

# **Infrastruktur für einen kausalen Multicast mittels Vektoruhr**

Belal-Ahmad Karimzai

2. Juli 2021

Verteilte Systeme - Referat  
Prof. Dr. Christopher Klauck  
Hochschule für Angewandte Wissenschaften (HAW) Hamburg  
SoSe 2021

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>3</b>
<b>1 Einführung</b>	<b>4</b>
1.1 Einordnung - logischer Uhren, Gruppenkommunikation und kausaler Multicast	5
<b>2 Algorithmus: Causal Broadcast (CBCAST) und Vektoruhren</b>	<b>8</b>
<b>3 Entwurf</b>	<b>9</b>
3.1 Vorgehensweise . . . . .	9
3.2 Bausteinsicht: Komponenten . . . . .	9
3.2.1 Datenstrukturen . . . . .	11
3.2.2 Definition der Schnittstellen . . . . .	11
3.3 Laufzeitschicht . . . . .	16
3.3.1 Kommunikationseinheit . . . . .	16
3.3.2 Methoden der Vektoruhr-ADT . . . . .	22
3.3.3 Zusammenfassung . . . . .	25
3.4 Reflexion der Vorgehensweise . . . . .	25
3.5 Sicherstellung der Anforderung . . . . .	27
<b>4 Theorie und Praxis</b>	<b>28</b>
4.1 Analyse: Vor und Nachteile von Vektoruhren . . . . .	28
4.2 Anwendungsszenario . . . . .	30
<b>5 Fazit - Reflexion</b>	<b>31</b>
<b>Literatur</b>	<b>33</b>

## Abbildungsverzeichnis

1	Bausteinsicht: Infrastruktur der Anwendung . . . . .	9
2	Middleware-Architektur der Kommunikationseinheit . . . . .	16
3	Laufzeitschicht: Vorbereitungsphase . . . . .	17
4	Laufzeitschicht: Initialisierungsphase . . . . .	18
5	Kommunikation beim Sendeereignis . . . . .	19
6	interne Verarbeitung beim Erhalt einer Multicast-Nachricht . . . . .	20
7	Blockierende Auslieferung einer Nachricht an anfragenden Client (read) . . .	21
8	Terminierung einer Kommunikationseinheit . . . . .	22
9	isVT(VT) - Überprüfung der korrekten Datenstruktur . . . . .	23
10	compVT(VT1, VT2) - Vergleichsmethode für Vektoruhren . . . . .	24
11	aftereqVTJ(VT, VTR) - Überprüfung, ob VTR auslieferbar ist an die DLQ .	25
12	Beispielszenario zwischen 3 Prozessen: A, B und C . . . . .	28

# 1 Einführung

Ein fundamentale Schwierigkeit im Umgang mit und im Entwickeln von verteilten Systemen ist es, die Nachvollziehbarkeit von Nachrichten und Ereignissen nach ihrem zeitlichen Verlauf ordnungsbasierend sicherzustellen. Durchaus fallen auch noch weitere Anforderungen an verteilte Systeme an, wie dem Aspekt der Verfügbarkeit, dass es im System keinen Single Point of Failure geben sollte oder dem Aspekt der Kontrolle der Nebenläufigkeit, um beispielsweise Deadlocks zu vermeiden oder den gemeinsamen Zugriff auf Ressourcen zu koordinieren. Den Problemen gegenüber steht die Hoffnung mit Skalierbarkeit und Performance zu punkten und von weiteren Vorteilen der verteilten Anwendung zu profitieren. So ist es als Entwickler und Informatiker notwendig Strategien zu kennen, um verteilte Anwendungen konsistent, zuverlässig und robust zu realisieren.

Im Rahmen dieser Arbeit wird sich der Aspekt von verteilten Anwendungen um Konsistenzerhaltung in Gruppenkommunikationen betrachtet. Das Ziel hierbei ist es, die Fähigkeit zu erwerben, die verteilte Anwendung mittels logischer Uhren zu synchronisieren, um so einen konsistenten Zustand zwischen den Teilnehmer zu erreichen. Die zentrale Frage lautet: Wie kann in einer Gruppenkommunikation mittels Multicast gewährleistet werden, dass die Nachrichten, die untereinander ausgetauscht werden, in einer zuverlässigen kausalen Ordnung beim Endanwender verarbeitet beziehungsweise angezeigt werden.

Im Rahmen der Veranstaltung "Verteilte Systeme" im Bachelor-Studiengang Angewandte Informatik wurde von Herrn Prof. Klauck eine Multicast-Zentrale zur Verfügung gestellt, die Nachrichten ungeordnet zu beliebigem Zeitpunkt an die Teilnehmer des Netzwerkes verteilt. Die Aufgabe besteht nun darin, für die Gruppenteilnehmer eine Middleware zu implementieren, dessen Struktur und Verhalten es erlaubt, Nachrichten in kausalen Zusammenhang zu bringen und nach kausaler Ordnung an den Endanwender, in diesem Fall den Gruppenteilnehmer, auszuliefern. Diese Umsetzung erfolgt durch das Kausale Broadcast-Protokoll, auch bekannt unter der Abkürzung CBCAST und in der Literatur auch auffindbar als kausaler Multicast-Protokoll. Das Kernelement dieser Vorschrift wird die gewählte Datenstruktur, die Vektoruhr und die Unterscheidung auftretender Ereignisse sein. Mittels der Vektoruhr können dann Rückschlüsse auf die Kausalität der Nachricht getroffen werden. Zusätzlich zum Ordnungskriterium werden auch Aussagen über den Zuverlässigkeitsgrad des Protokolls erwähnt.

Um nun all die neu eingeführten Begriff in Zusammenhang zu bringen, wird anfangs eine theoretische Einführung in das Themenbereich der logischen Uhren sowie dem Konzept der Gruppenkommunikation und des kausalen Multicast gegeben. Der Fokus besteht hier, die Begriffe thematisch einzuordnen und zu definieren, und gleichzeitig eine Motivation für diese Themenbereiche zu entwickeln.

Anschließend wird der CBCast-Algorithmus vorgestellt. Hierbei werden auch die ersten Bezüge zu der im späteren Verlauf vorgestellten Anwendung gesetzt. Darauf folgend wird die Anwendung mittels der 4 Sichten nach Kruchten skizziert, wobei hier nur die Bausteinsicht und die Laufzeitschicht präsentiert werden. In diesem Zusammenhang werden die in der Implementierung genutzten Datenstrukturen vorgestellt und die Schnittstellen für die Kommunikationen zwischen den unterschiedlichen Akteure vorgestellt. Abschließend folgt eine Zusammenfassung der technischen Umsetzung und die dort auftretenden Probleme. Zudem

nutze ich in diesem Abschnitt die Chance, die Vor- und Nachteile der Vektoruhr zu beschreiben. Zu guter Letzt werden Anwendungsszenarien vorgestellt, um einen praktischen Bezug zur Theorie zu entwickeln.

## 1.1 Einordnung - logischer Uhren, Gruppenkommunikation und kausaler Multicast

**Zeit und Ursache-Wirkung-Beziehungen** Zeit ist ein Mittel, damit Menschen sich verständigen, Aktionen und Vorhaben koordinieren und planen, sowie Ereignisse auf einer Zeitachse einordnen können. Betrachtet man sich folgenden Auszug eines Gruppenchat aus einer verteilten Anwendung im Sinne einer Gruppenkommunikation:

A: Die Vorlesung findet statt. (9.00 Uhr)  
B: Findet morgen die VS-Vorlesung statt? (9.01 Uhr)  
C: Donnerstag um 8:15 Uhr. (9.02 Uhr)  
B: Dann bis morgen. (9.04 Uhr)  
A: Wann findet morgen die Vorlesung nochmal statt? (9.03 Uhr)

erkennt man, dass die Reihenfolge der Nachrichten nach logischem Verständnis nicht sinnvoll erscheint. Einer Nachricht ist ein Zeitstempel aufgedrückt, womit die Anwendung die Chance bekommt die Nachrichten nach ihrem zeitlichen Verlauf einzuordnen und zu sortieren, unter der Annahme, dass die Ereignisse hintereinander stattgefunden haben. Jedoch gibt der Chat die logischen Zusammenhänge nicht korrekt wieder, denn die erste Nachricht ist eine Antwort bzw. eine Wirkung auf eine ausgehende Frage bzw. Ursache gewesen. Demnach muss der ersten Nachricht eine Frage zeitlich vorausgegangen sein. Nun stellt sich die Frage, warum die Nachrichten trotz Zeitstempel kausal in einer falschen Reihenfolge zu sehen ist.

Im Wesentlichen besteht das Problem, dass die verteilte Anwendung kein zentrales Verständnis von Zeit besitzt, wie in einem zentralisierten System. Die Zeitstempel der verteilten Anwender weichen aufgrund von physischen Eigenschaften, wie dem Offset oder dem Drift der Zeit, voneinander ab. So kann es sein, dass aufgrund der Timestamps eine Antwortnachricht vor einer Fragenachricht datiert ist.

Ein weiteres Problem ist, dass die Nachrichten bei einer Gruppenkommunikation über ein Netzwerk transportiert werden und der Qualität sowie der Belastung des Kommunikationsmedium unterlegen sind. So kann es sein, dass Nachrichten zu einem früheren Zeitpunkt abgesendet worden sind, aber aufgrund des Delays verzögert ausgeliefert werden.

So ergeben sich die Fragen, mit welcher Zeiteinheit ein kausaler Zusammenhang garantiert werden kann und wie die Zustellungslogik der Middleware aussehen muss, um auch verzögerte Nachrichten in einem korrekten kausalen Zusammenhang einzuordnen.

Trivialerweise könnte man versuchen die Uhren der jeweiligen Prozesse zu synchronisieren und damit ein gleiches Verständnis von Zeit aufbauen. Jedoch bedarf Uhrensynchronisation einen hohen Konfigurationsaufwand und Kommunikationsaufwand über das Netzwerk und ist typischen Fehlerquellen in Netzwerkkommunikationen ausgesetzt, wie dem Verlust von Synchronisierungspaketen oder fehlerhaften Werten in den Synchronisierungspaketen und

weiterer Faktoren. Dieser Ansatz verfolgt eine Echtzeituhren-Synchronisation über paketorientierte Kommunikation und werden beispielsweise im Network Time Protokoll standardisiert und definiert.

Ein anderer Ansatz besteht die Zeit sich abstrakter aus einer rein logischen Perspektive zu betrachten. Charakteristisch für solche logische Uhren ist ihr einfacher Ansatz, Kausalitäts-Beziehungen von Nachrichten festzuhalten. Hierfür wird streng monoton ein positiv ganzzahliger Ereigniszähler inkrementiert. Die Aufgabe besteht nun für jeden Prozesse einen solchen Ereigniszähler zu verwalten und bei Eintreten von Ereignissen den Zähler zu inkrementieren bzw. unter gewissen Umständen die logische Uhren miteinander zu synchronisieren. Diese Koordination wurde von dem Mathematiker Lamport im gleichnamigen Lamport-Algorithmus festgehalten. In der Literatur unterscheidet man zwischen zwei logischen Uhren: der Lamport Uhr und der Vektoruhr. Dieser Unterschied wird im Folgenden herausgearbeitet.

Die logische Zeit ist eine einfache Abbildung von Zeitpunkten beziehungsweise Ereignissen, die total geordnet sind, und somit sind die in der Menge auftretenden Zeitpunkte miteinander vergleichbar, denn jedem Ereignis wird ein eindeutiger Ereigniszähler zugewiesen. Somit kann folgendes ausgedrückt werden, wobei  $L$  die Abbildung von Ereignissen auf einen eindeutigen positiven Ereigniszähler ist:  $L(e_1) < L(e_2)$

Ein Beispiel bietet uns die eingangs erwähnte Chat-Applikation, dass eine Antwortnachricht logisch gesehen zu einem späteren Zeitpunkt als die Fragenachricht auftreten muss, so kann man Folgendes definieren, dass zuerst eine Fragenachricht geschieht, und dann eine Antwortnachricht aus ihr resultiert. Diese Relation ist in der Literatur als happens-before Relation bekannt und wird mathematisch mit dem Symbol  $\rightarrow$  definiert. Sie ist eine einfache Ursache Wirkung-Beziehung und wurde vom Mathematiker Leslie Lamport definiert

Wenn diese Bedingung gilt, steht auch fest, dass der Vergleich beider Ereignisse in folgendem Zusammenhang stehen  $request \rightarrow reply$ , dann  $L(request) < L(reply)$ . Damit nun auch die Umkehrung gilt, sodass man über den Vergleich der Ereigniszähler Ableitungen treffen kann, ob Ereignisse in einer happens-before Relation stehen, bedarf es mehr Informationen als nur seinen eigenen Ereigniszähler zu verwalten. Denn bereits bei 3 teilnehmenden Akteure kann es sein, dass  $L(e_1) < L(e_2)$  gilt, aber anhand der Zeitstempel kann noch nicht gesagt werden, dass  $e_1$  eine Ursache für  $e_2$  ist. Hierfür ist es notwendig, dass der Verlauf der Ereignisse als Historie abgespeichert wird und die Datenstruktur hierfür bezeichnet man als Vektoruhr. Mit Hilfe einer solchen Konstruktion kann man nun Rückschlüsse auf die Kausalität von Ereignissen treffen. Der Vektor hält in Form der Ereigniszähler lokal für einen Prozess fest, welche Nachrichten bisher von einem Prozess gelesen bzw. empfangen worden sind oder abstrakt welche Ereignisse von einem Prozess bereits eingegangen sind.

