$

Diese Schnittstelle empfängt eine Prozess ID. Anschließend wird überprüft ob in der Map des Prozesses bereits ein Objekt mit einer solchen Prozess ID gespeichert ist. Wenn ja, wird die zugehörige Identität zurückgeschickt. Ansonsten wird eine neue Identität erzeugt, mit der Prozess ID zusammen in der Map gespeichert und zurückgeschickt. Diese neue Identität berechnet sich durch $length(Map) + 1$.

3.6 Anwendung

Zur Demonstration der korrekten Ausarbeitung wurde eine Anwendung implementiert. Diese liegt in der *app.erl*. Aufgabe der Anwendung ist, ein Szenario zu demonstrieren (siehe Abb. 18), in welchem zuerst Nachrichten von verschiedenen Nutzern an einen Multicast geschickt werden. Aufgrund eines simulierten Fehlers, werden im Multicast alle Nachrichten in der Reihenfolge vertauscht. Anschließend liest ein dritter Nutzer die Nachrichten aus. Die kausale Ordnung muss bestehen bleiben.

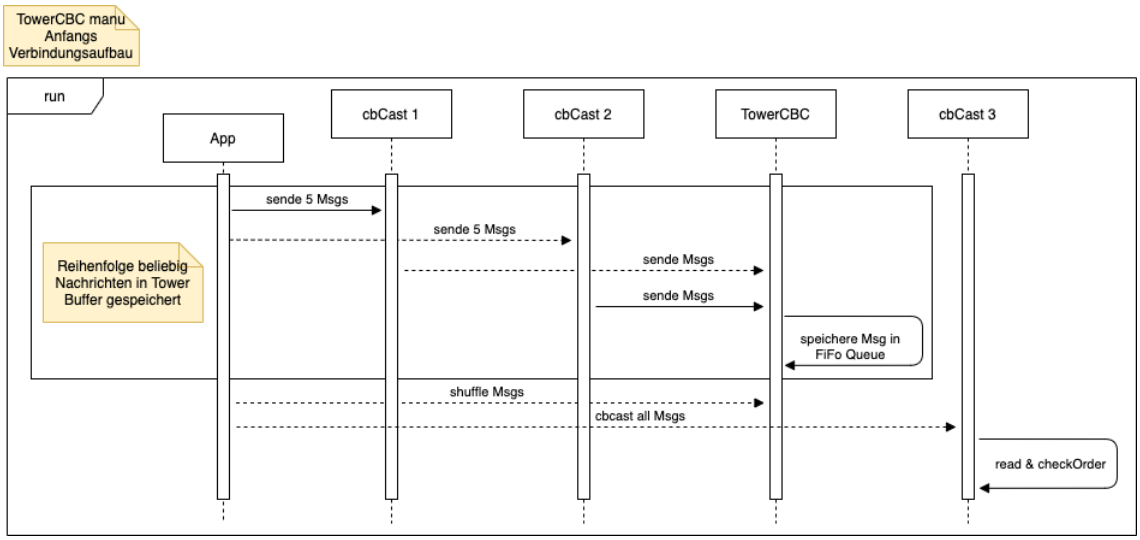

Abbildung 18: Entwurf der Anwendung

Nach erfolgreicher Ausführung sind folgende Logs (siehe Abb. 19) zu sehen:

```
Sending messages...
Sent 1.1 from CbCast1: done
Sent 2.1 from CbCast2: done
Sent 1.2 from CbCast1: done
Sent 2.2 from CbCast2: done
Sent 2.3 from CbCast2: done
Sent 1.3 from CbCast1: done
Sent 1.4 from CbCast1: done
Sent 1.5 from CbCast1: done
Sent 2.4 from CbCast2: done
Sent 2.5 from CbCast2: done
```

```
Forwarding messages...
Forwarded 1 to CbCast3: true
Forwarded 2 to CbCast3: true
Forwarded 3 to CbCast3: true
Forwarded 4 to CbCast3: true
Forwarded 5 to CbCast3: true
Forwarded 6 to CbCast3: true
Forwarded 7 to CbCast3: true
Forwarded 8 to CbCast3: true
Forwarded 9 to CbCast3: true
Forwarded 10 to CbCast3: true
```

```
Shuffle messages...

OldBuffer:
[{"1.1",{1,[1]}},
 {"2.1",{2,[0,1]}},
 {"1.2",{1,[2]}},
 {"2.2",{2,[0,2]}},
 {"2.3",{2,[0,3]}},
 {"1.3",{1,[3]}},
 {"1.4",{1,[4]}},
 {"1.5",{1,[5]}},
 {"2.4",{2,[0,4]}},
 {"2.5",{2,[0,5]}}]
```

```
NewBuffer:
[{"1.5",{1,[5]}},
 {"2.5",{2,[0,5]}},
 {"1.2",{1,[2]}},
 {"1.4",{1,[4]}},
 {"1.3",{1,[3]}},
 {"2.1",{2,[0,1]}},
 {"1.1",{1,[1]}},
 {"2.4",{2,[0,4]}},
 {"2.3",{2,[0,3]}},
 {"2.2",{2,[0,2]}}]
```

```
Read messages...
Read from CbCast3: "2.1"
Read from CbCast3: "1.1"
