NVS Projekt: Synchronisation durch den verteilten Algorithmus

Florian Schwarzl

10. Januar 2021

Inhaltsverzeichnis

1 Einleitung

Die Synchronisation eines Ablaufs, welcher auf mehreren Systemen verteilt abgearbeitet wird, ist ein essentieller Teil moderner Softwareentwicklung. Durch die immer näher Rückende Grenze der Hardware-Leistung müssen Programme, um trotz zunehmender Komplexität eine kurze Laufzeit zu erreichen, parallelisiert werden. Beim Bearbeiten einer Aufgabe gibt es jedoch meist Abschnitte, die nicht parallel abgearbeitet werden können. Beispielsweise das Zusammenführen der Ergebnisse. Damit es in eben diesen kritischen Abschnitten zu keiner Race-Condition[1] oder ähnlichen Problemszenarien kommt, ist eine Möglichkeit des gegenseitigen Ausschlusses (Engl.: mutual exclusion) notwendig. Dadurch kann sichergestellt werden, dass keine zwei Knoten des verteilten Systems zeitgleich in einen kritischen Abschnitt eintreten. Diese Ausarbeitung befasst sich mit diesem Problem und im spezifischen mit dem Lösungsansatz der verteilten Synchronisation und wie dieser im beiliegenden Programm umgesetzt wurde.

2 Algorithmus

Der Synchronisationsalgorithmus, welcher hier implementiert wurde, ist jener von Glen Ricart und Ashok K. Agrawala[1]. Er beschreibt einen dezentralen Ansatz zum gegenseitigen Ausschluss in einem verteilten Computersystem. Dezentral bedeutet in diesem Zusammenhang, dass, anders als bei einem zentralen Algorithmus, es eben keinen zentralen Koordinator gibt, welcher sich um die Synchronisation kümmert, sondern sich die Knoten untereinander die Reihenfolge des Eintritts in den kritischen Abschnitt ausmachen.

Der Algorithmus basiert auf einem System von Anfrage- und Antwortnachrichten, welche von jedem Knoten an jeden anderen Knoten gesendet werden können. Dadurch entsteht die Notwendigkeit, dass jeder Knoten mit jedem anderen Kommunizieren kann. In weiterer Folge muss jeder Knoten eine Liste aller anderen Knoten halten, anders als bei einem zentralisierten Algorithmus, wo nur der zentrale Koordinator alle Knoten kennen muss.

2.1 Vorteile und Nachteile

Ein offensichtlicher Vorteil jedes verteilten Algorithmus ist, dass es keinen single-point-of-failure gibt. Der Algorithmus selbst hat zwar keine Ausfallserkennung eingebaut, diese kann jedoch beispielsweise durch ein Timeout nach einer bestimmten Zeit realisert werden.

Ein Nachteil dieses Algorithmus ist einerseits, dass er komplizierter zu Implementieren ist. Da jeder Knoten gleichrangig ist, ist deutlich mehr Aufwand notwendig, um zu einem gegenseitigen Ausschluss zu kommen. Auch ist die Anzahl der Nachrichten ein Problem, da dadurch die Skalierbarkeit verschlechtert wird. In einem System mit n Knoten fallen für jeden Knoten, der den kritischen

[1] Ergebnis der Operation ist vom zeitlichen Ablauf abhängig.

Abschnitt betreten will $2*(n-1)$ Nachrichten an. Es werden $n-1$ Anfragenachrichten versendet und ebenfalls $(n-1)$ Antwortnachrichten empfangen.[1]

2.2 Ablauf

Der Ablauf des Algorithmus basiert darauf, dass jeder Knoten eine eigene logische Uhr, mit deren Hilfe jede Nachricht mit einem Zeitstempel versehen wird. Dieser Zeitstempel wird zum Vergleich der Nachrichten herangezogen und legt damit indirekt fest, welcher Knoten zuerst in den kritischen Abschnitt eintreten darf. Anstatt des Zeitstempels könnte auch eine zufällig generierte Zahl verwendet werden, die Verwendung des Zeitstempels sorgt jedoch dafür, dass jene Knoten, welche die Nachricht zuerst gesendet haben, auch wahrscheinlich zuerst in den kritischen Abschnitt eintreten dürfen. Mit diesem Wissen kann nun der Ablauf wie folgt beschrieben werden[2]:

