Simulation einer verteilten Synchronisation mit einem zentralen Koordinator

David Fischer, 5CHIF

Jänner 2021

Contents

1 Einleitung

1.1 Problematik

Laut Angabe war das Ziel dieser Aufgabe, eine Simulation einer verteilten Synchronisation mit einem zentalen Koordinator zu erstellen. Die Simulation soll mit einer beim Aufruf definierten Anzahl an Nodes gestartet werden.

Die folgenden Illustrationen beschreiben den Prozess, den ein Koordinator durchläuft, sobald eine Node den kritischen Abschnitt betreten will.

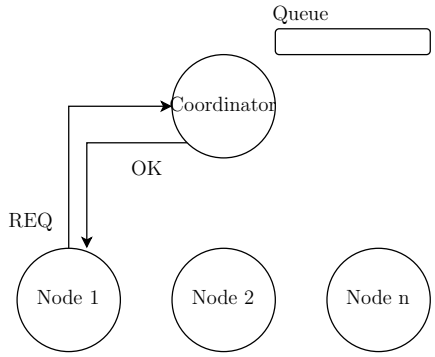

Figure 1: Node 1 sendet einen "Request" an den Koordinator. Da die Queue leer ist, bekommt Node 1 ein "OK" und betritt den kritischen Abschnitt.

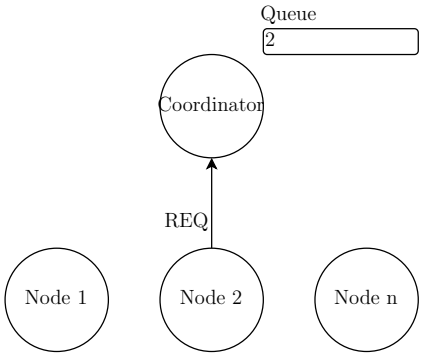

Figure 2: Node 2 sendet einen "Request" an den Koordinator. Da schon jemand im kritischen Abschnitt ist, wird Node 2 in die Queue gesetzt.

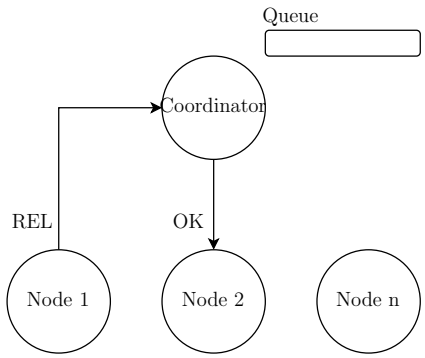

Figure 3: Node 1 verlässt den kritischen Abschnitt und sendet ein "Release" an den Koordinator. Der Koordinator sieht, dass Node 2 noch in der Queue steht, und sendet diesem damit ein "OK".

2 Implementierung

Für Variablen sowie Funktionen wurde "Snake case" benutzt. Sämtliche vorgegebene Codierungskonventionen wurden nach Möglichkeit eingehalten. Um sämtliche Teile des Programmes zu vereinfachen, wurden mehrere externe Bibliotheken verwendet, diese sind in Sektion 2.2 kurz beschrieben.

2.1 Kurze Beschreibung diverser Codeblöcke

2.1.1 Erstellen einer dynamischen Nummer an Nodes

Um, wie nach Angabe, eine dynamische Nummer an Workerthreads zu erstellen, wird ein Thread Vector erstellt. Dieser wird mit Threads der *operator*()() Funktion von Node Klassen befüllt. Genauer sieht man den Prozess in Snippet 1.

```
// create vector of threads to store the nodes operator()() threads
vector<thread> node_container;
node_container.resize(no_of_nodes);

for (int cnt = 0; cnt < no_of_nodes; cnt++) {
    Node tmp_node(cnt, ref(coord), opt);
    node_container.at(cnt) = thread{tmp_node};
}
```

```
// join threads from node_container
for( auto &t : node_container ) {
    t.join();
}
```

Source Code 1: Erstellen sowie Befüllen des node_container Vectors.

2.1.2 Anfragen von Workerthreads an den Koordinator

Um die Kommunikation zwischen Workerthreads und dem Koordinator zu ermöglichen, wird dem Konstruktor jedes Workerthreads eine Referenz auf den Koordinator mitgegeben. Alternativ ist es möglich, HTTP als Kommunikationsmedium zu benutzen, dies wird genauer beschrieben in Sektion 2.2.2.

2.1.3 Verarbeitung von REQ Anfragen an den Koordinator

Um zu signalisieren, dass ein Workerthread den kritischen Abschnitt verlassen hat, wird eine Bedingungsvariable benutzt. Dank dieser ließ sich die Verarbeitung von REQ und REL Anfragen einfach gestalten. Da der Funktionsaufruf den Rest des Ablaufs des Workerthreads blockiert, kann das Verlassen der Funktion als "OK" gewertet werden.

```
void Coordinator::message_req(int id){
    unique_lock<mutex> ul{mtx};

    node_queue.push(id);

    while(id != node_queue.front()){
        spot_taken.wait_for(ul, 1s);
    }

    spdlog::info("Coord : OK to Node {}", id);
}
```

Source Code 2: message_req() Funktion, die von Workerthreads aufgerufen wird.