Read from CbCast3: "2.2"
Read from CbCast3: "1.2"
Read from CbCast3: "2.3"
Read from CbCast3: "1.3"
Read from CbCast3: "2.4"
Read from CbCast3: "1.4"
Read from CbCast3: "2.5"
Read from CbCast3: "1.5"
```

Abbildung 19: Entwurf der Anwendung

Die totale Ordnung ist nicht mehr vorhanden, zum Beispiel ist zuerst die Nachricht ’1.1’ und dann die Nachricht ’2.1’ versendet worden. Ausgeliefert wird aber zuerst die ’2.1’ und

dann die '1.1'. Die kausale Ordnung ist aber noch intakt. Die Reihenfolge der von *cbCast1* und *cbCast2* gesendeten Nachrichten stimmt soweit, wenn man diese getrennt voneinander betrachtet. Nach '2.1' folgt '2.2' und nach '1.1' folgt '1.2', usw..

3.6.1 Implementierungsbesonderheiten

Um die Schnittstellen der jeweiligen Module nicht zu verändern wurden in der *app.erl* verschiedenen Hilfsfunktionen implementiert.

cbCast Das Starten und lokale Registrieren der *Prozesse der Kommunikationseinheiten* erfolgt remote über ein *spawn/2*. Die Hilfsfunktion *startC/1* ist in der *app.erl* implementiert, wird aber auf dem Node des jeweiligen *Prozesses der Kommunikationseinheit* ausgeführt. Somit kann eine lokale Prozess ID erzeugt werden und der Prozess mit einem individuellen Namen registriert werden.

```
1  % app.erl
2
3  % Verbindungsaufbau
4  case net_adm:ping(CbCast1Node) of ...
5
6  % Erzeugen des Prozesses remote auf dem Node des Comms
7  spawn(CbCast1Node, fun() -> startC(CbCastName) end)
8
9  % Hilfsfunktion um Prozess lokal auf dessen Node zu registrieren
10 startC(Name) ->
11     CommCBC = cbCast:init(),
12     erlang:register(Name,CommCBC),
13     ...
```

towerCBC Der *Tower* empfängt für die Anwendung eine zusätzliche Schnittstelle $\{From, \{shuffleMessages\}\}$. In dieser werden alle Nachrichten, welche in der Variable *Buffer* gespeichert, sind gemischt. Aufgerufen wird die Schnittstelle vom Prozess der Anwendung aus.

```
1  % app.erl
2
3  Tower ! {{app, erlang:node()}, {shuffleMessages}},
4      receive
5          {replycbc, ok_shuffleMessages, {OldBuffer, NewBuffer}} -> ...
6      after 1000 -> ...
7      end.
```

4 Analyse

4.1 Korrektheitsbeweis

4.1.1 Betrachtung des Algorithmus

In Abb. 20 ist der Algorithmus für einen kausalen Multicast beschrieben.

