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.

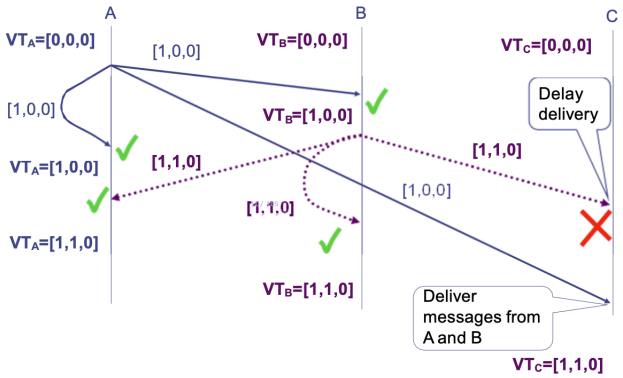

Abbildung 1: CBCAST [Kla24]

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.

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.
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. Falls es zu einem Stillstand im System kommt läuft zusätzlich ein Intervall-Timer, welcher die Auslieferbarkeitsprüfung regelmäßig aufruft.
Aus der Queue gelöscht werden, müssen Nachrichten welche

1.2.2 Delivery Queue

Die *Deliveryqueue* ist die zweite Queue eines Kommunikationsprozesses. Sie enthält alle auslieferbaren Nachrichten und hat eine eigene Vektoruhr. Diese Vektoruhr wird synchronisiert mit jeder Vektoruhr einer, in der *Deliveryqueue* neu hinzugefügten, Nachricht.

1.3 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. 1.4) melden.

1.4 Vektoruhr Zentrale/Tower

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

1.5 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].
Der TowerCBC (siehe Abb. 2) übernimmt in dieser Ausarbeitung die Aufgabe des ungeordneten Multicasts. Ungeordnet ist dieser, weil die Nachrichten nicht in der Reihenfolge weitergegeben werden, in der sich die jeweiligen Teilnehmer/Prozesse beim TowerCBC registriert haben.

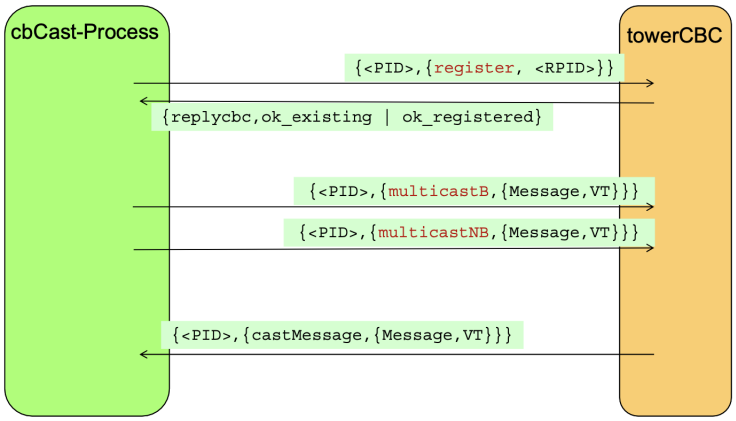

Abbildung 2: ungeordneter Multicast [Kla24]

In Abb. 2 ist eine abstrakte Kommunikation eines Prozesses mit dem Multicast 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.

1.6 Aufgabenstellung

Fragen:

- vektorclock zählt prozesse hoch -¿ auf überlauf des zählers achten - towercbc -¿ auto oder manu - bei manu empfängt der tower nur nachrichten. Aufrufe nur manuell von außen möglich - bei auto entscheidet tower selbst wann er welche nachrichten rausschickt. Schickt z.b. direkt raus. - blocking/nonblocking: im manuellen zustand kann nur einer der beiden verwendet werden. -

Infos von Klauck aus der VL heute:

1) Eine Nachricht verlässt die dlq wenn der anwender an entsprechender comm einheit die funktion read oder receive aufruft.

2) Eine dlq hat eine eigene vektoruhr

3) Beim send vom anwender schickt dieser seine Vektoruhr mit und die Nachricht wird in hbq der anderen comm module einsortiert und eventuell verglichen mit der vektoruhr der dlq. (siehe Bild)

4) Beim sendeereignis tickt die uhr des anwenders. wenn sende und empfangsereignisse in der Reihenfolge unklar sein könnten, dann müsste auch beim empfangsereignis getickt werden (Bei uns aber nicht)

5) im cbcast.erl steht das tick() im send() (zuerst)!