2.1.4 Verarbeitung von REL Anfragen an den Koordinator

Sobald ein Workerthread den kritischen Abschnitt verlässt ruft dieser die *message_rel*() Funktion auf. Die Funktion vergleicht, ob die Node, die vorne in der Queue steht, gleicht mit der Node ist, die die REL Anfrage sendet. Damit ist eine minimale Fehlerverarbeitung gegeben.

```
void Coordinator::message_rel(int id){
    unique_lock<mutex> ul{mtx};

    int removed = node_queue.front();

    if(removed != id){
        spdlog::critical("Coord : REL Request from Node {}",
                         "didn't match current Node {}",
                          id, removed);
    }else{
        node_queue.pop();
    }

    spot_taken.notify_one();
}
```

Source Code 3: message_rel() Funktion, die von Workerthreads aufgerufen wird.

2.2 Externe Bibliotheken

2.2.1 CLI11

CLI11 [1] ermöglicht eine einfache Verarbeitung von Kommandozeilenargumenten mit eingebauten Methoden zum Überprüfen der angegebenen Werte. Auf diese Argumente wird in Sektion 3 näher eingegangen.

2.2.2 httplib

Eines der Kommandozeilenargumente erlaubt dem Nutzer, einen lokalen HTTP Server zu öffnen, um diesen als Kommunikationsmittel zwischen

dem Koordinator und den Nodes zu benutzen. Ein httplib [4] Server öffnet sich lokal auf port 5001 und stellt drei Routen zur Verfügung:

/req?node_id=x Um einen "Request" an den Koordinator zu senden, benutzen die Nodes diese Route. Sobald die Nodes eine Antwort mit dem Code 200 erhalten, wird das als "OK" gewertet.

/rel?node_id=x Um einen "Release" an den Koordinator zu senden, benutzen die Nodes diese Route.

/get Ist eine Route, die dem Nutzer eine Einsicht in das Programm ermöglicht, ohne Einsicht in die Logs zu haben. Eine Live-Demo dieser Funktionalität läuft seit längerem in einem VM Container, aufrufbar unter `https://s.konst.fish/fischer_projekt_1`.

2.2.3 tabulate

Sobald das Programm vom Nutzer mittels $Ctrl + C$ abgebrochen wird, wird eine Tabelle mittels tabulate [3] erstellt. Diese enthält diverse Informationen die während der Programmlaufzeit gesammelt wurden. Wenn die httplib Flag gesetzt ist, kann auf die Tabelle, wie in Paragraph 2.2.2 erwähnt, jederzeit zugegriffen werden.

```
+----------------------+--------------------+------------------------+
| No. of Admitted Nodes | Maximum Queue Size | Total Time Spent Running|
+----------------------+--------------------+------------------------+
| 536748               | 9                  | 2409006s               |
+----------------------+--------------------+------------------------+
```

Source Code 4: Beispiel einer tabulate Tabelle

2.2.4 spdlog

Um einfaches Loggen, parallel in der Konsole und in einem File, zu ermöglichen wurde die spdlog [2] Bibliothek verwendet.

```
sinks.push_back(file_sink);
sinks.push_back(console_sink);
auto combined_logger = make_shared<spdlog::logger>("CombSink",
                                                   begin(sinks),
                                                   end(sinks));
// register to access it globally
spdlog::register_logger(combined_logger);

spdlog::info("Beispiel Log Eintrag");
```

Source Code 5: Registrieren eines kombinierten Loggers, um global in diesem schreiben zu können.

3 Verwendung

3.1 Kommandozeilenargumente

3.1.1 Erforderlich

number Nummer an Workerthread, die erstellt werden soll. Standardmäßig limitiert auf 2 bis 200. Sollte die -r Flagge gesetzt sein, ist die Limitation von 2 auf 10 reduziert.

3.1.2 Optional

-o, –outage-simulation Lässt Workerthreads mit einer 3 % Chance pro Eintritt in den kritischen Abschnitt ausfallen, um einen Deadlock zu erzeugen.

-d, –outage-detection Nur zulässig, wenn –outage-simulation auch gesetzt ist. Lässt den Koordinator 8 Sekunden auf die Node warten. Wenn nach dieser Zeit keine REL Anfrage eingeht, wird die Node aus der Queue entfernt.

-r, –requests Lässt Kommunikation zwischen den Threads über HTTP laufen. Näher beschrieben in 2.2.2.

4 Projektstruktur

```
/
├── LICENSE
├── meson_options.txt
├── meson.build
├── README.md
├── .gitignore
├── include
│   ├── utils.h
│   ├── Node.h
│   └── Coordinator.h
├── src
│   ├── utils.cpp
│   ├── Node.cpp
│   ├── Coordinator.cpp
│   └── main.cpp
├── doc
│   ├── ausarbeitung.tex
│   ├── references.bib
│   └── ausarbeitung.pdf
└── build
```

References

[1] CLIUtils. Cli11. https://github.com/CLIUtils/CLI11, 2020.

[2] Gabi Melman. spdlog. https://github.com/gabime/spdlog, 2020.

[3] Pranav. tabulate. https://github.com/p-ranav/tabulate, 2020.

[4] yhirose. cpp-httplib. https://github.com/yhirose/cpp-httplib, 2020.