1. NVS Projekt: Wahlalgorithmus im Ring

Sebastian Grman

10. Jänner 2021

Inhaltsverzeichnis

1 Einleitung

In einem verteilten System mit geteilten Ressourcen ist es wichtig sicherzustellen, dass verschiedene Prozesse sich nicht gleichzeitig in sich ausschließenden Prozeduren befinden, also gleichzeitig auf die selbe Ressource versuchen zuzugreifen. Eine Möglichkeit, um eine globale Ressource und die Zugreife darauf zu verwalten, ist, eine Art leitende Aufsicht, einen *Leader* bzw. Leiter, zu haben. Dieser Leiter muss wiederum aus der Menge der Teilnehmer an einem verteilten System bestimmt werden und, sofern diese Entscheidung nicht im Voraus mittels Konfiguration getroffen wird und das System die Wahl selbstständig treffen soll, braucht man einen Wahlalgorithmus dafür.

2 Algorithmus

Ein Wahlalgorithmus, welcher sich für ein verteiltes System mit der Topologie eines Ringes eignet, ist der Algorithmus nach *Chang und Roberts*[2].Dieser ist insbesondere für gerichtete Ringe angebracht, da sämtliche Kommunikation nur in eine Richtung ausgeführt wird.

2.1 Anforderungen und Annahmen

Die grundlegendsten Anforderungen an ein jeden Wahlalgorithmus in einem verteilten System sind:

- Nach Ende der Wahl muss es genau einen Leiter geben.
- Nach Ende der Wahl wissen alle Prozesse, wer der neue Leiter ist.

Diese Anforderungen kann der Chang-Roberts-Algorithmus grundsätzlich erfüllen. Damit der Algorithmus auch wirklich wie erwartet funktioniert, müssen noch folgende Annahmen über das System zutreffen[1]:

- Der Algorithmus ist dezentral, d.h. jeder Prozess ist in der Lage den Algorithmus, also eine Wahl, zu initiieren.
- Jeder Prozess hat eine unter allen Prozessen einmalig vorkommende Kennung, *Id*, und diese *Id*s unterliegen einer Totalordnung.
- Alle Prozesse agieren bei der Wahl lokal nach dem selben Algorithmus.

Treffen diese Annahmen zu, so wird der Chang-Roberts-Algorithmus wie erwartet funktionieren und dem System zu genau einem Leiter, der allen bekannt ist, verhelfen.

2.2 Ablauf

Beim Chang-Roberts-Wahlalgorithmus wird der Prozess mit der größten Id zum Leiter ernannt. Der Ablauf, wie man zu diesem Endstand kommt sieht wie folgt aus:

1. Der Ausgangspunkt ist, dass kein Prozess an einer Wahl teilnimmt.
2. Ein Prozess, der das Fehlen eines Leiters bemerkt, leitet eine Wahl ein. Er stellt sich selber zur Wahl auf, indem er eine Nachricht mit seiner Id als Wahlvorschlag an seinen Nachbarn schickt – da die Wahl in einem gerichteten bzw. unidirektionalen Ring stattfindet, hat jeder Prozess nur einen Nachbarn, an den er schicken kann, und einen, von dem er empfängt – und markiert sich selber als Wahlteilnehmer.
3. Jeder Prozess, der einen Wahlvorschlag erhält und
   (a) noch kein Wahlteilnehmer ist, markiert sich als Wahlteilnehmer und
      i. sendet den Wahlvorschlag weiter, wenn seine Id niedriger als die vorgeschlagene ist.
      ii. sendet seine eigene Id als Wahlvorschlag an seinen Nachbarn, wenn diese höher als die vorgeschlagene ist.
      iii. der Vorschlag kann nicht gleich der eigenen Id sein, da jeder Prozess eine einmalige hat (siehe Annahmen 2.1).
   (b) bereits Wahlteilnehmer ist,
      i. ignoriert den Wahlvorschlag, wenn dieser niedriger als die eigene Id ist – dadurch werden überflüssige Wahlen vorzeitig beendet.
      ii. sendet den Wahlvorschlag weiter, wenn sein Id niedriger als die vorgeschlagene ist – ein anderer Prozess mit einer höheren Id wird die überflüssige Wahl beenden.
      iii. wird zum neuen Leiter und beginnt die Wahl zu beenden, wenn der Vorschlag gleich der eigenen Id ist.