Seien p1, p2, . . . , pn Gruppenmitglieder
Sei VTj eine Vektoruhr des Gruppenmitglieds pj
VTj[i] stellt die Anzahl der von pi gesendeten Multicast-Nachrichten dar, die potentiell kausal zu der letzten an pj gelieferten Nachricht geführt haben

1. Initialisiere alle VTi mit einem Nullvektor
2. Wenn pj eine Nachricht per multicast versendet
   - inkrementiere VTj[j] um eins
   - Füge aktuelles VTj als Vektorzeitstempel vt zur Nachricht hinzu
3. Nachricht ist bei pi ausgelieferbar, wenn (sei vt der Zeitstempel der Nachricht)
   - Nachricht muss die nächste in der von pj erwarteten Reihenfolge sein: VTi[j]+1 = vt[j] (,d.h. VTi[j] < vt[j])
   - Alle kausal vorhergehenden Nachrichten, die pj zugestellt wurden, wurden auch pi zugestellt: VTi[k] ≥ vt[k] (für k ≠ j)
4. Wenn die Nachricht bei pi zugestellt/ausgeliefert wird, synchronisiere den lokalen Zeitstempel VTi von pi mit dem in der Nachricht empfangenen Zeitstempel vt (jeweils das Maximum)

Abbildung 20: Algorithmus für einen kausalen Multicast [Kla24]

Aus diesem gehen die folgenden wesentlichen Merkmale hervor:

- Alle Prozesse werden mit einem sogenannten Nullvektor initialisiert,
- Inkrementiere die eigene Vektoruhr $VTj[j]$ um 1 beim Versenden einer Nachricht,
- Eine Nachricht ist in einem Prozess pi auslieferbar, wenn
  1. dessen Vektoruhr an der Position j (j = Identität von pj) um 1 höher ist als der Zeitstempel der Nachricht: $VTi[j] + 1 = vt[j]$,
  2. alle Nachrichten, die kausal vor der aktuellen Nachricht stehen, bereits bei pi angekommen und verarbeitet worden sind. Sichergestellt wird dies durch die Bedingung $VTi[k] \geq vt[k]$ für $k \neq j$.
- Synchronisiere den lokalen Zeitstempel des Prozesses beim Ausliefern einer Nachricht mit dem der ausgelieferten Nachricht.

Initialisierung mit Nullvektor Die Eindeutigkeit jedes Prozesses wird über die Vektoruhr Zentrale kontrolliert. Durch das Mapping der Prozess ID auf die Vektoruhr ID hat jeder Prozess eine eindeutige Vektoruhr ID. Die Nullvektoren haben entsprechend alle eine unterschiedliche Länge und sind ausschließlich mit Nullen gefüllt.

Inkrementierung beim Versenden Die Inkrementierung der Vektoruhr ist im *Prozess der Queues* implementiert. Die entsprechende Schnittstelle wird ausschließlich in der *receive*-Schnittstelle der *send* Funktion (siehe Kapitel 3.2.3) aufgerufen. Das Versenden eine Nachricht ist nur über diese Funktion möglich.