Wichtig zu erwähnen ist noch, dass zwei Ereignissen lokal nicht der gleiche Zeitpunkt zuweisbar ist, da jedes Ereignis lokal einen eindeutigen Ereigniszähler besitzt. Darüber hinaus wird mit der Lamport-Uhr nur die schwache Konsistenz erreicht, da zu einem die Umkehrung nicht gilt und zum anderen werden nebenläufige Nachrichten nicht erkannt, wobei nebenläufig hier bedeutet, dass Nachrichten kausal voneinander unabhängig sind. Eine kausale Ordnung ist irreflexiv, anti-symmetrisch, zyklonfrei und transitiv.

Durch Erfüllen der kausalen Bedingung darf eine Nachricht nur dann ausgeliefert werden, wenn potentiell keine Nachricht mehr vorhanden sind, die der Entstehung der nun ausste-

henden Auslieferung vorausgegangen war. Somit wird die Ursache-Wirkung-Beziehung in der Delivery berücksichtigt. Für unsere Chat-Applikation würde das heißen:

A: Wann findet morgen die Vorlesung nochmal statt? (9.03 Uhr)

C: Donnerstag um 8:15 Uhr. (9.02 Uhr)

B: Findet morgen die VS-Vorlesung statt? (9.01 Uhr)

A: Die Vorlesung findet statt. (9.00 Uhr)

B: Dann bis morgen. (9.04 Uhr)

**Gruppenkommunikation** Was zeichnet nun eine solche Gruppenkommunikation aus? Die Kommunikation findet zwischen mehreren verteilten Teilnehmern (hier Prozessen) innerhalb einer Anwendung statt. Demnach können Nachrichten als Punkt-zu-Punkt (unicast), Punkt-zu-Gruppe (multicast) oder als Punkt-zu-Alle (broadcast) verteilt werden. Den Teilnehmer einer Gruppenkommunikation wird eine eindeutige ID vergeben, worüber sie identifizierbar und adressierbar sind. In der Literatur wird auch noch zwischen geschlossener - feste Anzahl an Teilnehmer - und offener - dynamisch variierende partizipierende Teilnehmer - Gruppenkommunikation unterschieden. In der hier vorgestellten Anwendung wird von einer einfachen offenen Kommunikation ausgegangen.

An der Gruppenkommunikation werden hauptsächlich zwei Anforderungen gestellt: dem Zuverlässigkeits- und Ordnungsgrad der Nachrichten. Der Ordnungsgrad bezieht sich auf die Sortierung der Reihenfolge der Nachrichten. Hierbei unterscheidet man zwischen totaler (atomar), kausaler und der FIFO Ordnung. Auch kann es durchaus passieren, dass keine Garantie für eine Ordnung gegeben wird, somit sind keine Synchronisierungsmechanismen etabliert. Ein totales Ordnungsschema kennzeichnet sich dadurch aus, dass die Reihenfolge der Nachrichten nicht nur kausal korrekt zueinander stehen, sondern auch dass die Nachrichtenreihenfolge bei allen Teilnehmern bei der Auslieferung identisch aussieht. Bei einer kausalen Ordnung kann sich die Auslieferung der Nachrichtenreihenfolge bei den Teilnehmern durchaus unterscheiden, ohne dabei die kausale Beziehung der Nachrichten zu verletzen.

Der Zuverlässigkeitsgrad gibt Auskunft darüber mit welcher Garantie Nachrichten an die Gruppenmitglieder ausgeliefert werden können. Mit Zuverlässigkeit wird auch ausgedrückt, dass Nachrichten nicht dupliziert werden und auch nicht fehlerhaft ausgeliefert werden.

Der Zuverlässigkeitsgrad ist beispielsweise bei verteilten, replizierten Datenbank grundlegend, damit die Zustände der Replikate konsistent sind, wenn modifizierende Operationen an den Replikaten ausgeführt werden, müssen die anderen Replikate hierüber informiert werden. So ist es notwendig, dass alle Repliken über Nachrichtenaustausch informiert werden, um in einem einheitlichen konsistenten Zustand zu sein. Hierbei auftretende Probleme sind Abstürze von einzelnen Gruppenmitgliedern und wenn aufgrund von Skalierungsmaßnahmen neue Replikate auftauchen, müssen Informationen zur Konsistenzerhaltung miteinander ausgetauscht werden. In der hier vorgestellten Anwendung gehen wir von einer absoluten zuverlässigen Übertragung von Nachrichten. Diese Verantwortlichkeit unterliegt der vorgegebenen Multicast-Zentrale.

Zusammenfassend wurden die theoretischen Grundzüge und die Motivation der logischen Uhr und der kausalen Ordnung gesetzt. Im nächsten Abschnitt wird das kausale Broadcast-Protokoll beschrieben.

## 2 Algorithmus: Causal Broadcast (CBCAST) und Vektoruhren

Es liegen  $P_1$  bis  $P_n$  Gruppenmitglieder vor, die Nachrichten miteinander austauschen. Jedes Gruppenmitglied  $P_j$  besitzt eine eigene Vektoruhr mit  $VT_i$ , wobei  $j$  eine eindeutige Gruppenid ist und  $i$  ein eindeutiger Index ist, der auf den Ereigniszähler von  $P_j$  zeigt. Dieser Index  $i$  wird auch als `Pnum` bezeichnet.

Jedes Gruppenmitglied besitzt also einen internen Zähler. Im Unterschied zur Lamport-Uhr wird eine Liste von Zählern mitgeführt. Somit merkt sich jedes Gruppenmitglied den Zählerstand aller anderen Gruppenmitglieder, ohne die Garantie zu haben, dass diese bereits synchronisiert sind.

Anfangs wird ein Null-Vektor für alle Gruppenmitglieder initialisiert. Ereignisse führen dazu, dass Änderungen am Vektor vorgenommen werden. Folgenden Ereignisse sind voneinander zu unterscheiden:

- Lokales-Ereignis:

Unter einem lokalen Ereignisse ist eine Berechnung oder eine sonstige Tätigkeit inbegriffen, die lokal von der Anwendung ausgeführt wird. Lokale Ereignisse werden in dieser Vorgabe nicht berücksichtigt und ignoriert.

- Sende-Ereignis:

Ein Teilnehmer der Gruppe entscheidet sich eine Nachricht zu verfassen und an die anderen Gruppenmitglieder als multicast zu senden. Hierfür inkrementiert der Prozess seinen prozess-internen zugehörigen Ereigniszähler und fügt die modifizierte Vektoruhr an die Nachricht bei.

- Empfang-Ereignis:

Wenn Teilnehmer der Gruppe eine Nachricht vom Multicast empfängt, wird die Nachricht solange zurückgehalten, bis die Nachricht aufgrund von kausalen Zusammenhängen ausgeliefert werden darf. Die Bedingung des kausalen Zusammenhangs ist, dass eine unmittelbare happens-before-Relation gelten muss. Unmittelbar heißt in diesem Fall, dass zwischen den Ereigniszähler der eigenen lokalen Vektoruhr ( $VT$ ) und der mit der Nachrichten eingegangenen Vektoruhr ( $vt$ ) an Position `Pnum` von  $VT$  eine Differenz von 1 herrscht. Es muss gelten  $VT[i] < vt[i]$ . Gleichzeitig dürfen die Ereigniszähler von  $VT$  an keiner anderen Position kleiner als die Ereigniszähler von  $vt$  sein, außer an der Stelle des eigenen Ereigniszähler (siehe erste Annahme). Es muss gelten  $VT_i[k] + 1 \geq vt[k]$ ; wobei  $k \neq i$ .

- Lese-Ereignis:

Wenn eine Nachricht an  $P_i$  zugestellt wird, dann findet eine Synchronisierung zwischen der lokalen Vektoruhr  $VT$  und dem in der Nachrichten empfangenen Vektoruhr  $vt$ . Hier wird elementweise das Maximum der Ereigniszähler genommen.

Besonderem Augenmerk wird dem Senden und dem Empfangen der Nachrichten geschenkt. Beim Senden der Nachricht wird der Zähler des Gruppenmitgliedes erhöht. Beim Empfangen einer Nachricht wird ein elementweises Maximum aus aktuellen und empfangenen Vektor gebildet, um den neuen Stand der Uhr zu ermitteln.



Da nun die Grundlagen des Algorithmus sowie die theoretischen Bezüge bekannt sind, werden im nächsten Kapitel die Anwendung mit Hilfe eines Entwurfes skizziert und beschrieben.

## 3 Entwurf

### 3.1 Vorgehensweise

Das zu implementierende System wird mit Hilfe der 4 Sichten nach Kruchten vorgestellt. Die Beschreibung des Systems erfolgt in einem Top-Down-Ansatz. Anfangs wird ein Überblick über die Struktur der Anwendung mittels der Bausteinsicht erworben. Hier werden auch die Verantwortlichkeiten der einzelnen Komponenten vorgestellt. Anschließend werden die Datenstrukturen beschrieben, die für die Zustellungslogik der Middleware relevant sind. Danach werden die Schnittstellen präsentiert, die die Grundlage der Kommunikation zwischen den Komponenten bilden. Abschließend werden die einzelnen Ereignisse als Szenario in Form von Sequenzdiagrammen vorgestellt und besondere interne Abläufe werden durch Flussdiagramme hervorgehoben. Die Anforderungen des Systems wurden von Prof. Herrn Klauck gestellt und online unter dem [Link](#) zur Verfügung gestellt.

### 3.2 Bausteinsicht: Komponenten

Das System besteht im Wesentlichen aus drei Komponenten: der Kommunikationseinheit, der Vektoruhr-ADT zusammen mit der Vektoruhrzentrale (towerClock) und dem Multicastsender (towerCBC).

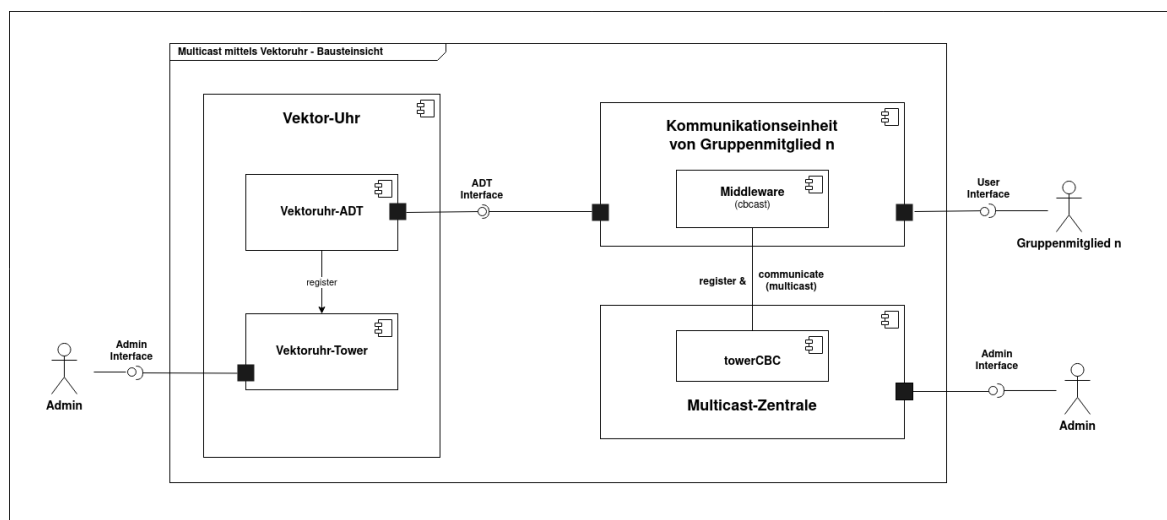


Abbildung 1: Bausteinsicht: Infrastruktur der Anwendung

**Kommunikationseinheit** Die Kommunikationseinheit ist eine Middleware und bietet seine Schnittstelle den Gruppenmitglieder an, um diesen den Dienst des kausalen Multicast anzubieten. Hierfür pflegt die Kommunikationseinheit eine Vektoruhr, die von der Vektoruhr-ADT zur Verfügung gestellt wird. Die Kommunikationseinheit verfügt über Lese- und Sende-Operationen. Die Funktionalitäten der Kommunikationseinheit werden in der Datei `cb-Cast.erl` implementiert und zur Verfügung gestellt.

Die Kommunikationseinheit meldet sich beim Multicast-Tower an, um eine eindeutige Gruppenteilnehmer ID zu erhalten. Für die Gewährleistung, dass die Nachrichten in kausaler Ordnung an der Anwender ausgeliefert werden, wird eine interne Holdbackqueue (HBQ) und eine interne Deliveryqueue (DLQ) genutzt. Die kausale Ordnung wird durch die Funktionen der Vektoruhr-ADT in der HBQ sichergestellt. Erfüllt eine Nachricht mit seinem jeweiligen logischen Zeitstempel die Anforderungen, die in Abschnitt 2 unter Empfangereignis beschrieben wurden, kann die Nachricht mitsamt Zeitstempel in die Deliveryqueue (DLQ) übertragen werden. An der Kommunikationseinheit geschehen also alle relevanten Ereignisse und bildet in dieser Anwendung die zentrale Komponente, um die Idee des kausalen Broadcast zu realisieren.

**Vektoruhr-ADT und Vektoruhr-Zentrale** Die Vektoruhr setzt sich aus zwei Komponenten zusammen. Einerseits der Vektoruhr-Zentrale und der abstrakten Datenstruktur der Vektoruhr.

Die Vektoruhr-ADT setzt die logischen Anforderungen der Vektoruhr um und bietet die notwendigen Funktionalität an, um a) den CBCAST-Algorithmus umzusetzen und b) die kausale Ordnung zu garantieren. Die Kommunikationseinheit greift auf die Vektoruhr-ADT zu, um den von ihr zur Verfügung gestellten Vektorzeitstempel zu verwalten und zu pflegen. Der Vektorzeitstempel stellt die Vektoruhr dar. Eine Vektoruhr besteht einerseits aus der eindeutigen ID, die auf den eigenen internen Ereigniszähler verweist und dem Vektor, der die Zeitstempel aller Prozess aufbewahrt.