4. Wenn ein Prozess zum neuen Leiter gewählt wurde, bedeutet es, dass sein Wahlvorschlag mit seiner Id eine ganze Runde bis zu ihm zurück gemacht hat. Man kann sich also sicher sein, dass er die höchste Id von allen Prozessen im Ring hat. Der Prozess beendet nun seine Wahlteilnahme und beginnt die Wahl zu beenden, indem eine Nachricht, dass er die Wahl gewonnen hat, mit seiner Id an seinen Nachbarn schickt.

5. Jeder nicht gewählte Prozess beendet ebenfalls seine Wahlteilnahme, sobald ihn die Nachricht vom Wahlsieger erreicht hat, und leitet diese weiter.

6. Hat die Nachricht vom Wahlsieg eine ganze Runde gemacht und ist beim Wahlsieger und Leiter wieder angelangt, leitet dieser sie nicht mehr weiter und die Wahl ist beendet.

Sofern die Kommunikation fehlerfrei abgelaufen ist, also alle Prozesse vollständig an der ganzen Wahl teilnahmen und keine Nachrichten verloren gingen, hat man am Ende ein System, das sich bereits wieder im Ausgangszustand für eine mögliche nächste Wahl befindet – kein Prozess ist als Wahlteilnehmer markiert –, mit einem Leiter und alle wissen wer dieser ist.

Für einen erfolgreichen Wahlablauf benötigt man bei n Prozessen im Ring und einem Initiator im ungünstigsten Fall $3n - 1$ Nachrichten je zwischen zwei Nachbarn. Dieser ungünstigste Fall tritt ein, wenn der Initiator der Nachbar nach dem prädestinierten Leiter, dem Prozess mit der höchsten Id, ist. In diesem Fall braucht es $n - 1$ Nachrichten, damit der prädestinierte Wahlsieger an der Wahl teilnimmt und sich vorschlägt, n Nachrichten, damit ihn sein Vorschlag wieder erreicht und er gewählt ist, und weitere n Nachrichten, damit alle anderen auch davon erfahren.

2.3 Alternativen

Der Chang-Roberts-Algorithmus ist durchaus nicht der effizienteste Wahlalgorithmus, den man im Ring verwenden kann. So gibt es zum Beispiel *Franklins Algorithmus*[3] für ungerichtete Ringe und den *Dolev-Klawe-Rodeh-Algorithmus*, welcher versucht Verbesserungen von Franklins Algorithmus auf gerichtete Ringe zu bringen[1].

3 Implementierung

Der grundlegende Chang-Roberts-Algorithmus ist nicht besonders schwer, doch kann man bei der Implementierung einen gewissen Entscheidungsspielraum versuchen zu erkunden und den Algorithmus auch mit einer sichereren Kommunikation unterstützen.

3.1 Ring

Bei der Ring-Klasse und dem konkreten Ring-Objekt handelt es sich im Grunde um einen eher einfachen Builder, welcher die Arbeiterprozesse mit einzigartigen

Ids anlegt und sie über Zeiger in einer Ringstruktur miteinander verbindet. Weiter ist Ring auch für das starten und stoppen der Arbeiterprozesse verantwortlich und bietet mit der Möglichkeit, Wahlen zu initiieren, auch eine grundlegende Schnittstelle für die Benutzung im Programm.

3.2 Worker

Jeder Arbeiter hat eine Id, kennt seine Position im Ring und hat auch Verweise auf alle Arbeiter im Ring, sich selbst eingeschlossen. Dadurch kann ein Arbeiter beim Ausfall seines Nachbars, die Nachrichten an den nächsten Prozess, also seinen neuen Nachbarn, leiten und das System kommt nicht zum erliegen, da alle Nachrichten bei einem ausgefallenen Prozess hängen bleiben.

Jeder Arbeiter hat seinen eigenen Nachrichtenpuffer, zu welchem er eine Schnittstelle zur Verfügung stellt und von dem er Nachrichten von seinem Nachbarn in Gegenrichtung oder von der Ring-Schnittstelle – in dem Fall wird der Arbeiter stoppen oder eine Wahl initiieren müssen – erfasst. Je nachdem, um welche Art von Nachricht es sich handelt und welchen Inhalt sie hat, wird entsprechend gehandelt.