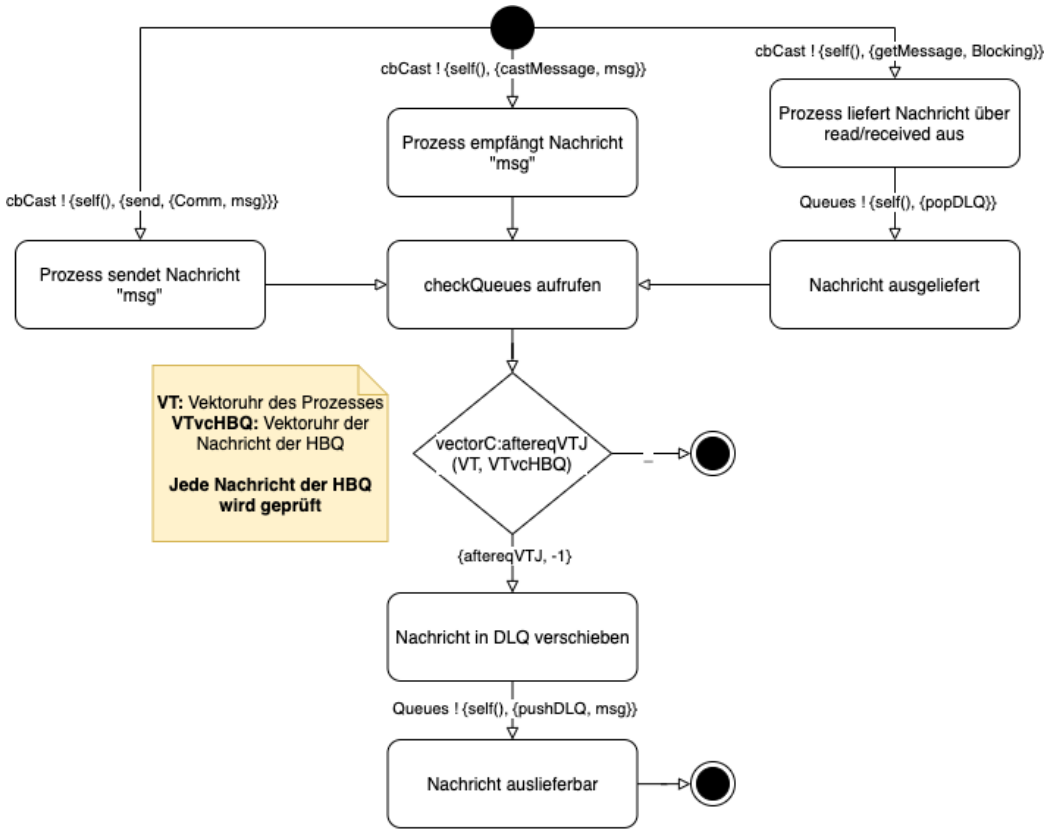

Abbildung 21: Auslieferbarkeit

Auslieferung der Nachrichten In Abb. 21 ist dargestellt, wann genau Nachrichten ausgeliefert werden können und wann Nachrichten ausgeliefert werden.
Auslieferbar sind ausschließlich Nachrichten, welche in der *Delivery Queue* gespeichert sind. Geprüft ob Nachrichten in die *Delivery Queue* verschoben werden dürfen, wird, bei jeder Veränderung der *Holdback* oder *Delivery Queue* über die Funktion *checkQueues*. Verschoben werden ausschließlich Nachrichten dessen Vektoruhr an der Position j (j = Identität des Prozesses) um 1 kleiner sind als der Zeitstempel des Prozesses: $VTi[j]+1 = vt[j]$. Dies wird durch die Funktion *vectorC:aftereqVTJ/2* sichergestellt.
Die Kausalität bleibt durch die Implementierung der *Delivery Queue* als FiFo-Queue bestehen. Die einzige Möglichkeit Nachrichten auszuliefern bieten die Schnittstellen *read/1* und *received/1*.

Synchronisierung beim Ausliefern Auch die Synchronisierung der Vektoruhr des *Prozesses der Kommunikationseinheit* mit der Vektoruhr einer ausgelieferten Nachricht ist in dem *Prozess der Queues* implementiert. Diese Schnittstelle wird ausschließlich in der *receive*-Schnittstelle der *read*, bzw. *received* Funktion (siehe Kapitel 3.2.4) aufgerufen. Sowohl *read* als auch *received* greifen auf diese *receive*-Schnittstelle zu. Eine andere Möglichkeit als über eine diese beiden Funktionen Nachrichten auszuliefern ist nicht implementiert.

4.1.2 Nebenläufigkeit

In der Aufgabenstellung sind drei Tests aufgeführt, welche die Kausalität des Algorithmus testen. Anhand der geloggten Ergebnisse nach Ausführung der *testCBC:test()* Funktion auf dem Node des *Towers* und anhand der Tests in Abb. 22 kann nun die Nebenläufigkeit demonstriert werden. Relevant sind hierbei vor allem die log-Dateien der *Prozesse der Kommunikationseinheiten* und die log-Datei des Tests *testCBC*.log*.

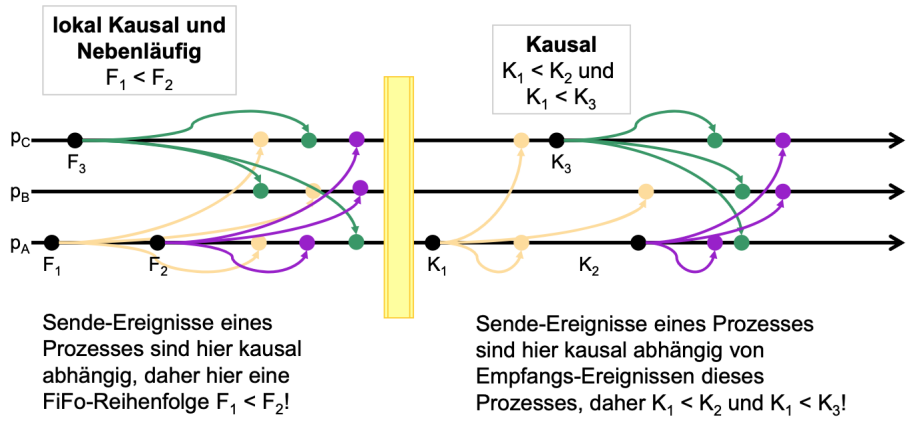

Abbildung 22: Kausalitätstest 1 und 2 [Kla24]