Die Vektoruhr-Zentrale hat als Aufgabe den Gruppenmitgliedern bei Anfrage die eindeutige ID (`Pnum`) als positive ganze Zahl an die Vektoruhr auszustellen. Diese ID verweist dann als Index auf den Ereigniszähler, den die Kommunikationseinheit lokal zu pflegen hat. Die Vektoruhr-Zentrale ist bereits als kompiliertes Programm vorgegeben und wird als eine Service von der Vektoruhr-ADT bei Initialisierung verwendet.

**Multicast-Zentrale** Diese Einheit wird dazu verwendet die Nachrichten an die Gruppenmitglieder als Multicast-Nachricht zu streuen und bietet einen zuverlässigen ungeordneten Multicast. Zuverlässig heißt hier, dass Nachrichten nicht verloren gehen und dementsprechend ohne total Ausfälle stets ausgeliefert werden.

Die Multicast-Zentrale kann durch Angabe des Clients manuell oder automatisch betrieben werden. Die manuelle Konfiguration kann für Testzwecke genutzt werden, um die Nachrichten manuell an die Kommunikationseinheiten zu verteilen, und die Richtigkeit der kausalen Ordnung zu verfolgen. Die Kommunikatioknteilnehmer müssen sich bei der Multicast-Zentrale anmelden und registrieren. Die Multicast-Zentrale verwaltet dann eine interne Liste aller

bereits bekannten Kommunikatioknteilnehmer und speichert pro Teilnehmer eine eindeutige adressierbare Prozess-ID ab. Die Multicast-Zentrale kann sowohl blockierend, als auch nicht-blockierend arbeiten.

### 3.2.1 Datenstrukturen

Bevor nun die Schnittstellen definiert und vorgestellt werden, werden nun die Datenstrukturen vorgestellt, die von der Kommunikationseinheit verwendet werden.

Im folgenden werden 3 Datenstrukturen vorgestellt, die von der Kommunikationseinheit gepflegt und verwendet wird: die HBQ, die DLQ und die Vektoruhr (VT).

Die VektorUhr-ADT ist ein abstrakter Datentyp und wird nur durch die zur Verfügung gestellte Methoden (siehe Schnittstellendefinition) von der Kommunikationseinheit modifiziert. Die Datenstruktur wird als Tupel bestehend aus zwei Elementen definiert.  $VT := \{Pnum, VektorList := [ts_1, ts_2, \dots, ts_n]\}$  Das erste Element ist die eindeutige ProzessID Pnum des Vektorzeitstempels. Das zweite Element ist die VektorListe, die die Zeitstempel als ganze Zahlen abspeichert. Die ProzessID Pnum verweist darauf, welcher Zeitstempel der Kommunikationseinheit zugehörig ist. Bei der Initialisierung einer VT wird die Länge Pnum übernommen. Wenn die interne Vektoruhr mit anderen Vektoruhren synchronisiert werden soll, wird die VT entsprechend auf die Länge des größeren VTs erweitert. Grundsätzlich müssen bei der Operationen zwischen zwei VT die unterschiedlichen Längen berücksichtigt werden (siehe syncVT, compVT oder aftereqVTJ)(TODO Verweis).

Die HBQ garantiert die kausale Ordnung. Hier werden Nachrichten solange zurückgehalten, bis die kausale Ordnung es zulässt, die Nachricht an die DLQ weiterzuleiten, um diese bei Anfrage auszuliefern. Über das Kriterium, wann eine Nachricht auslieferbar ist, wurde bereits in Kapitel TODO thematisiert. Die HBQ ist einfache Liste bestehend aus einem zwei elementigen Tupel. An erster Position steht die Nachricht eines Gruppenmitgliedes. An zweiter Position steht die Vektoruhr.

$$HBQ := [e_1 := \{Message_{e_1}, vt_{e_1}\}, e_2, \dots, e_n]$$

Die DLQ stellt die Multicastreihenfolge dar. In der DLQ treffen die Nachrichten sortiert nach Vorgabe der HBQ ein. Die Struktur der DLQ orientiert sich an die HBQ.

$$DLQ := [e_1 := \{Message_{e_1}, vt_{e_1}\}, e_2, \dots, e_n]$$

In den folgenden Kapiteln werden die Abkürzungen verwendet.

### 3.2.2 Definition der Schnittstellen

In diesem Abschnitt werden die Schnittstellen der Komponenten vorgestellt, um festzulegen, wie Akteure und Komponenten miteinander kommunizieren können. Die Schnittstellen werden hier in einer Auflistung wiedergegeben mit Verweise auf Informationen, welche Parameter gegebenenfalls übergeben werden können und welche Rückgabe erwartet wird.

**Kommunikationseinheit** Die Kommunikationseinheit dient dazu, die Sende- und Lese-Ereignisse auszuführen und pflegt den lokalen Vektorzeitstempel. Grundsätzlich ist die Kommunikationseinheit an sich eine Schnittstelle zwischen einem Teilnehmer und den rechtlichen Teilnehmer der Gruppenkommunikation. Wie bereits angesprochen wird in der Kommunikationseinheit, die Zustellungslogik der Nachrichten festgehalten.

- **init()**: Initialisierung eines Prozesses für die Kommunikationseinheit. Die Multicast-Zentrale wird per Ping einbinden, um sich bei diesem zu registrieren. Zusätzlich wird mit der Schnittstelle der Vektoruhr-ADT eine Vektoruhr initialisiert. Die Information zum Anpingen werden aus einer Konfigurationsdatei - towerCBC.cfg - gelesen

Rückgabe: PID. Über die PID ist die Kommunikationseinheit ansprechbar

- **stop(Comm)**: Terminierung des Prozesses Comm. Prozess meldet sich lokal ab. Inhalte der HBQ und DLQ werden ausgegeben. VT wird gelöscht

Parameter: Comm ist PID. Rückgabe: done | null

- **send(Comm, Message)**: Die Kommunikationseinheit Comm sendet eine Nachricht Message an den Multicast-Tower.

Parameter: Comm ist PID und Message ist ein beliebiger String.

Rückgabe: done

- **read(Comm)**: Dem anfragenden Client wird eine Nachricht aus der DLQ nicht blockierend ausgeliefert. Befindet sich keine Nachricht in der DLQ, wird null zurückgegeben.

Parameter: Comm ist PID

Rückgabe: Message | null

- **receive(Comm)**: Dem anfragenden Client wird eine Nachricht aus der DLQ nicht blockierend ausgeliefert. Befindet sich keine Nachricht in der DLQ, wird solange gewartet, bis eine Nachricht eintrifft, die von der HBQ in die DLQ übertragen werden kann und somit an den Anfragenden Teilnehmer ausgeliefert wird. Rückgabe: Message

- **{<PID>,{castMessage,{<Message>,<VT>}}}**: Dies ist eine Empfangsnachricht, die der Multicast an alle Gruppenmitglieder sendet. Die PID ist der ursprüngliche Absender der Nachricht. Die Nachricht wird <Message>,<VT> in die HBQ eingefügt. Die Empfangsnachricht beinhaltet die Information castMessage und die Nachricht Message von PID, sowie den Vektorzeitstempel VT von PID

Parameter: PID ist der Absender, der das Messageformat <Message>,<VT> an Multicast-Zentrale gesendet hat

Rückgabe: Message | null

#### **selbsthinzugefügte Schnittstellen:**

**send(Comm, Message, Blocking)**

- **send(Comm, Message, Blocking)**: Über Blocking kann festgelegt werden, ob Nachrichten blockierend (true) oder nicht-blockierend gesendet werden.

### **selbsthinzugefügte Nachrichtenformate:**

1.  $\{send, Message, Blocking\}$  2.  $\{PID, read, NotBlocking\}$  und 3.  $\{PID, stop\}$

Da der Client die Kommunikationseinheit über die PID erreicht, müssen die entsprechenden Informationen an den Prozess mittels dieser Nachrichtenformate übertragen werden, damit dieser unterscheiden kann, welche interne Abläufe er zu leisten hat.

**Vektoruhr-ADT** Die Vektoruhr bietet seine Schnittstellen der Kommunikationseinheit an, um die Vektoruhr zu modifizieren oder Vektoruhren miteinander zu vergleiche, um Rückschlüsse auf die Kausalität zu erhalten. Die Begriffe Vektoruhr und Vektorzeitstempel, sowie Zeitstempel und Ereigniszähler werden hier synonym verwendet. Darüber hinaus steht die Abkürzung VT für den abstrakten Datentyp der Vektoruhr.

- **init()**: Initialisierung eines initialen Vektorzeitstempel. Erfragt beim **towerClock** eine eindeutige Identität **Pnum** ab. Die notwendigen Informationen sind aus der Konfigurationsdatei [towerClock.cfg] zu lesen. Die Länge der Vektoruhr beträgt **Pnum**.

Rückgabe: initialer Vektorzeitstempel im Format : **Pnum, VektorList**

- **myVTid(VT)**: Rückgabe der ProzessID **Pnum** aus dem Vektorzeitstempel VT als ganze Zahl  
Parameter: VT ist Vektoruhr

Rückgabe: **Pnum**

- **myVTvc(VT)**: Rückgabe des Vektors mit den Zeitstempel als Liste aus ganzen Zahlen inklusive 0

Rückgabe: **VektorList**

- **myCount(VT)**: Rückgabe des eigenen Zeitstempel als ganze Zahl

Rückgabe: eigener Ereigniszähler  $e_{pnum}$

- **foCount(J, VT)**: Rückgabe des Zeitstempel an Position J aus **VektorList** als ganze Zahl. Die Nummerierung der Indizes beginnt mit 1  
Rückgabe: Ereigniszähler  $e_j$

- **isVT(VT)**: Überprüfung, ob VT ein Vektorzeitstempel ist

Rückgabe: true | false

- **syncVT(VT1, VT2)**: Synchronisierung der **VektorList** von VT1 und VT2. Es wird das elementweise Maximum aus beiden **VektorList** gebildet.

Parameter: VT1 ist eigene Vektoruhr, VT2 ist fremde Vektoruhr aus der DLQ

Rückgabe: modifizierter Vektorzeitstempel

- **tickVT(VT)**: Inkrementierung des eigenen Ereigniszählers. An Position **Pnum** der **VektorList** wird der Zeitstempel um 1 inkrementiert

Rückgabe: modifizierter Vektorzeitstempel

- **compVT(VT1, VT2)**: Vergleich zweier Vektorzeitstempel. Diese Vergleichsoperation kann zur Sortierung der HBQ verwendet werden. Parameter: VT1 ist der Vektorzeitstempel der in der Kommunikationseinheit neu eingetroffene Nachricht. VT2 sind die Vektorzeitstempel aus der HBQ  
Rückgabe: afterVT, beforeVT, equalVT oder concurrent VT.

- **aftereqVTJJ(VT, VTR)**: Vergleich zweier Vektorzeitstempel ohne dabei den Index J von VT zu beachten, wobei J der Index **Pnum** ist. Mit Hilfe dieser Operation wird festgestellt,

ob eine Nachricht von der HBQ in die DLQ überführbar ist. Zusätzlich wird eine Distanz zwischen VT[j1] und VTR[j2] berechnet, wobei j1 und j2 die ProzessID Pnum von VTR ist. Nur bei einer Distanz von -1 ist die Nachricht auslieferbar. **Parameter:** VT ist eigener Vektoruhr und VTR ist ein Element aus der HBQ

Rückgabe: aftereqVTJ, <Distanz an der Stelle J> | false

**Vektoruhr-Zentrale** - **init()**: Start der Vektoruhr-Zentrale + Einlesen notwendiger Informationen aus towerClock.cfg

Rückgabe: PID

- **stop(<PID>)**: Terminieren der Vektoruhr-Zentrale

**Parameter:** PID der Vektoruhr-Zentrale.

Rückgabe: true | false

- **{getVecID,<PID>}**: Mithilfe dieses Nachrichtenformats wird eine eindeutige ID an PID für eine Vektoruhr angefragt. **Parameter:** PID als Rückgabeadresse.

Rückgabe: an PID: {vt,<Prozess-ID>}, wobei <Prozess-ID> die eindeutige ID der Vektoruhr ist

**Multicast-Zentrale** - **init()** | **init(auto|manu)**: Start einer manuellen oder automatischen Multicast-Zentrale.

Rückgabe: PID

- **stop(PID)**: Beendet Multicast-Zentrale.

**Parameter:** PID Kontaktadresse der Multicast-Zentrale

Rückgabe: true | false

- **reset(PID)**: Zurücksetzen der Multicast-Zentrale in initialen Zustand.

Rückgabe: true | false

- **listtall()**: Auflistung aller bisher bekannten bzw. registrierten Kommunikationseinheiten in die Log-Datei.

Rückgabe: true | false

- **cbcast(<Receiver>,<MessageNumber>** : Zustellen der Nachricht MessageNumber an Receiver. Diese Schnittstelle kann bei Start einer manuellen Multicast-Zentrale verwendet werden.

**Parameter:** MessageNumber ist ein Index und verweist auf die MessageNumber-te eingetroffene Nachricht bei der Multicast-Zentrale. Receiver ist die Gruppenid eines an der Gruppenkommunikation beteiligten Teilnehmer. Rückgabe: true | false

- **{<PID>,{register,<RPID>}}**: Mithilfe dieses Nachrichtenformats wird die Kommunikationseinheit RPID bei der Multicast-Zentrale registriert. **Parameter:** PID erhält die Rückgabeadresse. An RPID werden die Multicast-Nachrichten gesendet.