Der Arbeite implementiert den Chang-Roberts-Wahlalgorithmus so wie in Ablauf 2.2 beschrieben, mit dem Zusatz, dass wenn der noch amtierende Leiter, falls es ihn noch gibt, einen Wahlvorschlag erhält, dieser nicht nur an der Wahl teilnimmt sonder auch von seiner Position als Leiter zurücktritt. Sonst könnte es theoretisch zu mehr als einem Leiter kommen. Diese Ergänzung ist allerdings nur nötig, da in diesem Programm eine Wahl nicht erst eingeleitet wird, wenn ein Leiter fehlt, sondern durch die Wünsche des Benutzer bereits früher.

3.3 Message Buffer

Der Nachrichtenpuffer bietet eine Möglichkeit Nachrichten von einem Arbeiter zum nächsten bzw. vom Ring-Controller zu einem Arbeiter zu senden. Er bietet zwei verschieden Möglichkeiten dafür an, eine Möglichkeit, bei der die Nachricht lediglich sobald als möglich in den Puffer gelegt wird und der Sender mit seiner Tätigkeit wieder weitermachen kann, und eine andere Möglichkeit, bei der zusätzlich gewartet wird, bis die Nachricht vom Gegenüber entnommen wurde oder zu viel Zeit verstrichen ist. Die Arbeiter verwenden letzteren Weg für die Kommunikation untereinander. Wenn die Wartezeit ausläuft bevor der Nachbar die Nachricht entnommen hat, wird angenommen, dass er ausgefallen ist.

```
lock_guard<mutex> assign_and_wait_lck{
    assign_and_wait_mtx
};

unique_lock<mutex> rendezvous_lck{rendezvous_mtx};
message_is_taken = false;
rendezvous_lck.unlock();

assign(message);

rendezvous_lck.lock();
return message_taken.wait_for(
    rendezvous_lck, chrono::milliseconds(waittime),
    [this](){ return message_is_taken; }
);
```

Listing 1: senden und warten

Bei der Funktion `assign_and_wait` bzw. *senden und warten* sieht man, dass durch den Lock gleich am Anfang nur ein Aufrufer auf einmal sich in dieser Funktion aufhalten darf. Das liegt daran, dass der Aufrufer, nachdem er die Nachricht mit `assign(message)` sendet, unten darauf wartet, dass die Nachricht entnommen wird. Da bei einer Condition Variable nicht zwingend der Thread aufgeweckt wird, der länger gewartet hat, ist es zu vermeiden, dass mehrere Aufrufer gleichzeitig warten bzw. zur Überprüfung, ob sie warten müssen, kommen, sonst könnte dem falschen Sender glaubhaft gemacht werden, dass seine Nachricht bereits entnommen wurde. Außerdem bestünde noch die Gefahr, dass ein Aufrufer `message_is_taken` auf falsch setzt kurz, nachdem es vom Empfänger auf wahr gesetzt wurde und noch bevor der wartende Aufrufer darauf reagieren konnte. Diese Probleme sind in diesem Programm zwar nicht zu erwarten, da es je Puffer eigentlich nur einen Thread gibt, der diese Funktion benutzt. Das ist der vorrangige Nachbar. Doch da dieser Lock und Mutex in dieser konkreten Implementierung eigentlich nicht benötigt werden, blockieren sie also auch niemanden.

```
unique_lock<mutex> buffer_lck{buffer_mtx};
message_assignable.wait(
    buffer_lck, [this](){ return is_empty(); }
);

this->message = message;
message_assigned = true;
message_takable.notify_one();
```

Listing 2: nur senden, ohne warten

Bei der Funktion `assign` bzw. *senden* sieht man, dass am Anfang gewartet wird, dass man etwas in den Puffer legen kann, und es danach in den Puffer gelegt wird. Außerdem kann man in dieser Funktion auch erkennen, dass es der Puffer nur eine Nachricht halten kann. Der Puffer ist lediglich eine primitive Variable. Die Entscheidung, einen Puffer der Größe 1 zu benutzen, wurde gemacht, weil einerseits erspart man sich damit ein größeres Objekt und bei der Benutzung stört es nicht wirklich, da es meistens nur einen Sender, den vorrangige Nachbarn, gibt und manchmal auch einen zweiten, den Ring-Controller, mehr aber nicht.

```
1  unique_lock<mutex> buffer_lck{buffer_mtx};
2  message_takable.wait(
3      buffer_lck, [this](){ return message_assigned; }
4  );
5
6  message_assigned = false;
7  message_assignable.notify_one();
8
9  lock_guard<mutex> rendezvous_lck{rendezvous_mtx};
10 message_is_taken = true;
11 message_taken.notify_one();
12
13 return message;
```