Im linken Test *'lokal Kausal und Nebenläufig'* werden die Nachrichten F1 und F2 nebenläufig zueinander von zwei verschiedenen Prozessen aus gesendet. Kausal abhängig ist hier nur die Nachricht F2 von F1. F2 wird erst verschickt, wenn F1 verschickt wurde. In Prozess B ist also wichtig, dass F1 vor F2 ausgeliefert wird. Wann genau F3 ausgeliefert wird ist nicht weiter relevant.

```
8  Nachrichten 1: beige, 2: gruen, 3: lila an Multicast gesendet.
9  Nachrichten per Multicast zugestellt: A lila; B gruen; C beige,gruen,lila.
10     >>>Empfange Nachrichten: A "beige" (beige); B "gruen" (gruen); C "gruen" (gruen);
11 Nachrichten per Multicast zugestellt: A beige; B lila.
12     >>>Empfange Nachrichten: A "lila" (lila); B null (null); C "beige" (beige);
13 Nachrichten per Multicast zugestellt: A gruen; B beige.
14     >>>Empfange Nachrichten: A "gruen" (gruen); B "beige" (beige)|"lila" (lila); C "lila" (lila);
15 Test f@r lokal Kausal und Nebenl@ufig beendet.
```

Abbildung 23: Ergebnisse 3

In Abb. 23 sind die Logs des Tests zu sehen. Die Nachrichten kommen in folgender Reihenfolge bei den jeweiligen Prozessen an:

- **Prozess A** $F1 \rightarrow F3 \rightarrow F2$
- **Prozess B** $F3 \rightarrow F1 \rightarrow F2$
- **Prozess C** $F3 \rightarrow F1 \rightarrow F2$

$F1 < F2$ ist bei allen Prozessen eingehalten. F3 ist bei Prozess A und bei Prozess B an unterschiedlichen Stellen. Dies zeigt sowohl das Einhalten der kausalen Ordnung, als auch die Nebenläufigkeit.

4.2 Komplexitätsanalyse

4.2.1 Kommunikationseinheit

Die Komplexität der Kommunikationseinheit hängt im wesentlichen von der Funktion *checkQueues/4* ab. Diese greift auf folgende Funktionen zu:

- *vectorC:myVTvc/1* mit einer Komplextität von $O(1)$
- *vectorC:aftereqVTJ/2* mit einer Komplexität von $O(m+n)$ (siehe Kapitel 4.2.2)
- *pushDLQ/2* mit einer Komplexität von $O(1)$
- *removeFromList/2* mit einer Komplexität von $O(n^2)$

Durch die Funktionen *vectorC:aftereqVTJ/2* und *removeFromList/2* ergibt sich $O(n^2) + O(n^2) = O(n^2)$. Die rekursive Tiefe ist $O(n)$, wodurch eine Gesamtkomplexität von $O(n) * 0(n^2) = O(n^3)$ für die Funktion *checkQueues/4* resultiert.

4.2.2 Vektoruhr-ADT

Die beiden komplexeren Funktionen dieses Moduls sind *compVT/2* und *aftereqVTJ/2*.

compVT/2 Die genutzte Hilfsfunktion *padWithZeros/2* hat eine Komplexität von $O(m)$. m ist die Länge des resultierenden Vektors.
Die zweite genutzte Hilfsfunktion ist *compareLists/2*. Es werden zwei Listen elementweise verglichen, wodurch eine Komplexität von $O(n)$ resultiert. n ist die Länge der Vektoren, welche durch das Padding die gleiche Länge haben.
Die Gesamtkomplexität dieser Funktion ist also $O(m+n)$. m und n sind die Längen der beiden übergebenen Vektoren.