Rückgabe: an PID: {replycbc, ok\_registered | ok\_existing}

- {<PID>,{multicastB,{<Message>,<VT>}}}: Mithilfe dieses Nachrichtenformats wird an allen Kommunikationseinheiten die Nachricht <Message>,<VT> als ungeordneter, blockierender Multicast gesendet. Blockierend heißt, dass die Anfragen sequentiell abgearbeitet werden.  
Parameter: PID wird für Log-Zwecke verwendet.

- {<PID>,{multicastNB,{<Message>,<VT>}}}: Mithilfe dieses Nachrichtenformats wird an allen Kommunikationseinheiten die Nachricht <Message>,<VT> als ungeordneter, nicht-blockierender Multicast gesendet. Nicht-Blockierend heißt, dass die Anfragen parallel abgearbeitet werden.  
Parameter: PID wird für Log-Zwecke verwendet.

### 3.3 Laufzeitschicht

In diesem Abschnitt werden die interne Abläufe der Kommunikationseinheit und der Vektoruhr-ADT vorgestellt. Bezüglich der Kommunikationseinheit werden auch Sequenzdiagramme eingeführt, um die Kommunikation mit der Multicast-Zentrale und der Vektoruhr-ADT vorzustellen.

#### 3.3.1 Kommunikationseinheit

Die Kommunikationseinheit wurde als Middleware definiert, die einen Dienst an die Anwendung anbietet. Diese Dienst sind mit den Ereignissen Senden und Empfangen von Nachrichten verknüpft. In Abbildung 2 wird die Kommunikation unter den Gruppenteilnehmer und den auftretenden möglichen Ereignissen dargestellt.

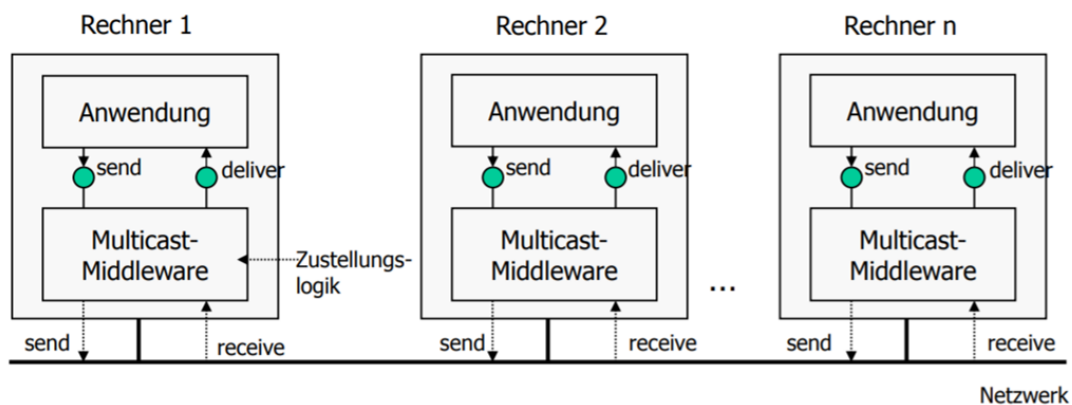


Abbildung 2: Middleware-Architektur der Kommunikationseinheit  
[Vorlesungsfolien Peter Mandl Seite 30](#)

Beim Empfangen von Nachrichten wird unterschieden, ob die Kommunikationseinheit (receive) oder der Leser (deliver) die Nachricht empfängt. Denn die Aktion "Nachrichtempfang" beider Beteiligten lösen unterschiedliche interne Abläufe in der Kommunikationseinheit aus, die auch im Folgenden besprochen werden. Beginnend wird die Vorbereitungs- und Initialisierungsphase der Kommunikationseinheit vorgestellt.



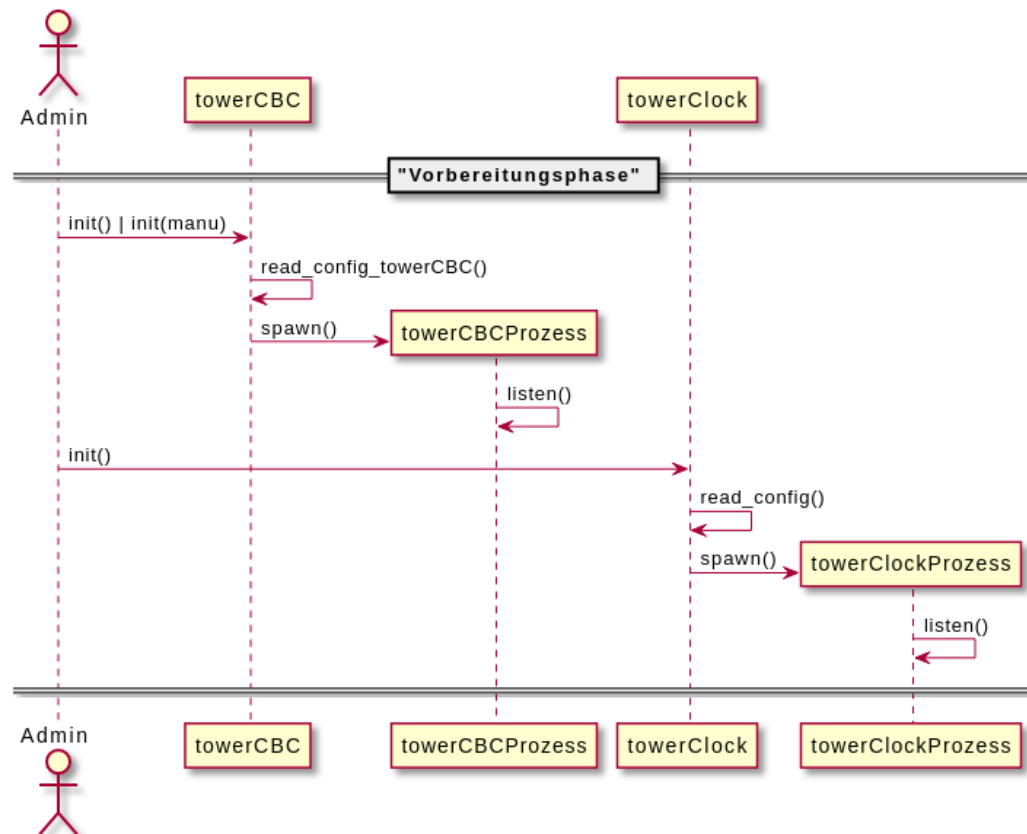


Abbildung 3: Laufzeitschicht: Vorbereitungsphase

**Vorbereitungs- und Initialisierungsphase** In der Vorbereitungsphase (Abbildung 3) werden die Multicast-Zentrale (towerCBC) und Vektoruhr-Zentrale (towerClock) auf je einer Node als Prozess mit `init()` gestartet. Nun kann auf einer anderen beliebigen Node bzw. Nodes beliebig viele Kommunikationseinheiten gestartet werden (Abbildung 4), sodass sich Gruppenteilnehmerbeliebig viele Nachrichten zu senden können. Die Kommunikationseinheit wird durch einen beliebigen Client gestartet. Die Kommunikationseinheit ist somit dann die Schnittstelle, um mit anderen Clients in Kommunikation zu treten. Der Client bekommt beim Starten einer Kommunikationseinheit die Prozess-ID (PID) zurück. Mit Hilfe der Prozess-ID kann der Client den Service (`send`, `read`, `received`, `stop`) der Kommunikationseinheit ansprechen. Innerhalb der Initialisierungsphase registriert sich der neu gestartete Prozess bei der Multicast-Zentrale. Die Informationen zum Kontaktieren der Multicast-Zentrale werden aus der `towerCBC.cfg` - Konfigurationsdatei ausgelesen.

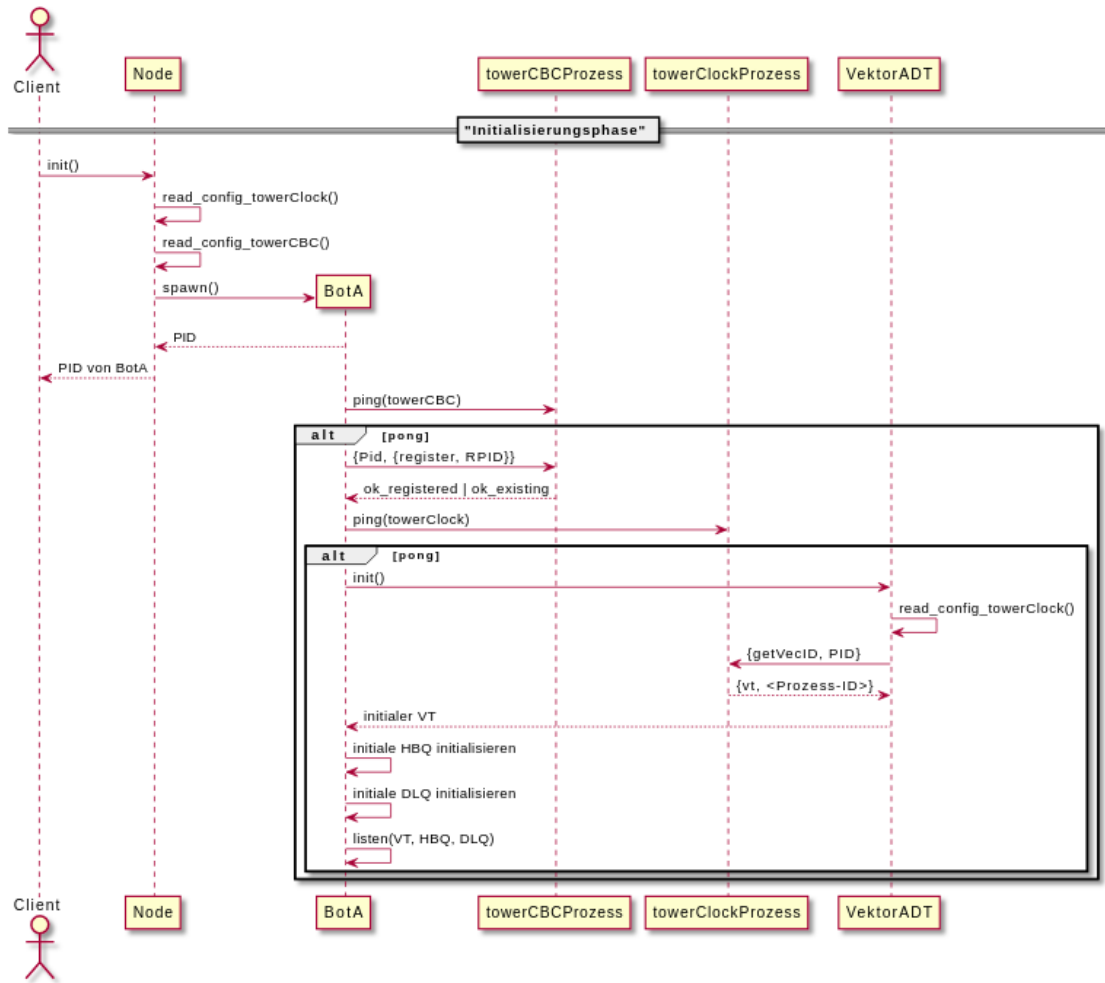


Abbildung 4: Laufzeitschicht: Initialisierungsphase

Des Weiteren beauftragt der Prozess die Vektoruhr-ADT eine Vektoruhr zu initialisieren. Hierfür fragt die Vektoruhr-ADT die Vektoruhr-Zentrale nach einer positiv eindeutigen Indexnummer (Pnum) für die Vektoruhr. Die Informationen zum Kontaktieren der Vektoruhr-Zentrale werden aus der **towerClock.cfg** - Konfigurationsdatei ausgelesen. Diese Indexnummer, sowie eine initialisierte Vektoruhr der Länge Indexnummer, wird an den beauftragten Prozess zurückgegeben. Der Prozess begibt sich dann mit der initialen Vektoruhr, einer leeren HBQ und einer leeren DLQ in den listen Modus.

Das in dem Sequenzdiagramm dargestellte Szenario repräsentiert hierbei einen fehlerfreien Ablauf. Sollte es zu einem Fehlerzustand kommen, beispielsweise ist die Multicast-Zentrale oder die Vektoruhr-Zentrale nicht erreichbar, dann wird dies mit einem Log-Eintrag vermerkt und der Client informiert.