Listing 3: Nachricht holen

Die Funktion `take` bzw. *Nachricht holen* verwendet der Empfänger, um sich die Nachricht zu holen. Am Anfang wird überprüft, ob eine Nachricht vorhanden ist und wenn nicht wird gewartet. Die Nachricht wird dann entnommen und, falls Threads darauf warten, dass sie eine Nachricht in den Puffer legen können oder dass ihre Nachricht entnommen wurde, werden diese verständigt.

3.4 Message

Bei den Nachrichten selbst handelt es sich lediglich um Strukturen, die einen Nachrichtentyp und eventuell dazugehörige Daten enthalten.

4 Bedienung

Das Programm `ring_voting` bietet einen Ring mit zufällig generierten und einzigartigen Arbeitern, die den Chang-Roberts-Algorithmus ausführen.

Das Programm kann über die Kommandozeile bedient werden. Für die Konfiguration gibt es folgende Parameter:

Parametername	Datentyp	Standardwert	Beschreibung
`size`	positive ganze Zahl > 0		Anzahl der Arbeiter
`-h, --help`			Ausgabe der Hilfe
`-c, --config`	Dateiname		Liest die Konfiguration aus der angegebenen Datei aus

`-n,`
 	
`--number-of-elections`
	ganze Zahl $>= 0$	0	Führt die angegebene Anzahl an Wahlen durch bis es stoppt.
			Bei 0 stoppt es nicht.

Parametername	Datentyp	Standardwert	Beschreibung
`--sleep`	ganze Zahl >= 0	5.000	Die Wartezeit in Millisekunden nach jeder Wahl, bevor die nächste ausgeführt wird
`--worker-sleep`	ganze Zahl >= 0	500	Die Zeit in Millisekunden, die jeder Arbeiter nach einer erledigten Aufgabe wartet, bevor er die nächste ausführt
`--log`			Aktiviert Loggen, wenn nicht weiter spezifiziert ist das Log-Level INFO und die Ausgabe ist auf die Konsole
`--log-file`	Dateiname		Aktiviert Loggen und schreibt den Log in die angeführte Datei. Die Datei wird erstellt, wenn sie noch nicht existiert, ansonsten wird der Log am Ende angehängt.
`log-date`			Logt in die Datei nicht nur die Uhrzeit, sondern auch das Datum
`--log-level`	0..5	2 bzw. INFO	Setzt das Log-Level auf den angeführten Wert. 0 ist TRACE, 1 ist DEBUG, 2 ist INFO, 3 ist WARN, 4 ist ERROR und 5 ist CIRTICAL
`--no-config-log`			Die benutze Konfiguration wird nicht als DEBUG-Nahricht geloggt

Die Größe des Ringes muss entweder in der Kommandozeile oder in der Konfigurationsdatei angegeben werden.

Die Konfigurationsdatei hat die außer der Hilfsnachricht die selben konfigurierbaren Optionen, wie die Kommandozeile.

Literatur

[1] Borzoo Bonakdarpur. *Distributed Algorithms. Leader Election.* URL: http://web.cs.iastate.edu/~borzoo/teaching/15/CAS769/lectures/week4.pdf (besucht am 10. 01. 2021).

[2] Ernest Chang und Rosemary Roberts. „An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes“. In: *Commun. ACM* 22.5 (Mai 1979), S. 281–283. ISSN: 0001-0782. DOI: 10.1145/359104.359108. URL: https://doi.org/10.1145/359104.359108.

[3] Randolph Franklin. „On an Improved Algorithm for Decentralized Extrema Finding in Circular Configurations of Processors“. In: *Commun. ACM* 25.5 (Mai 1982), S. 336–337. ISSN: 0001-0782. DOI: 10.1145/358506.358517. URL: https://wrf.ecse.rpi.edu/p/14-cacm82-ring.pdf.