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.

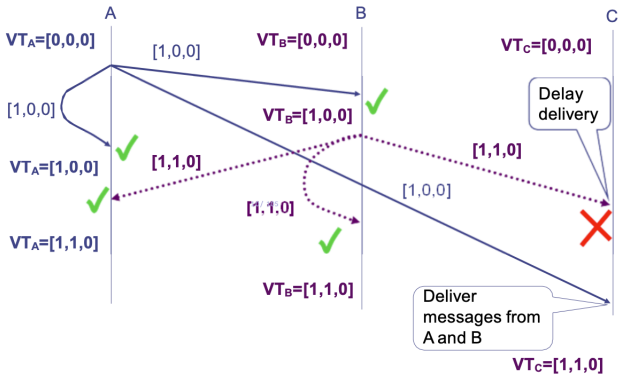

Abbildung 1: CBCAST [Kla24]

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 große 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 *Tower* 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.

1.2.1 Holdback Queue

Beim Empfangen einer Nachricht, wird diese zuerst in eine *Holdbackqueue* sortiert. Die *Holdbackqueue* ist eine *Priorityqueue* und enthält alle Nachrichten, die nicht ausgeliefert werden dürfen. Sortiert wird anhand der, der Nachricht angehängten, Vektoruhr.
Hierbei gibt es 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.

Ob Nachrichten auslieferbar sind und die Queue verlassen dürfen oder ob Nachrichten aus der Queue gelöscht werden müssen, wird geprüft, wenn neue Nachrichten der Queue hinzugefügt werden. Auslieferbar ist eine Nachricht, wenn diese nach oder gleich der, der *Delivery Queue* zuletzt hinzugefügten Nachricht, liegt und eine Distanz von -1 hat.
Falls es zu einem Stillstand im System kommt läuft zusätzlich ein Intervall-Timer, welcher die Auslieferbarkeitsprüfung regelmäßig aufruft.

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 der *Delivery Queue* synchronisiert.

1.3 Ungeordneter Multicast

Ein *Multicast* verteilt Nachrichten an Teilnehmer in einem Netzwerk. Der Unterschied zum *Broadcast* besteht darin, dass beim *Broadcast* Inhalte verbreitet werden, die – mit geeigneter Empfangsausrüstung – jeder ansehen kann, wohingegen beim *Multicast* vorher eine Anmeldung beim Sender erforderlich ist [Wik23].

Ungeordnet ist ein *Multicast*, wenn die Nachrichten nicht in der Reihenfolge weitergegeben werden, in der sich die jeweiligen Teilnehmer/Prozesse beim *Multicast* registriert haben.

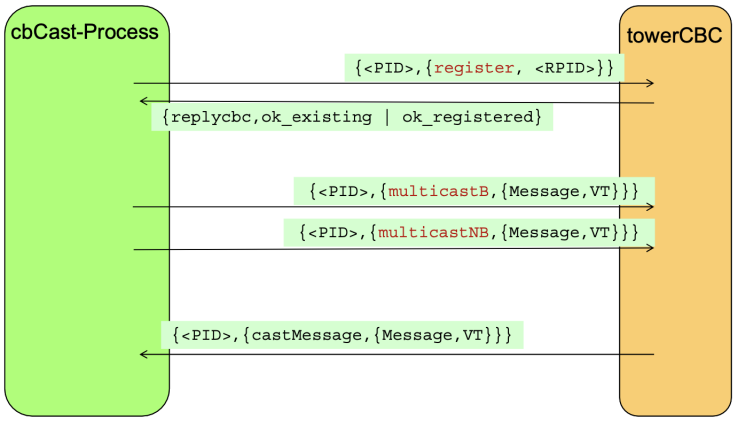

Abbildung 2: ungeordneter Multicast [Kla24]

In Abb. 2 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 wieder kann mit *castMessage* Nachrichten 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. 3 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.