1. Ein Knoten, der in den kritischen Abschnitt eintreten will, sendet eine Nachricht (Request) mit dem Zeitstempel an alle anderen Knoten.
2. Erhält ein Knoten eine Nachricht, hängt die Antwort von seinem derzeitigen Zustand ab:
   (a) Befindet sich der Knoten nicht im kritischen Abschnitt und will diesen auch nicht betreten, sendet er OK zurück.
   (b) Befindet sich der Knoten im kritischen Abschnitt, speichert er die Anfrage in einer Warteschlange.
   (c) Befindet sich der Knoten nicht im kritischen Abschnitt, will diesen jedoch betreten, dann wird der Zeitstempel der eingehenden Nachricht mit dem eigenen (aus der eigenen Nachricht) verglichen. Ist der eingehende kleiner, wird ein OK an den Sender gesendet, andernfalls kommt der Request in die Warteschlange.
3. Hat ein Knoten von jedem anderen Knoten ein OK auf seine Anfrage erhalten, tritt er in den kritischen Abschnitt ein.
4. Nach dem Verlassen des kritischen Abschnitts entnimmt der Knoten die gespeicherten Anfragen aus der Warteschlange und sendet ein OK an die ursprünglichen Sender zurück.

Durch diesen Ablauf ist sichergestellt, dass zu jedem Zeitpunkt höchstens ein Knoten im kritischen Abschnitt befindet.

3 Implementierung

Das Programm implementiert diesen Algorithmus und simuliert ihn in einer lokalen Umgebung, wobei die einzelnen Knoten durch Threads repräsentiert werden. Die Anzahl der Knoten kann über Kommandozeilenparameter beim Aufruf des Programm geändert werden. Sollte kein Parameter übergeben werden, wird

der Standardwert von 3 verwendet. Der Algorithmus ist in 2 Klassen implementiert. Die Kommunikation zwischen den Threads erfolgt über eine Pipe, welche in pipe.h impementiert ist. Eine Nachricht wird durch die Klasse Message dargestellt.

Die Nachrichten werden der Pipe über eine Funktion übergeben. Intern werden die Nachrichten in einer Warteschlange gespeichert. Die Pipe ist durch einen Mutex thread-safe. Das Entnehmen eines Wertes aus der Pipe ist ein blockierender Aufruf, der dann blockiert, wenn die Pipe leer ist. Sollte dies der Fall sein, wird durch eine Bedingungsvariable darauf gewartet, dass ein Wert in die Pipe gegeben wird. Erst wenn ein Wert aus der Pipe entnommen wurde, setzt das Programm fort. Durch diese Implementierung ist es leicht, auf einkommende Nachrichten zu hören.

3.1 Messsage-Klasse

Diese Klasse stellt eine Nachricht dar. Sie enthält Sowohl die Nummer des Senders als auch jene des Empfängers. Der Typ der Nachricht wird als Enum gespeichert und gibt an, ob es sich um eine Anfrage oder eine Antwort handelt, wobei im Algorithmus eine Antwort mit einer Bestätigung gleichgestellt ist.

Der Wert einer Nachricht ist der Zeitstempel, den der Worker beim Erstellen des Requests festlegt.