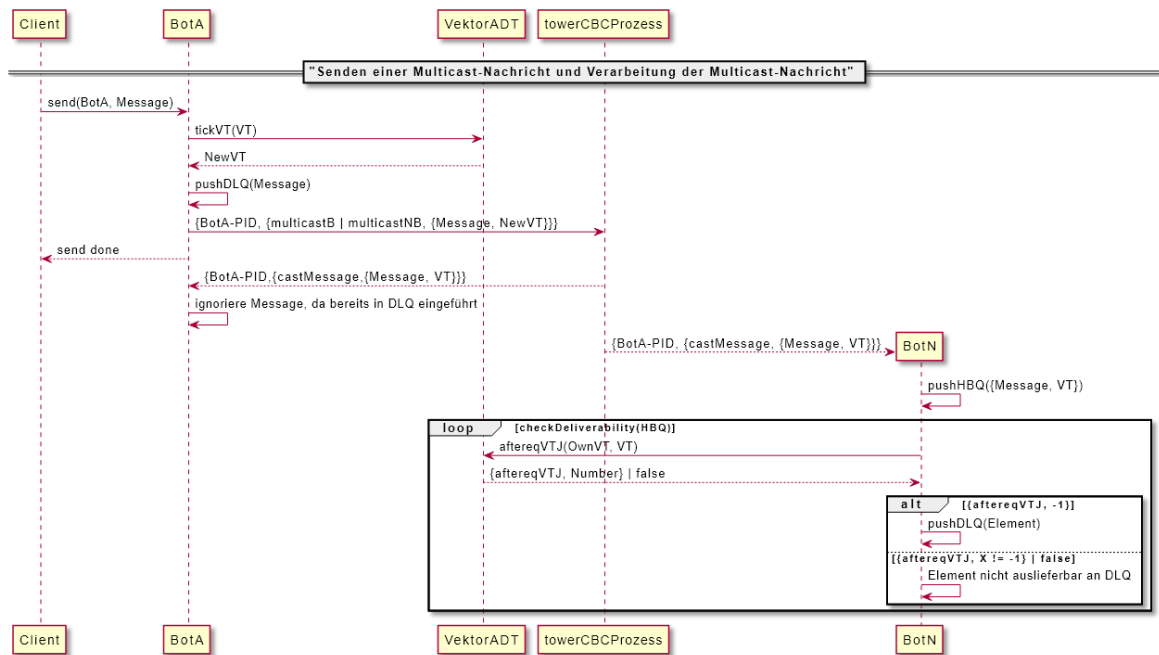


Abbildung 5: Kommunikation beim Sendeereignis

**Sendeereignis und Empfangereignis an der Kommunikationseinheit** Ein Client kann mit Angabe der Prozess ID seine Kommunikationseinheit mit `send(PID, Message)` ansprechen, um eine Nachricht als Multicast zu versenden (Abbildung 5). Beim Eintreffen eines solchen Befehls wird die lokale Vektoruhr mit der Funktion `tickVT(VT)` an der Stelle `Pnum` inkrementiert. Diese modifizierte VT wird dann als neue lokale Vektoruhr übernommen. Anschließend wird die Nachricht mitsamt der modifizierten VT in die DLQ übertragen und an die Schnittstelle der Multicast-Zentrale mit dem Nachrichtenpattern

`{BotA-PID, {multicastB | multicastNB, {Message, NewVT}}}`

versendet. Die Multicast-Zentrale übernimmt anschließend die Aufgabe die einkommenden Nachrichten an alle Kommunikationsteilnehmer zu versenden.

Über die selbsterweiterte Schnittstelle `send(PID, MSG, Blocking)` kann kontrolliert werden, ob die Nachrichten blockierend oder nicht-blockierend als Multicast gesendet werden soll. Blockierend bedeutet in diesem Fall, dass die Anfragen sequentiell abgearbeitet werden.

Nach einer Zeitdauer X treffen die Nachrichten in einem ungeordnet Multicast ein.

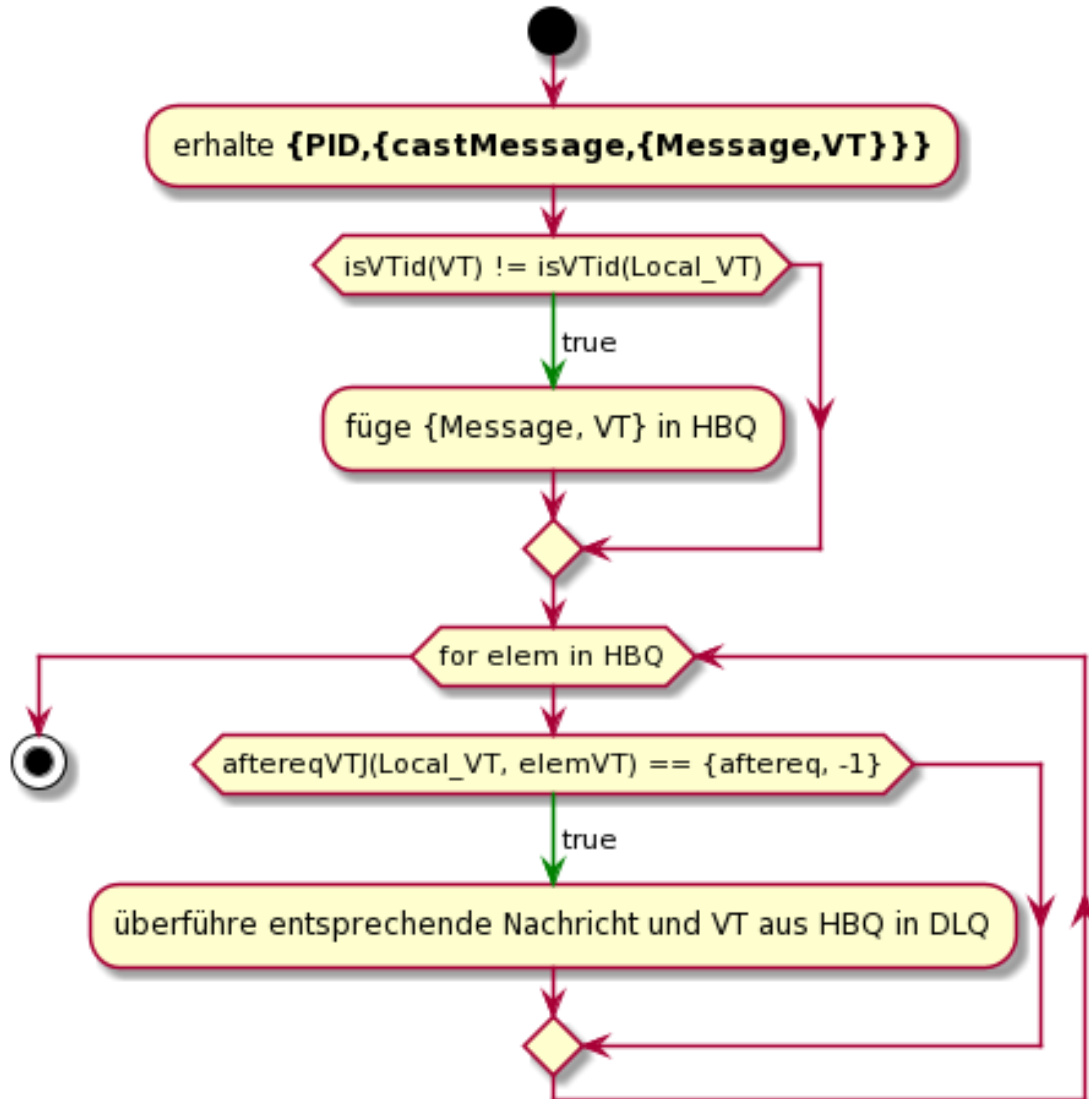


Abbildung 6: interne Verarbeitung beim Erhalt einer Multicast-Nachricht

Die Nachricht wird in die HBQ überführt, wenn sich die **Pnum** der an der Nachricht beigefügten Vektoruhr von der **Pnum** der lokalen Vektoruhr unterscheidet (Abbildung 6). Anschließend wird überprüft, ob sich Nachrichten in der HBQ befinden, die in die DLQ überführbar sind. Ausschlagkriterium hierfür ist, die im Algorithmus definierte Bedingung: elementweise werden die Vektoruhren aus der HBQ mit der lokalen Vektoruhr verglichen. Die Vektoruhr aus der HBQ muss größer gleich als die lokale Vektoruhr sein, außer an Position **J**, wobei **J** die **Pnum** der Vektoruhr aus der HBQ ist. Zusätzlich muss die Differenz der Ereigniszähler beider Vektoruhren an Position **J** -1 betragen. Diese Bedingungen resultieren aus der in Abschnitt 2 vorgestellten Auslieferungsbedingungen und definieren die Zustellungslogik der Nachrichten nach kausaler Ordnung. Mit Hilfe der HBQ und der DLQ sowie den Funktionalitäten der Vektoruhr-ADT ( $\text{aftereqVTJ}(\text{Local-VT}, \text{ElemHBQ-VT})$ ,  $\text{compVT}(\text{VT1}, \text{VT2})$ ) kann die kausale Ordnung garantiert werden. Die Überprüfung der Zustellung findet sowohl

beim Empfangen von Nachrichten statt, als auch bei der Anfrage des Lesers eine Nachricht ausgeliefert zu bekommen. Dies wird in dem nächsten Abschnitt besprochen.

Nachdem die DLQ und HBQ gegebenenfalls modifiziert wurden, geht der Prozessablauf wieder zurück in den listen-Modus.

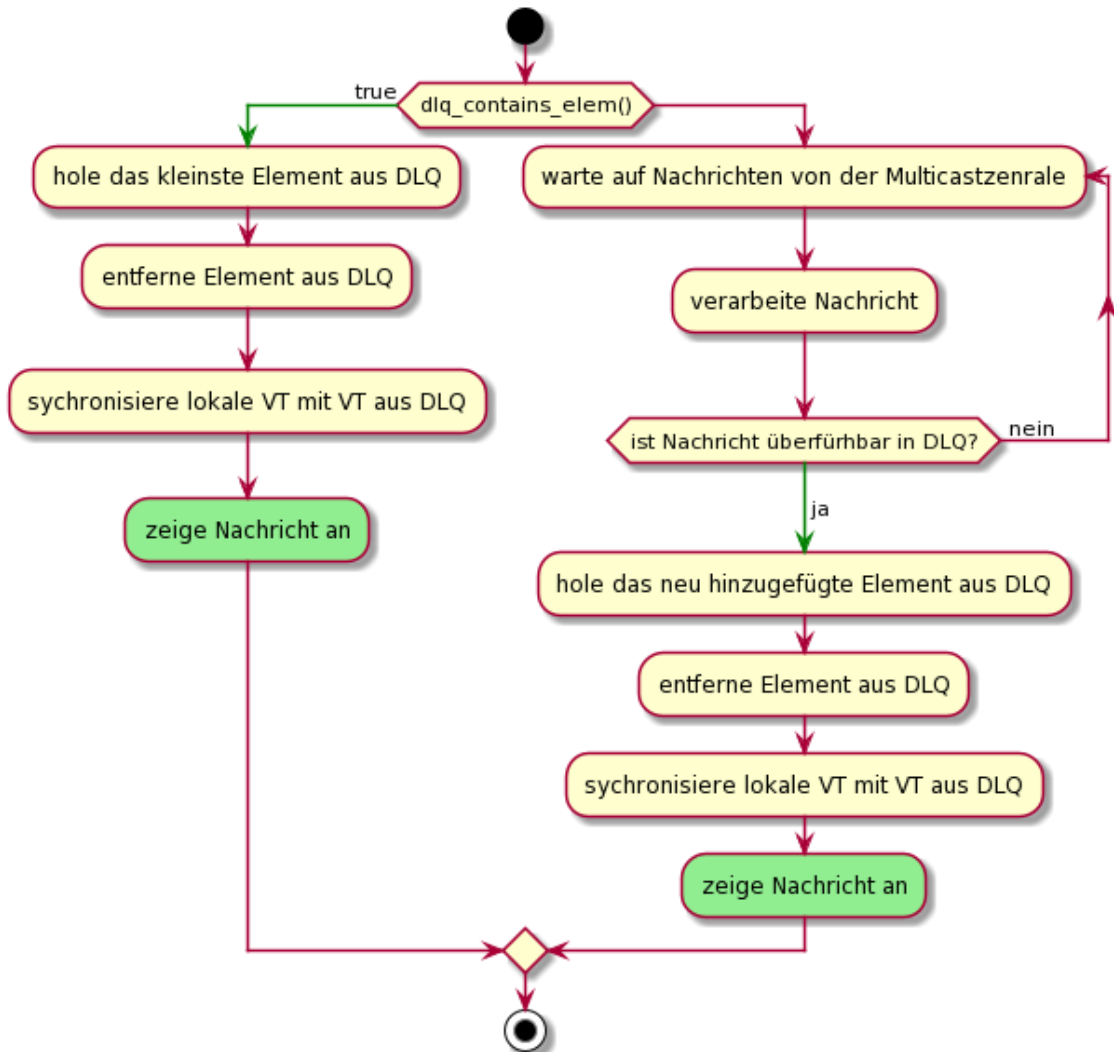


Abbildung 7: Blockierende Auslieferung einer Nachricht an anfragenden Client (read)

**Leseabfrage eines Clients** Die Kommunikationseinheit bietet für seinen Client zwei unterschiedliche Lese-Operationen bereit: `read(PID)` und `received(PID)` (Abbildung 7). Grundsätzlich wird bei beiden Operationen als erstes geschaut, ob Nachrichten in der HBQ vorhanden sind, die an die DLQ überführbar sind. Bei erfolgreicher Bedingung wird die Nachricht zurückgegeben. `read()` ist nicht blockierend und liefert dem Leser bei Anfrage auch dann eine Antwort, wenn keine Element in der DLQ vorhanden sind. Demnach wäre die Abzweigung nach rechts in Abbildung 7 eine einfache Rückgabe von `null` an den anfragenden Client.

`received` hingegen arbeitet blockierend und wartet solange auf eine Nachricht von der Multicast-Zentrale, die an die DLQ ausgeliefert werden kann. So kann die an die DLQ neu gelieferte Nachricht an den blockierend-anfragenden Client ausgeliefert werden.

Wenn Nachrichten an den Client ausgeliefert werden, muss die lokale Uhr synchronisiert werden. Dafür vergleicht man die lokale Vektoruhr mit der aus der Nachricht versehenen Vektoruhr und bildet das elementweise Maximum aus beider Vektoruhren.

Anschließend geht der Prozessablauf wieder zurück in den listen-Modus.