aftereqVTJ/2 Die erste genutzte Hilfsfunktion ist *removeJ/2*. Diese nutzt die im vorgehenden Paragraphen beschriebene Hilfsfunktion *padWithZeros/2*. Danach wird durch eine Liste iteriert ($O(n)$) und an einen Akkumulator angehängt, was zu $O(k)$ führt. k ist die Länge des Akkumulators. Es folgt eine Komplexität von $O(n) + O(n) + O(n^2) + O(n^2) = O(n^2)$.
Die zweite genutzte Funktion ist *compVT/2*. Wie auch im vorherigen Paragraphen beschrieben ist dessen Komplexität $O(m+n)$, bzw. $O(n)$, da die beiden übergebenen Vektoren die gleiche Länge haben.
Es ergibt sich eine Gesamtkomplexität von $O(n^2) + O(n) = O(n^2)$.

4.2.3 Ungeordneter Multicast

Die maximale Gesamtkomplexität aller Schnittstellen in diesem Modul ist $O(n)$.

4.2.4 Vektoruhr Zentrale/Tower

Die maximale Gesamtkomplexität aller Schnittstellen in diesem Modul ist $O(n)$.

5 Fazit

5.1 Von der Theorie zur Praxis

Zu Beginn der Ausarbeitung waren viele Fragen offen. Als Gruppe haben wir die Vielzahl an Schnittstellen und den neuen Algorithmus diskutiert und technische Fragen geklärt. Literatur spezifisch zum Thema *CBCAST*-Algorithmus zu finden, hat sich als schwieriger herausgestellt.
Nachdem eine erste Idee entstanden ist, hat das Entwerfen verschiedener Ablauf- und Sequenzdiagramme vieles erleichtert. Parallel wurde in Einzelarbeit der Entwurf geschrieben und der Code implementiert. Die Realisierung ist nachträglich aufgearbeitet worden.
Während des gesamten Prozesses wurden immer mehr Fragen geklärt, wodurch das Schreiben der Analyse und die Implementierung der Anwendung vergleichsweise wenig Zeit gekostet haben.

5.1.1 Erlang

Die Programmiersprache *Erlang* hat sich für diese spezifische Aufgabenstellung als sehr geeignet herausgestellt. Das Pattern Matching und die Guards ersparen tief verschachtelten Code. Hier hätte ich mir dennoch gewünscht, dass das Nutzen eigener Guards, wie zum Beispiel *vectorC:isVT/1* möglich gewesen wäre. Auch, dass bei mehreren *if* und *switch* Statements hintereinander geschaltet werden müsst, trägt nicht zu einer besseren Lesbarkeit des Codes bei.
Die Möglichkeiten von *Erlang* auf verschiedenen Nodes zu arbeiten und ein verteiltes System mit vielen Nebenläufigkeiten zu simulieren ist für mich bisher einzigartig gewesen und hat mich positiv überrascht.
Das Arbeiten mit einer funktionalen Programmiersprache bietet aus meiner Sicht viele Vorteile, gerade beim Debuggen. Bis auf die Prozessvariablen gibt es keine versteckten Zustände und auch keine falschen Referenzen.
Im direkten Vergleich ist eine Java Anwendung bei asynchronen Threads quasi nur über *prints* zu debuggen. Der *Erlang*-Debugger, welcher für jeden Node einzeln gestartet werden kann, hat teilweise Probleme mit den *receive* Timeouts, löst aber das Problem der Java Anwendung.

5.2 Bewertung des Algorithmus anhand der Anwendung

Die Anwendung simuliert folgendes Verhalten:

1. Zwei verschiedene Prozesse schicken Nachrichten zum Multicast
2. Die Reihenfolge der Nachrichten wird im Buffer des Multicasts verändert
3. Die Nachrichten werden zu einem dritten Prozess weitergeleitet.
4. Der dritte Prozess liefert die Nachrichten in korrekter Reihenfolge nach kausaler Ordnung aus

Dieser Ablauf kann zum Beispiel mit dem Verschicken mehrerer Mails von zwei verschiedenen Usern an einen dritten User verglichen werden. Die Reihenfolge der Mails, betrachtet pro Sender, wird weiterhin eingehalten. Auch wenn vom Email-Provider Nachrichten vertauscht werden, durch zum Beispiel einen internen Fehler oder einen Angriff von außen.