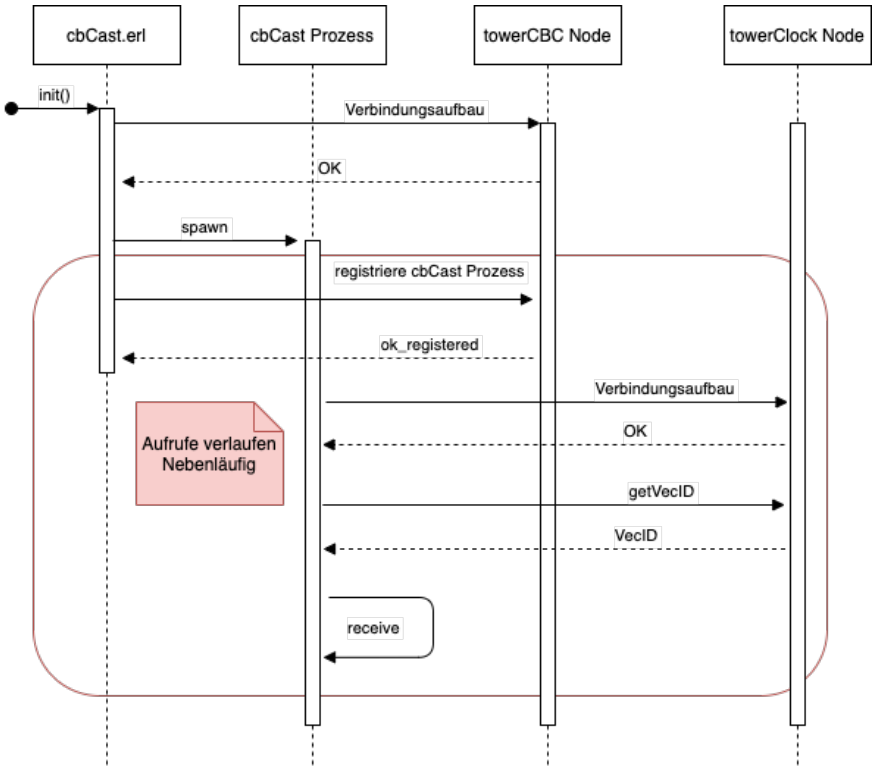

Abbildung 3: Sequenzdiagramm Initialisierung

Alternativ könnte sich der *cbCast.erl* auch erst beim *towerClock* die *VektorID* holen und dann den *cbCast* Prozess erzeugen. Durch den Ablauf in Abb. 3 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 verschiedene Wege. Zuerst wird der Prozess gestoppt, das bedeutet das eine 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. 4) wird eine Nachricht *(Msg)* an den als Parameter übergebenen *Kommunikationsprozess (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 könnte. Wenn die gesendete Nachricht vom *Multicast* empfangen wird, wird diese dennoch zuerst in die *Holdback Queue* einsortiert um eine kausale Ordnung zu garantieren.

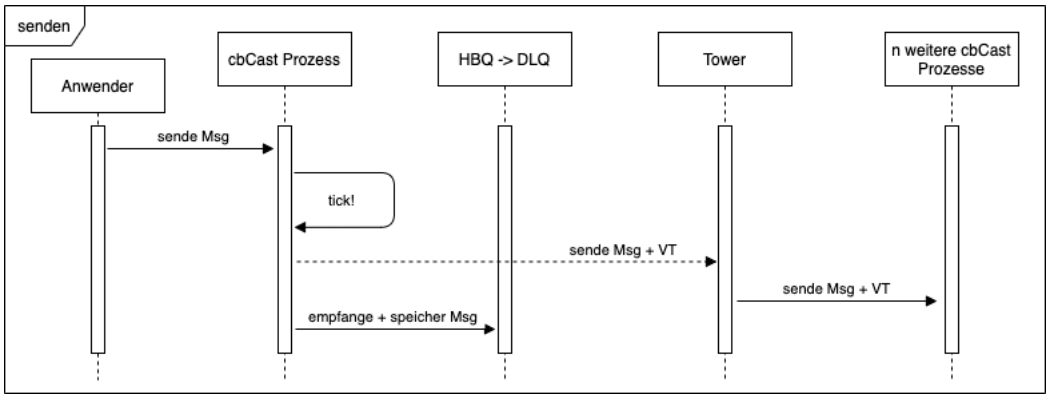

Abbildung 4: Sequenzdiagramm Senden

2.1.4 read/1 & received/1

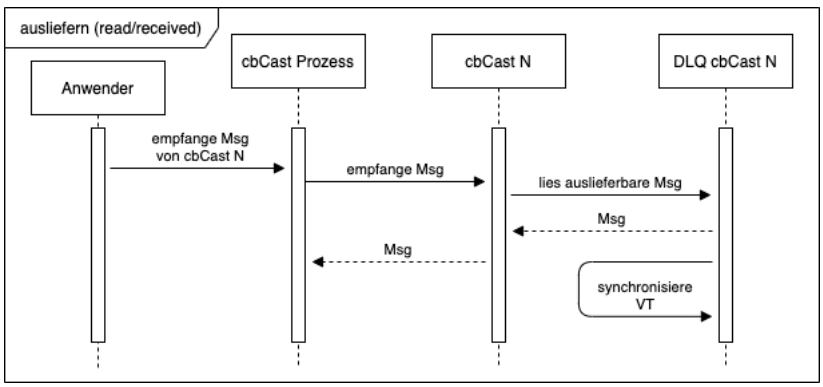

Abbildung 5: Sequenzdiagramm Ausliefern

Das Ausliefern einer Nachrichten (siehe Abb. 5) liest eine Nachricht aus der *Delivery Queue*

des *Kommunikationsprozesses (cbCast N)*, welcher als Parameter übergeben wurde. Hierfür 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 verschickt 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 $\{\langle PID\rangle, \{castMessage, \{\langle Message\rangle, \langle VT\rangle\}\}\}$ der *Kommunikationsprozesses* empfangen (siehe Abb. 6). Daraufhin wird die Nachricht in der *Holdback Queue* des Prozesses einsortiert.