3.2 Worker-Klasse

In dieser Klasse ist jener Teil des Algorithmus implementiert, der sich mit der zeitlichen Regelung befasst. Das heißt, in dieser Klasse wird durch eine Zufallsvariable zwischen 3 und 5 Sekunden festgelegt, wann der Knoten in den kritischen Abschnitt eintreten will und wie lange er in diesem verbleibt. Zur Simulation des kritischen Abschnitts wird der Thread, in dem sich der Worker befindet, schlafen gelegt. Dieser Ablauf läuft in einer Endlosschleife, so dass sich die Simulation immer wieder wiederholt. Die Worker-Klasse hat zwei bool'sche Variablen, die ihren Status beschreiben. Diese zwei Variablen beschreiben, ob der Knoten in den kritischen Abschnitt eintreten will und ob er sich gerade in diesem befindet. Sie sind notwendig, um diese Information dem MessageHandler mitzuteilen. Dieser muss, je nach dem, in welchem Zustand sich der Worker befindet, die eingehenden Nachrichten anders behandeln.

```
1  wants_to_enter = false;
2  fmt::print("Worker {} no longer wants to enter critical section\n",
       Id);
3  this_thread::sleep_for(chrono::milliseconds(int (dis(gen) * 1000)))
       ;
4  fmt::print("Worker {} wants to enter critical section.\n", Id);
5  wants_to_enter = true;
6
7  chrono::system_clock::time_point timestamp{chrono::system_clock::
       now()};
8  mh.set_enter_timestamp(timestamp);
9  mh.send_to_all(MessageType::REQ, timestamp);
10 fmt::print(fg(fmt::color::blue), "Worker {} sent a Request with
       Timestamp {:%H:%M:%S} to all other Workers.\n", Id, timestamp);
11
12 unique_lock ul{mtx};
13 can_enter.wait(ul, [this] { return got_all_ok; });
14 in_crit_section = true;
15 got_all_ok = false;
16 fmt::print(fg(fmt::color::gold), "Worker {} entered critical
       section.\n", Id);
17 this_thread::sleep_for(4s);
18 fmt::print(fg(fmt::color::gold), "Worker {} left critical section.\
       n", Id);
19 in_crit_section = false;
20 mh.done();
```

Listing 1: Zeitliche Regelung

Im vorhergehenden Code-Beispiel ist der in einer Endlosschleife laufende Code des Worker-Threads dargestellt. Zu Beginn eines Durchlaufs will Worker nicht in den kritischen Abschnitt eintreten und wartet eine zufällige Zeitspanne zwischen 3 und 5 Sekunden ab (Zeile 3). Nach dieser Pause will der Worker in den Abschnitt eintreten und erstellt sich einen Zeitstempel für die Anfrage, die er dann über den MessageHandler (welcher hier die Variable mh ist) an die anderen Knoten weitergeleitet. Danach wird in

3.3 MessageHandler-Klasse

Diese Klasse ist, als Thread gestartet, für die Verwaltung der Nachrichten eines Workers zuständig. Diese Klasse beinhaltet eine Pipe, die für die eingehenden Nachrichten verwendet wird und als Inbox bezeichnet wird. Um mit den anderen Knoten zu kommunizieren, hat jede Instanz der Klasse einen Vektor mit einer Referenz auf die Pipes der jeweils anderen Knoten.

```
1 // Alle Nachrichten an den Worker werden in seine Inbox geschrieben
2 Pipe<Message> inbox;
3
4 // Enthaelt Zeiger zu den Inboxen der anderen Worker
5 std::vector<Pipe<Message>*> outboxes;
```

Listing 2: Inbox und Outbox

In die Inbox werden alle Nachrichten geschrieben, die an diesen Knoten gerichtet sind. Somit schreiben in die Inbox immer die anderen Knoten und nur

jener, dem die Inbox gehört, liest aus dieser aus.

In dieser Klasse wird auch der Ablauf implementiert, wie er zuvor in 2.2 Ablauf beschrieben wurde. Der Status des Workers, also ob er in den kritischen Abschnitt eintreten will oder nicht, wird dem MessageHandler durch zwei Pointer vom Typ bool mitgeteilt. Bei jedem Durchgang dieses Ablaufs wird eine Nachricht aus der Inbox entnommen und bearbeitet. Je nach Status wird dann entweder eine Antwort gesendet, oder die Nachricht in der Warteschlange gespeichert. Der gesamte Ablauf läuft in einer Endlosschleife.

```
1  if (m.get_message_type() == MessageType::REQ) {
2      if (!*wants_to_enter && !*in_crit_section) {
3          send_ok(m.get_sender());
4      } else if (*wants_to_enter && !*in_crit_section) {
5          if (m.get_value() < enter_timestamp) {
6              send_ok(m.get_sender());
7          } else {
8              mq.push(m);
9          }
10     } else if (*in_crit_section) {
11         mq.push(m);
12     }
13 } else if (m.get_message_type() == MessageType::OK){
14     ok_cnt++;
15 }
```