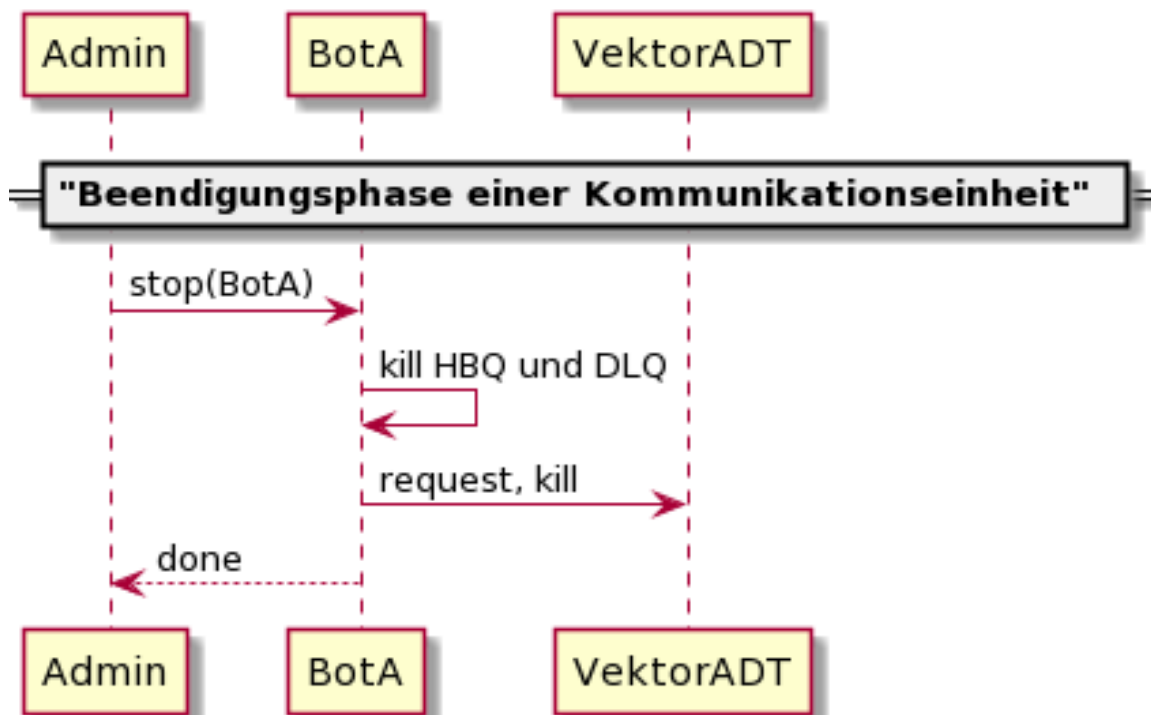


Abbildung 8: Terminierung einer Kommunikationseinheit

**Terminierung** Die Terminierung der Kommunikationseinheit-Middleware ist sehr trivial und besteht aus einer einfachen Ausgabe der aktuellen Informationen und der Rückgabe `done`. Letztlich wird der listen-Modus der Kommunikationseinheit verlassen, so dass sich der Prozess terminiert und nicht mehr erreichbar ist. Sollte ein Prozess nicht erreichbar sein, um diesen zu terminieren, wird der Client mit Nachricht `null` informiert.

### 3.3.2 Methoden der Vektoruhr-ADT

In diesem Abschnitt werden die Funktionalitäten der Vektoruhr-ADT detaillierter beschrieben. Triviale Methoden werden nur kurz beschrieben.

`myVTid(VT)`, `myVTvc(VT)` und `myCount(VT)` sind einfache Getter-Methoden, die über die jeweils angefragte Information Auskunft geben (siehe Schnittstellendefinition). Die Funktion `foCount(J, VT)`, wobei J eine positive ganze Zahl größer 0 ist, gibt den Ereigniszähler aus

VC an Stelle J zurück. Im Falle, dass  $J > \text{length}(\text{VT})$  ist, wird 0 zurückgegeben. Die Methode `tickVT(VT)` inkrementiert den Wert des internen Ereigniszähler um 1.

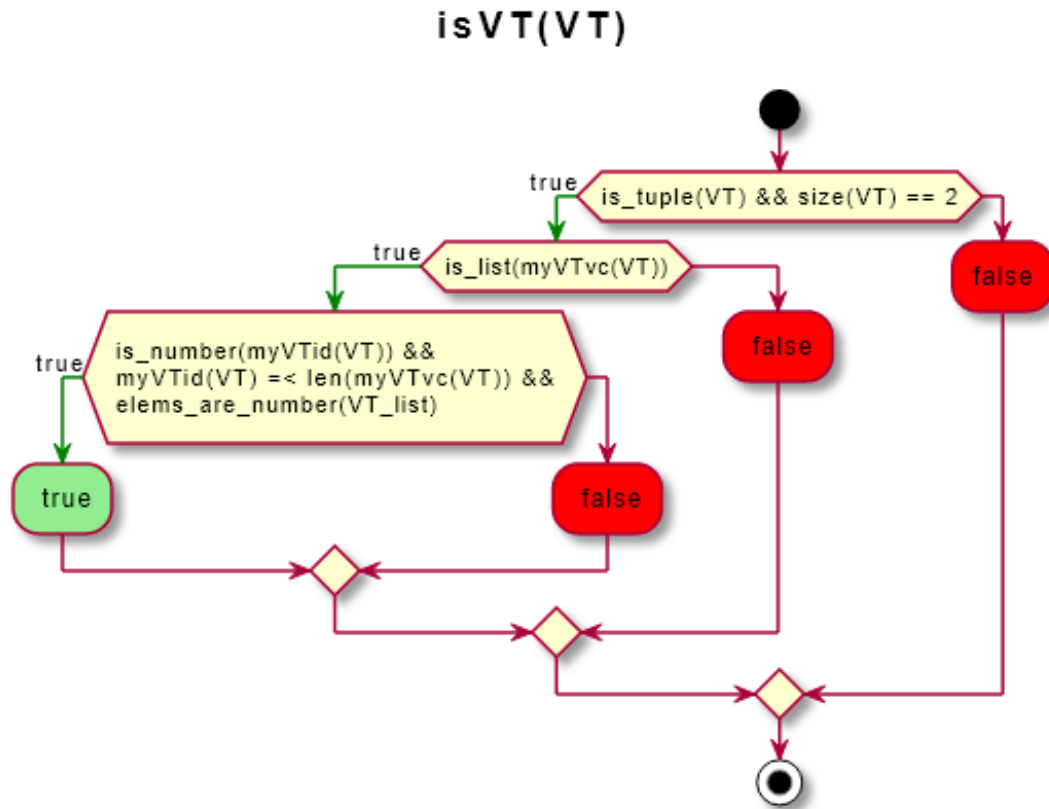


Abbildung 9: `isVT(VT)` - Überprüfung der korrekten Datenstruktur

Mit der Methode `isVT(VT)` kann überprüft werden, ob die definierte Datenstruktur eingehalten ist (siehe Abbildung 9) und wird als Hilfsmethode verwendet.

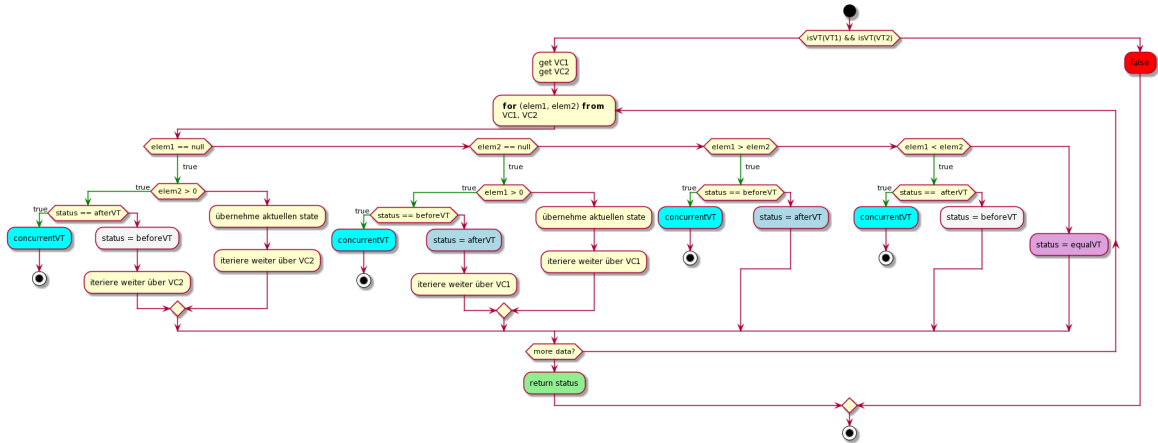


Abbildung 10: compVT(VT1, VT2) - Vergleichsmethode für Vektoruhren

Die Methode `compVT(VT1, VT2)` vergleicht die Vektoruhren miteinander. Vektoruhren sind `equalVT`, wenn an allen Position die Ereigniszähler identisch sind. Vektoruhren sind `afterVT`, wenn an mindestens einer Stelle der Ereigniszähler von VT1 größer als der Ereigniszähler von VT2 ist und allen anderen Stellen `equalVT`. Für `beforeVT` gilt genau der umgekehrte Vergleich mit kleiner. `ConcurrentVT` tritt dann auf, wenn beim Vergleich sowohl `beforeVT` als auch `afterVT` aufgetreten ist. Diese Bedingungen sind auch in Abbildung 10 zu sehen.

Beim Vergleich der Vektoruhren kann es passieren, dass sich die Vektoren in ihrer Länge unterscheiden können und müssen deshalb eingangs berücksichtigt werden. Die Methode `compVT()` wird genutzt, um die HBQ zu sortieren. Mit Hilfe des Vergleichs kann die HBQ aufsteigend oder absteigend sortiert werden. Bei aufsteigender Sortierung stehen die Elemente an erster Stelle logisch gesehen unmittelbar näher dem lokalen Zeitstempel, als die Elemente die sich weiter hinten in der HBQ befinden. Bei Überprüfung der Auslieferung von Nachrichten muss deshalb nicht die gesamte HBQ untersucht werden, sondern nur das kleinste Elemente der HBQ.

Der Vergleich der Vektoruhren wird in eine interne Methode `compVTlist(VC1, VC2)` ausgelagert, die dann auch von der Methode `afterreqVTJ` wiederverwendet wird. VC steht für die Liste der Zeitstempel.

Die Methode `afterreqVTJ(VT, VTR)` (Abbildung 11) überprüft, ob das Element VTR aus der HBQ verglichen mit VT unmittelbar in kausaler Beziehung steht. Hierfür werden aus den beiden zu vergleichenden Vektoruhren jeweils die Vektorzeitstempel entnommen, ohne dabei die Position J zu betrachten, wobei J die Pnum von VTR ist. Diese Listen werden mit der internen Methode `compVTlist` verglichen. Wenn die Vektorzeitstempel von VT größer gleich der Vektorzeitstempel von VTR ist, wird noch die Distanz der Ereigniszähler an Position J berechnet ( $VC[j] - VCR[j]$ ).



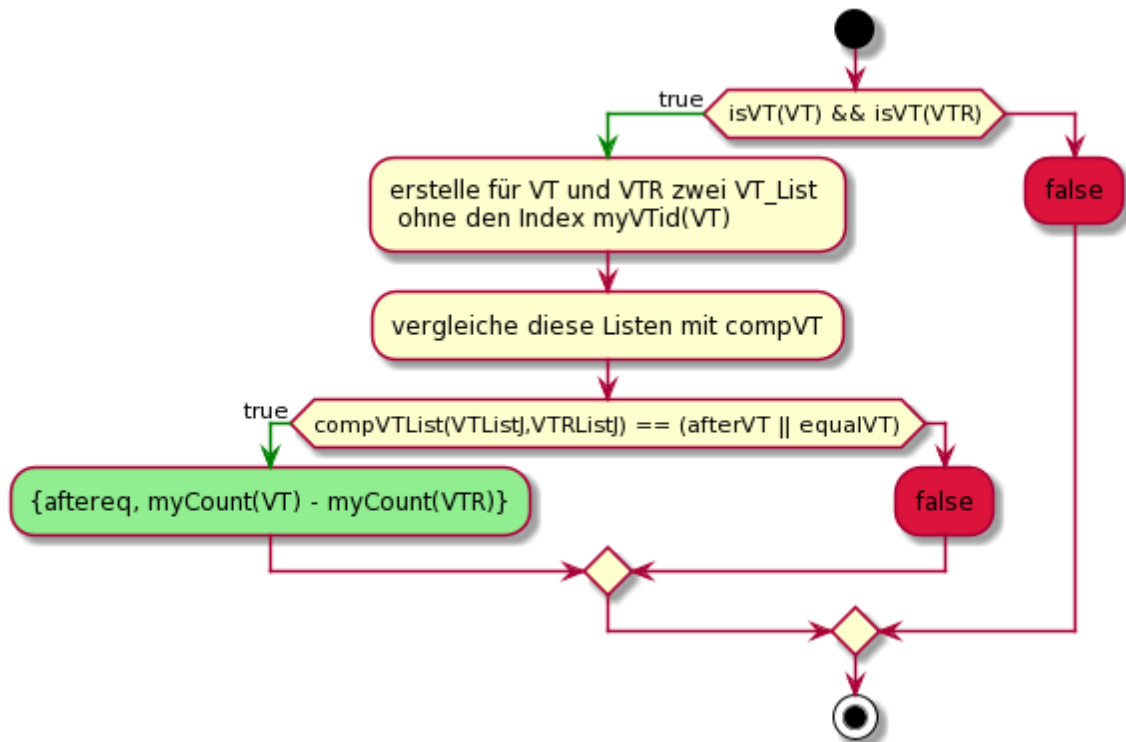


Abbildung 11: aftereqVTJ(VT, VTR) - Überprüfung, ob VTR auslieferbar ist an die DLQ

### 3.3.3 Zusammenfassung

Es wurden nun alle notwendigen Methoden und Abläufe vorgestellt, die intern ablaufen müssen, um einen kausalen Broadcast zu ermöglichen. Im nächsten Abschnitt wird vorgestellt, wie diese Anforderungen technisch umgesetzt sind.