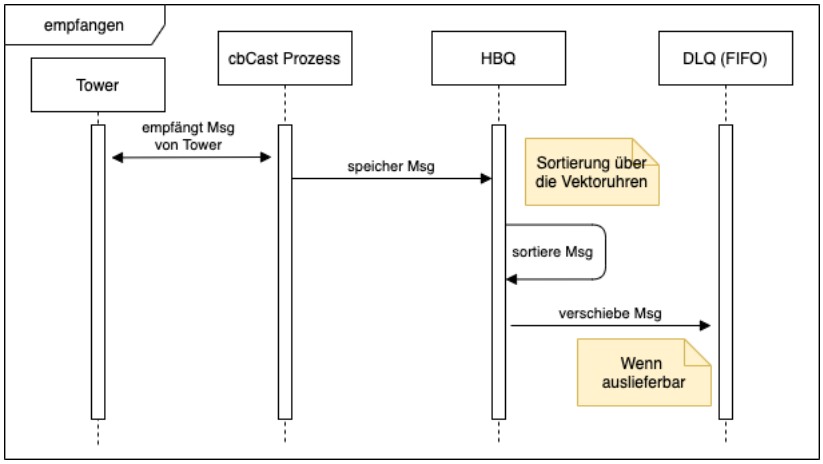

Abbildung 6: Sequenzdiagramm Empfangen

2.1.6 Die Queues

Sowohl die *Holdback* als auch die *Delivery Queue* sind Listen, welche die Nachrichten als Elemente enthalten. In der *Holdback Queue* werden die Nachrichten absteigend nach den Vektoruhren der jeweiligen Nachrichten sortiert. Ist eine Nachricht *after* (siehe 1.2.1) der ersten Nachricht der *Holdback Queue* wird diese am Anfang einsortiert. Ist die Nachricht *before* wird sie weiter mit den absteigenden Nachrichten verglichen, bis entweder das Ende der *Queue* erreicht wurde oder bis die Nachricht wieder *before* der an dem Index verglichenen Nachricht ist. Wenn eine Nachricht während des Einsortierens *concurrent* zu einer

Nachricht ist, wird sie vor dieser Nachricht einsortiert. Ist eine Nachricht *equal* zu einer Nachricht in der *Holdback Queue* wird sie verworfen.

checkQueues/3 TODO!

2.2 Vektoruhr-ADT

2.2.1 initVT/0

2.2.2 myVTid/1

2.2.3 myVTvc/1

2.2.4 myCount/1

2.2.5 foCount/2

2.2.6 isVT/1

2.2.7 syncVT/2

2.2.8 tickVT/1

2.2.9 compVT/2

2.2.10 aftereqVTJ/2

2.3 Vektoruhr Zentrale/Tower

Die zentrale Vektoruhr (*Tower*) verwaltet die Prozessnummern. Prozessnummern sind im Folgenden eindeutige IDs aus positiven ganzen Zahlen.

2.4 TowerClock

Der TowerClock implementiert eine wesentliche Schnittstelle, *getVecID*. Aus Gründen der Erweiterbarkeit und Sicherheit mappt die TowerClock die Prozess ID des anfragenden Prozesses auf dessen Vektoruhr ID.
Die kleinste Zahl für eine Vektoruhr ID ist 0.

2.5 Generelle Designentscheidungen

2.5.1 Logging

Es wird pro Node eine generische .log Datei geben. Zum Beispiel für den Node *'cbCast1@MacBook-Air-von-Kristoffer'* gibt es die Datei *'cbCast1@MacBook-Air-von-Kristoffer'.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 rausgestellt, dass diese Datei wenig Mehrwert bringt. Deswegen fehlt diese in der finalen Implementierung.

3 Realisierung

3.1 Vektoruhr-ADT

Die Vektoruhr (*vectorC*) wird in dieser Ausarbeitung als ein abstrakter Datentyp implementiert. Jeder Prozess hat seine eigene Vektoruhr *VT*. Um eine Identität und einen initialen Zeitstempel zu erhalten, muss sich jede Vektoruhr beim Tower (Kap. 2.3) melden. Wie in 2.1 bereits beschrieben und in Abb. 3 zu sehen, wird beim Aufruf der *init()* Funktion des *vectorC* ein Verbindungstest zur *towerClock* gestartet. Dieser terminiert das Programm, wenn keine Verbindung hergestellt werden kann.
Die allgemeine ADT der Vektoruhr ist ein Tupel aus dessen Vektoruhr ID als Integer und der Vektoruhr als Liste. Ein Beispiel ist *2, [1,3,4,2]*. Die Vektoruhr ID ist *2* und die Vektoruhr ist *[1,3,4,2]*. Da die Vektoruhr IDs bei 0 starten, wäre jetzt der eigene Zeitstempel dieser ADT *4*.

```
% TODO: muss noch in die Realisierung
VT = {2, [1,3,4,2]}.
vectorC:myCount(VT).
4
```

4 Analyse

4.1 Korrektheitsbeweis

4.2 Komplexitätsanalyse

5 Fazit

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.

(Ort, Datum) (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.

[Wik23] Wikipedia. *Multicast*. [Online; accessed 16-May-2024]. 2023. URL: https://de.wikipedia.org/wiki/Multicast.

A Anhang