Listing 3: Bearbeitung einer Nachricht

Dieser Abschnitt an Code aus der MessageHandler-Klasse entspricht dem zweiten Punkt des Ablaufs des Algorithmus. Er wird nach jeder erhaltenen Nachricht ausgeführt. Zuerst wird überprüft, ob es sich bei der eingetroffenen Nachricht um eine Anfrage oder eine Bestätigung handelt. Ist es eine Bestätigung, wird der Zähler für die Bestätigungen inkrementiert. Anhand dieses Zählers wird festgestellt, ob ein Knoten in den kritischen Abschnitt eintreten darf. Er darf dies nämlich sobald er von jedem anderen Knoten ein OK erhalten hat, also sobald er $n-1$ Antworten erhalten hat, wobei n die Gesamtanzahl de Knoten ist. Sollte es sich um eine Anfrage handeln, wird der Algorithmus abgearbeitet. Zuerst wird in Zeile 2 der Fall überprüft, dass der Knoten weder in den kritischen Abschnitt eintreten will, noch sich in diesem befindet. In diesem Fall wird eine Bestätigung an den Sender zurückgeschickt (siehe Zeile 4). Sollte der Knoten in den kritischen Abschnitt eintreten wollen, sich jedoch nicht in diesem befinden, wird die zweite Bedingung in der Zeile 5 ausgelöst. In diesem Fall wird der eigene Zeitstempel mit jenem aus der angekommenen Nachricht verglichen. Ist der eingehende Zeitstempel kleiner, wird eine Bestätigung an den Sender gesendet, andernfalls wird die Anfrage in der Warteschlange gespeichert. Sollte sich der Knoten zum Zeitpunkt des Eintreffens der Nachricht im kritischen Abschnitt befinden, wird diese ebenfalls in der Warteschlange gespeichert.

Die Warteschlange wird durch den Aufruf der Methode `done()` abgearbeitet. Sobald der Worker aus dem kritischen Abschnitt austritt, ruft dieser die Methode auf und der MessageHandler sendet an jede in der Warteschlange gespeicherte Nachricht eine Bestätigung zurück.

4 Bedienung

Das Programm kann nur durch den Aufruf von `./dist_sync` gestartet werden. Die Anzahl der Knoten ist dabei standardmäßig auf 3 gesetzt. Beendet wird das Programm durch Tastaturunterbrechung (`Ctrl + C`). Als Ausgabe in der Konsole erschienen verschiedene Statusmeldungen, die angeben, was derzeit im Programm passiert und welcher Knoten sich im kritischen Abschnitt befindet. In der nachfolgenden Tabelle sind alle Parameter angegeben:

Parametername	**Datentyp**	**Standardwert**	**Beschreibung**
	ganze Zahl >= 0	3	Anzahl der Knoten
`-h, --help`			Gibt die Hilfe aus
`-l, --log`		false	Aktiviert Logging nach `dist_sync_log.log`
`-d, --debug`		false	Setzt Log-Level auf debug, nur mit -l verwendbar

Ein Aufruf des Programms mit 0 als Parameter ist möglich, führt jedoch dazu, dass keine Knoten erzeugt werden und sich das Programm somit wieder beendet, ohne eine Ausgabe zu erzeugen. Dieses Verhalten ist geplant und so gewollt. Sollte eine negative Zahl übergeben werden, wird eine entsprechende Fehlermeldung ausgegeben und das Programm beendet sich.

5 Probleme bei der Umsetzung

Die größten Schwierigkeiten bei der Umsetzung bereitete die Aufteilung der Aufgaben eines Knoten in zwei Teile. Diese Aufteilung war aus meiner Sicht jedoch notwendig, da ein Knoten, während er sich im kritischen Abschnitt befindet, immer noch auf Nachrichten reagieren muss. Dementsprechend habe ich die Aufteilung so vorgenommen, dass die Behandlung der Nachrichten in einem separaten Thread durchgeführt wird. Das Problem lag primär in der Kommunikation zwischen dem MessageHandler-Thread und dem Worker-Thread. Derzeit erfolgt die Kommunikation über Pointer und eine Bedingungsvariable.

Literatur

[1] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[2] Dr. Christian Werner. Verteilte systeme: 5. synchronisation.