6) bei read und received wird sync aufgerufen (für uhr des anwenders) und ggf. auch in der dql

7) dlq kann auch länge 1 haben, macht er aber nicht so

8) wenn dlq leer ist und hbq hat lieferbare nachricht, dann sollte diese direkt in die dlq ohne das irgendwas von außen kommt - durch z.B. regelmäßige zeitliche prüfungen

9) Empfehlung von ihm: nachrichten mit after und beforeVT in HBQ einsortieren wenn concurrent, dann müssen alle geprüft werden (Wenn der vordere nicht rüberdarf, dann die danach ziemlich sicher auch nicht)

2 Entwurf

2.1 Kommunikationseinheit

Beim Initialisieren einer Kommunikationseinheit wird ein Prozess gestartet, welcher beim *towerCBC* registriert wird und sich bei der *towerClock* eine neue Vektoruhr ID holt. 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 also der Prozess, welcher den *cbCast* Prozess erzeugt diese Anfrage schicken, wäre dieses Mapping sinnlos.
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.

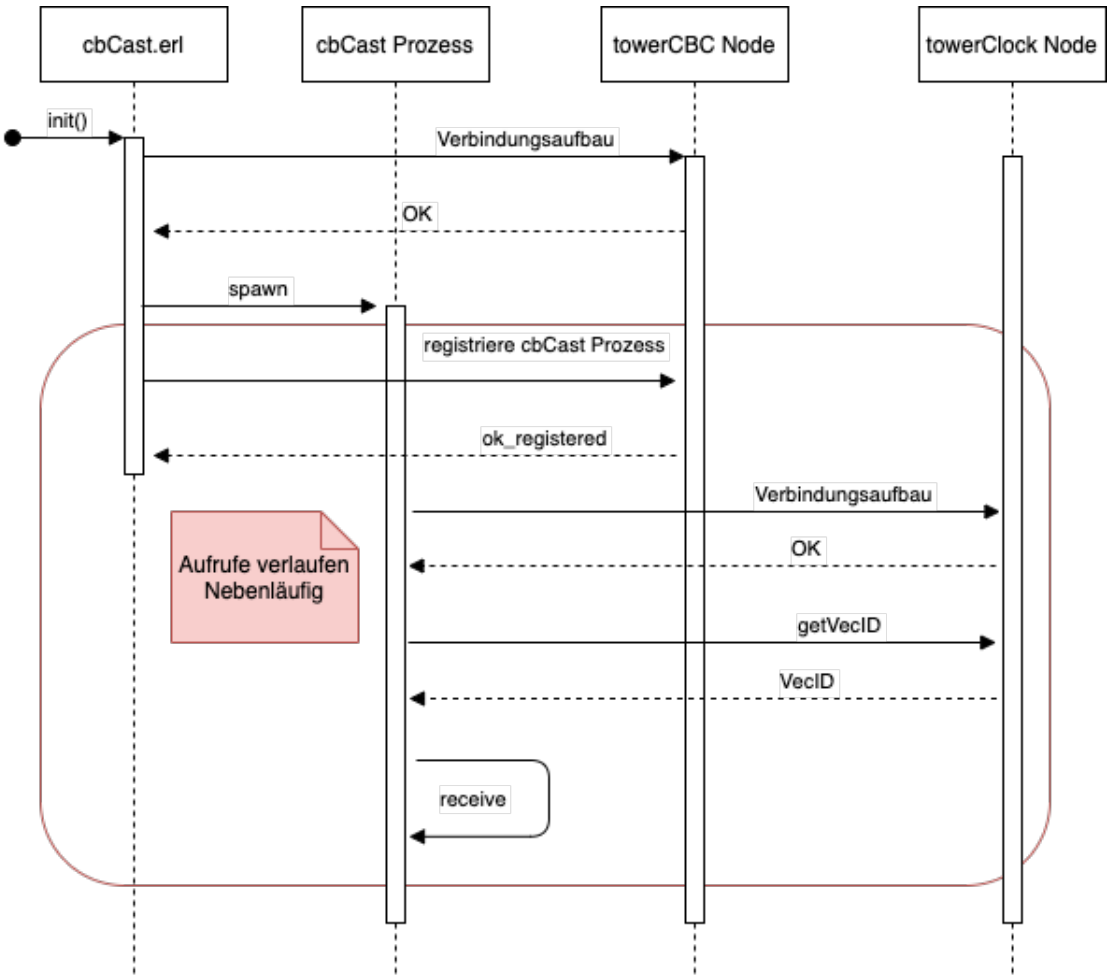

Abbildung 3: Sequenzdiagramm cbCast

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.

2.2 Vektoruhr-ADT

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
```

2.3 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.

2.4 Generelle Designentscheidungen

2.4.1 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

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