Vor allem die Robustheit des Algorithmus wird durch die Anwendung bewiesen. Trotz der zusätzlichen Vertauschung der Nachrichten bleibt die kausale Ordnung bestehen.
Der *CBCAST*-Algorithmus ist also eine effiziente und mittelschwer implementierbare Lösung zum Verschicken von Nachrichten, wenn die kausale Ordnung bewahrt werden muss. Wichtig ist hierbei, dass nur die kausale und nicht die totale Ordnung bestehen bleibt. Durch das Nutzen der logischen Vektoruhren ist sichergestellt, dass die einzelnen Teilnehmer nicht auf Systemzeiten angewiesen sind.

5.3 Schwächen des Algorithmus

5.3.1 *back-* und *postdating*

Der Algorithmus ist anfällig für sogenannte *backdating* und *postdating* Attacken (siehe [RBG92]). Eine Nachricht ist *backdated*, wenn eine Nachricht m' kausal nach einer Nachricht m liegt, das aber auf einem anderen Node anders herum der Fall ist. Dieser Ansatz bietet die Möglichkeit einen Angriff von außen zu simulieren und das System zu blockieren. Mit einer sicheren Firewall und anderen Schutzmechanismen sind diese Attacken aber abzuhalten. Das System ist dadurch also genau so sicher, wie die meisten anderen Systeme auch.

5.3.2 Skalierbarkeit

CBCAST kann in kleineren Systemen gut funktionieren. In sehr großen verteilten Systemen wiederum kann es schwierig sein, die Effizienz und Leistung aufrechtzuerhalten. Die Komplexität der Nachrichtenverwaltung und der Speicherbedarf steigen abhängig von der Anzahl der beteiligten Prozesse (siehe [SR19]).
Für folgende Arbeiten wäre die Skalierung der Prozesse ein interessanter Ansatz das implementierte System wiederzuverwenden.

5.3.3 Latenz

Alle Prozesse müssen sicherzustellen, dass sie alle abhängigen Ereignisse in der richtigen Reihenfolge empfangen und verarbeiten. Dafür müssen sie möglicherweise warten, bis sie alle benötigten Informationen erhalten haben. Dies kann die Latenz erhöhen, besonders in Netzwerken mit hoher Latenz oder unzuverlässigen Verbindungen.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meiner Hausarbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie die aus fremden Quellen (dazu zählen auch Internetquellen) direkt oder indirekt übernommenen Gedanken oder Wortlaute als solche kenntlich gemacht habe. Zudem erkläre ich, dass der zugehörige Programmcode von mir selbständig implementiert wurde ohne diesen oder Teile davon von Dritten im Wortlaut oder dem Sinn nach übernommen zu haben. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Hamburg, 09.07.24
(Ort, Datum)

K.
(Unterschrift)

Abbildungsverzeichnis

Literaturverzeichnis

[Bab12] Seyed Morteza Babamir. „Specification and verification of reliability in dispatching multicast messages“. In: *The journal of supercomputing* 63.2 (2012), S. 612. URL: https://doi.org/10.1007/s11227-012-0834-2.

[BC91] Kenneth Birman und Robert Cooper. „The ISIS project: real experience with a fault tolerant programming system“. In: *SIGOPS Oper. Syst. Rev.* 25.2 (Apr. 1991), S. 103–107. ISSN: 0163-5980. DOI: 10.1145/122120.122133. URL: https://doi.org/10.1145/122120.122133.

[Kla24] Christoph Klauck. *Aufgabe HA*. 2024.

[RBG92] M. Reiter, K. Birman und L. Gong. „Integrating security in a group oriented distributed system“. In: *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. 1992, S. 18–32. DOI: 10.1109/RISP.1992.213273.

[Sch21] Leon Schwarzenberger; Kristoffer Schaaf. „Entwurf Praktikum 2, Algorithmen und Datenstrukturen“. Dez. 2021.

[SR19] Válter Santos und Luís Rodrigues. „Localized Reliable Causal Multicast“. In: *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. 2019, S. 1–10. DOI: 10.1109/NCA.2019.8935065.

[TW11] Andrew S. Tanenbaum und David Wetherall. *Computer Networks*. 5. Aufl. Boston: Prentice Hall, 2011. ISBN: 978-0-13-212695-3. URL: https://www.safaribooksonline.com/library/view/computer-networks-fifth/9780133485936/.