## 3.4 Reflexion der Vorgehensweise

Die Anwendung wurde in Erlang/OTP realisiert. Die zu implementierenden Einheiten waren die Vektoruhr-ADT (vectorC.erl) und die Kommunikationseinheit (cbCast.erl). Anfangs wurde die Vektoruhr-ADT realisiert, da sie im größten Teil aus einfachen trivialen Methoden bestand und die Methoden compVT und aftereqVTJ ausschlaggebend sind, um den kausalen Multicast zu realisieren. Die korrekte Funktionsweise der einzelnen Methoden wurden mit Hilfe von Tests (vectorCTest.erl) abgedeckt und anschließend mit dem Test-System von Herrn Klauck getestet.

Anschließend wurde die Kommunikationseinheit realisiert. Die einzelnen Phasen, wie Senden und Empfangen wurden in Form von Sequenzdiagrammen herausgearbeitet, um festzuhalten, wie die Kommunikation zwischen den Akteuren aussieht. Darauf aufbauend wurden die interne Abläufe skizziert. Da der Anwender die Middleware über seine PID erreicht, wurde eine listen\_loop() definiert, in der die entsprechenden Ereignisse (send, read, stop) verarbeitet werden. Die listen\_loop() hat dabei ständig Zugriff auf die aktuelle Vektoruhr, auf

die aktuelle Holdbackqueue und auf die aktuelle DeliveryQueue. Die `listen_loop()` beinhaltet einen receive Block, der unterschiedliche Nachrichtenformate erwarten kann (siehe erweiterte Nachrichtenformate unter Kommunikationseinheit). Es wurde schrittweise die Logik für die jeweiligen Nachrichtenformate implementiert. Abschließend wurde die Anwendung mit dem Test-System von Herrn Klauck getestet und versucht Szenarien zu konstruieren, die einen ungeordneten ungünstigen Multicast realisieren.

Nun werden Probleme und Designentscheidungen während der technischen Umsetzung vorgestellt, die besonders hervorzuheben sind.

**Probleme bei der Auslieferung von Nachrichten** Anfangs war ich davon ausgegangen, dass die Auslieferbarkeit von Nachrichten anhand des größten Elementes in DLQ bestimmt wird. Diese Annahme entstand unter anderem dadurch, dass bereits im Praktikum 1 die Auslieferbarkeit von Nachrichten durch die DLQ festgelegt wurde und habe die jeweilige Kontexte nicht miteinander berücksichtigt, dass der maßgebende Faktor diesmal die Vektorkuhr ist. Demnach wurden die Elemente der HBQ mit der lokalen Vektorkuhr verglichen, um zu schauen, ob die definierte Auslieferungsbedingung erfüllt ist. Diese Erkenntnis wurde erst gewonnen, als ich mich mit dem vorgegebenen Testsystem beschäftigt habe und versucht habe das Verhalten des Systems zu verstehen und zu analysieren.

Ein weiteres Missverständnis gab es über die Methode `aftereqVTJ(VT, VTR)`. Es hatte sich zwar das Verständnis aufgebaut, dass die Auslieferungsbedingung zweischrittig untersucht wird, aber ich habe die Definition von J falsch verstanden. Im ersten Schritt wird überprüft, ob  $VT \geq VTR$  ist. Hierbei wird die Stelle J nicht berücksichtigt. Und im zweiten Schritt wird gegebenenfalls die Distanz zwischen der Ereigniszähler an Position J berechnet. Ich habe in Schritt 1 angenommen, dass J die Pnum von VT ist und im zweiten Schritt die Pnum von VTR. Nach gründlichem Lesen hat sich dieses Problem mit der Zeit gelöst.

**Unterschiedliche Länge der Vektorzeitstempel** Da die Vektorzeitstempel mit unterschiedlichen Längen initialisiert werden, müssen beim Vergleich von Vektorzeitstempel die Längen berücksichtigt werden. Eine einfachere Strategie dieses Problem zu umgehen, ist es, die kleinere VT mit Nullen aufzufüllen. Da Erlang eine funktionale Sprache und Pattern Matching-Prinzip besitzt, habe ich mich dazu entschieden, die Ausnahmen über die Funktionsköpfe zu berücksichtigen. Dies ist mir auch grundsätzlich gut gelungen, jedoch habe ich den Fehler gemacht, dass ich die Rekursion zu früh abgebrochen habe, wenn das Ende der kleinere Liste erreicht wurde. Durch die Korrektur die Rekursion erst dann zu beenden, wenn das letzte Element der größere List gelesen wurde, haben sich auch die auftretenden Probleme gelöst.

**Blockierendes Ausliefern der Nachrichten** Der blockierende Zugriff eines Clients auf die Kommunikationseinheit wird durch ein receive-Block auf Seiten des Clients realisiert. Dieser receive-Block wird nur unter Bedingung aufgelöst, wenn eine Nachricht an den Client auslieferbar ist. Die Kommunikationseinheit ist nicht blockiert und kann Nachrichten von anderen Teilnehmern erwarten. An dieser Stelle habe ich mich dazu entschieden, alle anderen Dienste der Kommunikationseinheit inaktiv zu stellen. Sollte die Aufforderung erscheinen, eine Nachricht zu senden oder eine weitere Nachricht zu lesen, wird eine Benachrichtigung an den Anfragenden gesendet, dass zurzeit ein Client bedient wird. Denn ich gehe davon

auch aus, dass nur ein Client die Middleware bedient und demnach schien es mir sinnvoll, die anderen Dienste inaktiv zu stellen. Selbstverständlich könne man auch die Aufforderung ein Sendeereignis oder ein Leseereignisse zu starten, einbauen, und würde im Allgemeinen keine große Herausforderungen sein, da man nur die entsprechende Nachrichtenformate in den receive-Block setzt und die entsprechenden Funktionen aufruft.

**Synchronisierung der Vektoruhren** Anfangs war mir noch nicht klar, wann die lokale Vektoruhr synchronisiert wird und bin davon ausgegangen, dass die lokale Vektoruhr bei der Überlieferung einer Nachricht an die DLQ mit dem Vektor-Zeitstempel der Nachricht synchronisiert wird. Auch wenn diese Annahme sich falsch herausgestellt hat, habe ich mich versucht damit auseinanderzusetzen, welche Konsequenzen es mit sich tragen könnte, wenn die lokale Vektoruhr bei der Überführung von Elementen in die DLQ transportiert wird. Diese Annahme hatte sich unter anderem dadurch gebildet, dass ich mir erhofft hatte, dass die Nachrichten schneller in der DLQ landen. Denn stelle man sich folgendes Szenario vor. Ein Prozess hat nun mehrere Ereignisse empfangen und die in der DLQ befindlichen Elemente stehen verglichen mit den Elementen aus der HBQ unmittelbar in kausaler Relation. Dann wäre es ja eigentlich möglich diese auch in die DLQ zu übertragen, wenn die lokale Vektoruhr schon beim vorherigen Transport der Elemente in die DLQ synchronisiert wäre. Bisher wurde diese Annahme noch nicht umgesetzt, und überprüft, ob durch einen solchen Synchronisationsmechanismus die kausale Ordnung verletzt wird.

**Eigener Verbesserungsvorschlag für die interne Struktur der HBQ** Eine kleine Verbesserung, um eventuell die Effizienz des Algorithmus zu steigern, ist es, die HBQ als Hash zu realisieren, um so eventuell weniger Vergleiche zu haben. Der Schlüssel wäre dann die eindeutige Pnum, die auf den Wert, welche eine Liste ist, zeigt. Die Liste bewahrt alle von Pnum empfangenen Vektorzeitstempel auf und hat an erster Stelle den kleinsten Vektorzeitstempel. Sollte nun überprüft werden, ob Nachrichten auslieferbar sind, werden von allen Schlüsseln, das erste Element aus den Listen entnommen und mit der lokalen Vektoruhr verglichen. Diese Annahme wurde noch nicht überprüft. Jedoch begleitete mich diese Idee während des Arbeitsprozesses und wollte es an dieser Stelle erwähnt haben.

**Zusammenfassung** In diesem Abschnitt wurde meine Vorgehensweise beschrieben, sowie die bei der Entwicklung auftretenden Probleme zusammengefasst. Im nächsten Abschnitt wird festgehalten, wie Anforderungen bei der Umsetzung sichergestellt werden.

### 3.5 Sicherstellung der Anforderung

Nachdem die Implementierung der einzelnen Methoden innerhalb der Module fertiggestellt war, habe ich mir darüber Gedanken gemacht, wie man die kausale Ordnung der Auslieferung von Nachrichten feststellen kann. Unter anderem werden durch das Test-System von Herrn Klauck und der Vorgabe der Multicast-Zentrale zwei sehr effiziente Mittel dargeboten, um die Anforderungen der Module zu testen und sicherzustellen, dass diese korrekt umgesetzt sind. Des Weiteren wurden, wie bereits erwähnt, Unit-Tests für die Vektoruhr-ADT erstellt, um die jeweiligen Funktionen der Schnittstellen zu verifizieren. Gleichzeitig habe ich auch recherchiert, ob es Tools gibt, die versuchen in verteilten Systemen die Nachverfolgbarkeit

von Aktionen visuell darzustellen und bin auf das Open-Source-Projekt “Tracing-Jaeger” gestoßen. Jedoch ist es mir nicht gelungen die [Jaeger-Client-Library für Erlang](#) aufzusetzen und habe mich mit den bewährten Mittel auseinandergesetzt. Hierfür bin ich die Szenarie aus Abbildung 12 durchgegangen und habe für jede meiner selbst-implementierenden Komponenten die Szenarien durchgespielt. (Diese befinden sich im Verzeichnis 1-testSystem der Abgabe).

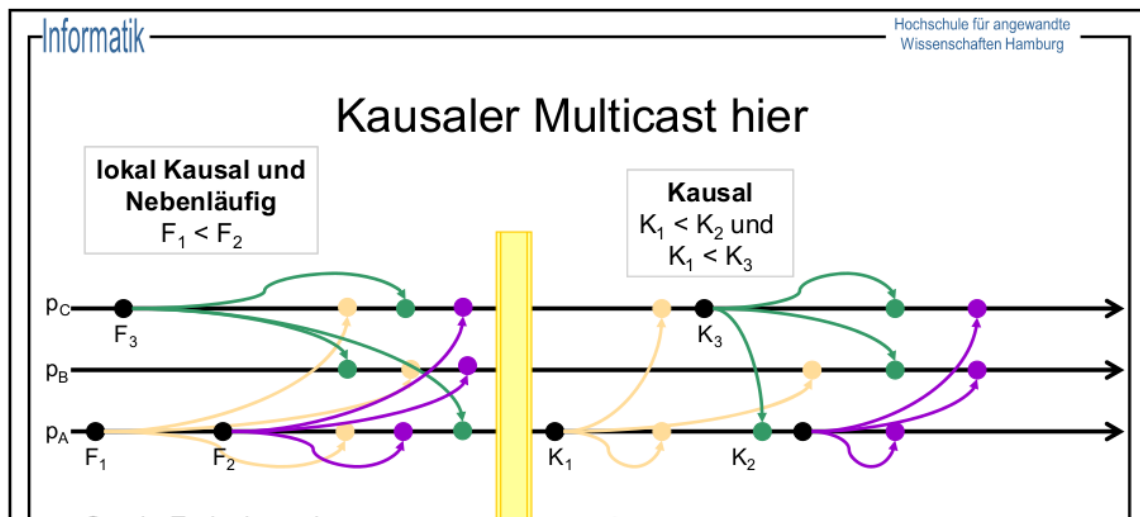


Abbildung 12: Beispielszenario zwischen 3 Prozessen: A, B und C  
Quelle: Foliensatz Aufgabe VS-Referat KLC Seite 3

## 4 Theorie und Praxis

Im letzten Abschnitt wird nochmal versucht, das Konzept der Vektoruhr zu analysieren und Vor- und Nachteile beim Einsatz von Vektoruhren zu beschreiben. Anschließend werden einige Anwendungsszenarien vorgestellt, wo kausaler Multicast eine Rolle spielen.

### 4.1 Analyse: Vor und Nachteile von Vektoruhren

In der Einführung dieser Hausarbeit wurde der Unterschied zwischen Lamport-Uhren und Vektoruhren erarbeitet und festgestellt, dass Lamport-Uhren nur eine schwache Konsistenz erbringen können und keine Rückschlüsse über die Historie der Ereignisse geben, da zu wenig Informationen abgespeichert werden. Die Vektoruhren hingegen bieten den Vorteil starke Konsistenz zu gewährleisten, und dementsprechend können ausgehend von Ereignisstem-peln Aussagen darüber getroffen werden, ob zwei Ereignisstem-pel in einer happens-before-Relation stehen. Demnach gilt:  $e_1 \rightarrow e_2 \Leftrightarrow L(e_1) < L(e_2)$ .

Die Tatsache, dass die Vektoruhren die Historie der Ereignisse mittels der Zeitstempel “aufzeichnen” birgt das Risiko, dass dem Netzwerk vertraut wird, dass keine Nachrichten verloren gehen. Dementsprechend ist eine Voraussetzung, dass der Multicast absolut zuverlässig realisiert ist und dies kann nicht in jedem Anwendungsszenario gewährleistet werden. Dementsprechend zeigen Vektoruhren ein blockierendes Verhalten, sobald eine Nachrichtenkette

nicht ausgeliefert werden kann, weil ein Ereignis aus dieser Nachrichtenkette noch nicht empfangen wurde.

Hieran ist aber auch zu erkennen, dass jeder Nachricht eindeutig ein Zeitstempel zugeordnet ist und der Datentransfer in der Hinsicht kompakter ist, als dass bei anderen Verfahren andere Datenstrukturen zum Einsatz kommen. Denn es gibt auch beispielsweise die Methodik, dass der historische Verlauf nicht über die Ereigniszähler festgehalten wird, sondern über die Nachrichten und demnach die vorausgegangenen Nachricht bei der Multicast-Nachrichten im Header beigefügt werden. Dadurch wachsen solche Pakete sehr schnell, besonders dann, wenn die Senderate hoch ist und gleichzeitig viele Teilnehmer an der Kommunikation teilnehmen.

Doch ein ähnliches Problem herrscht hier auch bei den Vektoruhren. Denn in verteilten Anwendungen ist die Skalierbarkeit ein enorm hohes Gut und demnach müssen die Designentscheidungen bewusst und reflektiert getroffen werden. Im Wesentlichen haben die Teilnehmerzahl der Gruppenkommunikation und die Rate der Ereignisse Auswirkungen auf die Robustheit der Vektoruhren. Die Länge der Vektoruhr wird festgelegt durch die Anzahl der Teilnehmer. Sie stehen in linearer Relation. Die Größe der Vektoruhr hat Auswirkungen auf die Größe des Datenpakets, die an der Nachricht beigefügt wird. "With vector slots of  $b$  bits in a system with  $n$  processes, we are adding a "tail" of  $(bn/8)$  bytes to each message. For instance, in a system with 200 processes and 32-bit slots, this adds 800 bytes to each message." [2] Ein solcher Overhead kann gerade in Verteilten Systemen einen enormen Defizit in Skalierbarkeit und Performance bedeuten. Jedoch muss hier auch wieder berücksichtigt werden, ob eine solche hohe Teilnehmerzahl überhaupt erwartet wird. Man nehme das Beispiel der replizierenden Datenbanken, da ist es nicht üblich in der einfachen Berufswelt 200 Datenbank-Replikate zu besitzen. Und wenn dies doch der Fall seine sollte, dann weiß man sich auch mit anderen Technologien zu helfen.

Natürlicherweise gibt es noch weitere Aspekte, die beispielsweise im Rahmen der Gruppenkommunikation berücksichtigt werden müssen, zum Beispiel die dynamische Sicht auf die Teilnehmerzahl, die wiederum auch Vor- und Nachteile mit sich bringen. Dabei kann die dynamische Sicht eine positive Auswirkung auf die Struktur der Vektoruhr haben. Wenn ein neuer Teilnehmer an einer bestehenden Gruppenkommunikation sich anmeldet, müssen die Vektoruhren zurückgesetzt werden. Somit kann die Vektoruhr nicht unendlich steigend wachsen und reduziert den Overhead. Jedoch bedarf es auch Kommunikationsaufwand, die Vektoruhren zum gleichen Zeitpunkt zurückzusetzen. Selbstverständlich gibt es auch andere Ansätze, diese Probleme zu lösen.

Anhand dieser kleinen Diskussion wird deutlich, dass Vektoruhren viele relevante Vorteile mit sich bringen, sei es die einfache Umsetzung eines kausalen Multicast-Protokolls, oder auch der Aspekt, dass die gesamte Historie der Ereignisse in den Vektoruhren abgespeichert werden. Nichtsdestotrotz müssen auch die Fallstricken der Vektoruhren berücksichtigt werden, um eine robuste und skalierbare Anwendung zu bauen und nicht das Gegenteil davon. In der Literatur findet man aber zahlreiche Bemühungen diese Fallstricken durch algorithmische Ansätze angepasst an die Vektoruhren zu lösen. So bin ich auf unterschiedliche Algorithmen gestoßen, die versuchen, die Größe der Vektoruhren zu komprimieren.[7]

## 4.2 Anwendungsszenario

Im Jahre 1985 wurden im Rahmen der Forschung ein System entwickelt, um Prozessgruppen zu verwalten und mit wechselnder Mitgliedschaft umzugehen. Dieses System und das damit einhergehende Protokoll ist unter dem Namen ISIS bekannt und hat Pionierarbeit geleistet, da es den Grundstein für Gruppenkommunikation und kausaler Ordnung gesetzt hat. In diesem Abschnitt wollen wir uns aus verschiedenen Perspektiven anschauen, wo kausaler Multicast in verteilten Systemen angewendet wird.

Aus dem Unterabschnitt Analyse haben wir herausgearbeitet, dass Vektoruhren eher in Anwendungen mit kleiner Teilnehmerzahl eingesetzt werden und die Skalierungsansprüche aufgrund der Billigung kausaler Ordnung fallen gelassen werden. Darüber hinaus muss toleriert werden, dass die Auslieferung von Nachrichten blockierend ist und eine hohe Zuverlässigkeit an Nachrichtentransport angestrebt wird.

Vektoruhren finden in verschiedenen Anwendungen Gebrauch. Aus der Recherche wurden folgende Bereiche identifiziert: Monitoring und Debugging von verteilten Anwendungen (Real Time Applications), Transaktionssystem, Message Publisher Systems, File Sharing Systeme sowie Replikationen von Datenbanken. Auch Anwendungen die im Sinne einer Konferenz geschehen, wie einem Gruppeneditor können Vektoruhren eingesetzt werden. Bevor nun die Ergebnisse aus der Recherche vorgestellt werden, möchte ich noch auf eine spontane Idee aufmerksam machen.

Im Rahmen der Veranstaltung “Verteilte Systeme” wurde im 1. Praktikum eine Server-Client-Anwendung implementiert, die eine einfache Chat-Applikation im Übertragenen Sinne realisiert. Ein Redakteur fasst Nachrichten, die durchnummeriert werden mit einer eindeutigen ID, und sendet diese Nachrichten an den Server, die dort anhand der ID sortiert werden und in der HBQ zurückgehalten werden, bis die Auslieferungsbedingung erfüllt ist, um die Nachrichten in die DLQ zu übertragen. Ein Leser fragt beim Server an, ob Nachrichten zum Lesen bereit stehen. Diese Anwendung könnte man nun so modifizieren, dass die Clients direkt miteinander kommunizieren und Nachrichten austauschen, ohne einen Server damit einzubinden. So könnte man das eingangs vorgestellte Problem der Chat-Applikation mittels einer Vektoruhr lösen, sodass auch die semantische Kausalität der Nachrichten gegeben ist.

Ein anderer Fall, wo die Vektoruhr genutzt werden kann, sind Transaktionssysteme. Ein Beispiel hierfür wird in diesem Artikel erörtert. [5] Es bezieht sich auf die verteilte Anwendung Ethereum und thematisiert, wie Transaktionsvorgänge miteinander in Zusammenhang gebracht werden können. Eine Ethereum-Transaktion ist im Wesentlichen eine Nachricht. Die Struktur der Nachricht enthält unter anderem Auskunft darüber, wie viel an Währung (hier Äther) transferiert werden soll und weiterer transaktionsnotwendiger Informationen, wie Empfänger etc. An einer Transaktion ist auch ein Zähler gebunden, der sich immer dann erhöht, wenn die Transaktion vom Adressinhaber bestätigt wird. Dies wird als Nonce bezeichnet. Diese Nonce hat die Rolle Replay-Versuche zu verhindern und die Reihenfolge der Transaktionen zu bestimmen. Demnach ist die Nonce einzigartig. Bereits an diesen Eigenschaften sind Ähnlichkeiten zum Konzept der logischen Uhr zu sehen, dass jedem Zeitpunkt ein Ereignis zugewiesen wird und eindeutig ist und dadurch Zeitpunkte eine kausale Ordnung zuweisbar sind. In diesem Sinne kann man auch das Transaktionssystem mittels einer

Vektoruhr realisieren und überträgt die Sende und Empfangereignisse auf die Transaktionsereignisse.

Ein weiteres Szenario um Vektoruhren zu verwenden, sind Echtzeitanwendungen eingebetteter Systeme, also Systeme die eine Monitoring-Funktionalität aufweisen. Wenn es darum geht eine Applikation oder eine Umgebung zu überwachen, wird die Verantwortlichkeit auf mehrere Nodes verteilt. Ein einfaches Beispiel sind Sensoren, die in der Wohnung Temperatur, Luftfeuchtigkeit und andere Vorkommnisse untersuchen, um beispielsweise bei erhöhter Temperatur die Wohnung zu kühlen. Demnach stehen die Nodes in Kommunikation miteinander, um relevante und wichtige Informationen miteinander auszutauschen. Es wurde bereits dargestellt, dass die Kommunikation über das Netz Verzögerungen mit sich bringt, und deshalb Nachrichten nicht in ihrer erwarteten Reihenfolge erscheinen. Aus diesem Grund ist es wichtig, dass die verteilten Knoten a) einen konsistenten Zustand aufweisen und b) eintreffende Nachrichten nach ihrer Kausalität unterscheiden können. Beispielsweise werden zwei Temperaturmessungen durchgeführt und anschließend den Gruppenteilnehmer zu gesendet. Nun wird angekommen, dass die zuletzt angekommene Temperaturmessung, die Aussage darüber was trifft, welche Reaktion bzw. Aktion geleistet wird. Belässt man es bei dieser Logik, kann es sein, dass alte Daten aufgrund der Verzögerung der Netzwerkübertragung zum Schluss ankommen. Mit Hilfe der Vektoruhren kann man gegen solche Phänomene entgegenwirken. Gleichzeitig ist die Anzahl der Sensoren klein genug, dass die Vektoruhren durchaus in diesem Szenario effizient arbeiten können.

Grundsätzlich gibt es auch andere Möglichkeiten in diesen Bereichen anderes zu verfahren, jedoch zeigt uns der abstrakte Ansatz der logischen Uhren seine Stärke, wenn es darum geht mit wenig Aufwand kausale Beziehungen zu konstruieren. Mit diesem Einblick in die praktische Welt und der damit Recherche ist festzustellen, dass Vektoruhren und kausale Multicast-Protokolle eine wichtige Relevanz sowohl in der Forschung als auch in der Arbeitswelt finden.

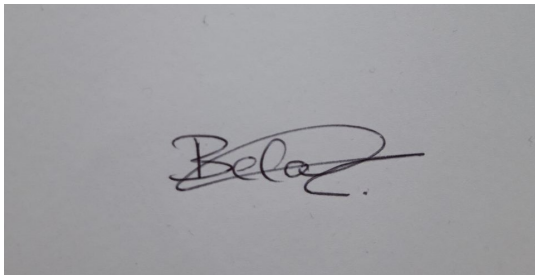
## 5 Fazit - Reflexion

Abschließend wurde im Rahmen dieser Hausarbeit eine Infrastruktur aufgesetzt, die kausalen Multicast mittels Vektoruhren ermöglicht. Die Anforderungen wurden hierzu in Form von Sequenzdiagrammen und Flussdiagrammen übertragen, um die internen und kommunikativen Abläufe der Anwendung zu verstehen. Des Weiteren wurde die Motivation für logische Uhren und Gruppenkommunikation erarbeitet, sowie ein Einblick in die Praxis und Forschung geworfen. Gleichzeitig wurde auch versucht die Effizienz von Vektoruhren und ihre Einschränkungen zu beschreiben.

Anfangs habe ich darauf aufmerksam gemacht, dass es für einen Informatiker wichtig sei, Strategien und Methoden zu kennen, um mit verteilten Anwendungen umzugehen. Seit Beginn der Ausarbeitung habe ich mich deshalb bemüht auch über den Tellerrand zu schauen und habe sehr frühzeitig angefangen, wissenschaftliche Artikel zum Thema Multicast, Kausalität und logische Uhren zu lesen. Zwar habe ich es nicht geschafft alle Artikel durchzuarbeiten und in vollem Maße zu verstehen, trotz dessen spüre ich, dass ich neue Einblicke in die Welt der Verteilte Systeme und ihre Probleme bekommen habe.

# ERKLÄRUNG ZUR SCHRIFTLICHEN AUSARBEITUNG DER HAUSARBEIT

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates 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 selbstständig 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, den 2. Juli 2021



## Literatur

- [1] David R. Cheriton und Dale Skeen. „Understanding the Limitations of Causally and Totally Ordered Communication“. In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, S. 44–57. ISBN: 0897916328. DOI: [10.1145/168619.168623](https://doi.org/10.1145/168619.168623). URL: <https://doi.org/10.1145/168619.168623>.
- [2] Ruben de Juan-Marin und Hendrik Decker. *Scalability approaches for causal multicast: a survey*. <https://doi.org/10.1007/s00607-015-0479-0>, Besucht am 24.06.2021. Dez. 2015.
- [3] Holger Karl. *Verteilte Systeme, Kapitel 7*. <https://www.youtube.com/watch?v=CcVeEIOkPKU&list=PLcVYkCRLcLtGHZfmkfYjdN8Ai9tkHaHvi>, Besucht am 28.06.2021. Okt. 2020.
- [4] Holger Karl. *Verteilte Systeme, Kapitel 7g, Lamport time , vector clocks, causal order using CBCAST*. <https://www.youtube.com/watch?v=zUSgoXPTjyo>, Besucht am 28.06.2021. Okt. 2020.
- [5] Lum Ramabaja. „The Binary Vector Clock“. In: *CoRR* abs/2004.07087 (2020). arXiv: [2004.07087](https://arxiv.org/abs/2004.07087). URL: <https://arxiv.org/abs/2004.07087>.
- [6] Chengzheng Sun und Wentong Cai. „Capturing causality by compressed vector clock in real-time group editors“. In: *Proceedings 16th International Parallel and Distributed Processing Symposium*. 2002, 8 pp-. DOI: [10.1109/IPDPS.2002.1015548](https://doi.org/10.1109/IPDPS.2002.1015548).
- [7] K.M. Zuberi und K.G. Shin. „A causal message ordering scheme for distributed embedded real-time systems“. In: *Proceedings 15th Symposium on Reliable Distributed Systems*. 1996, S. 210–219. DOI: [10.1109/RELDIS.1996.559724](https://doi.org/10.1109/RELDIS.1996.559724).