HTBLuVA Wiener Neustadt
Höhere Lehranstalt für Informatik

AUSBESSERUNGSARBEIT

Simple Raft-Simulator

Ausgeführt im Schuljahr 2019/20 von:

Recherche, Programmierung, Testing
Matthias Guzmits 5AHIF

Betreuer / Betreuerin:

none

Wiener Neustadt, am 24. April 2019/20

Abgabevermerk: Übernommen von:

Eidestattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die im Literaturverzeichnis angegeben Quellen und Hilfsmittel verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Wiener Neustadt am 24. April 2019/20

Verfasser / Verfasserinnen:

Matthias Guzmits

Inhaltsverzeichnis

Kapitel 1

Introduction

Author: Matthias Guzmits

Dieses Dokument beschäftigt sich im Allgemeinen mit dem Raft Consensus Algorithmus und seiner Implementation. Zusätzlich ist eine Beschreibung der Projektverwendung enthalten. Da die Installation von sobjectizer nicht trivial ist wird diese neben den verwendeten Funktionen ebenfalls beschrieben.

Kapitel 2

Installation von Sobjectizer

Sobjectizer ist ein Framework für C++ welches das Verwenden des Actor-Models vereinfachen soll. Das einbinden von diesem Framework in mein Projekt geschieht über eine Shared-Library. Um Sobjectizer jedoch verwenden zu können ist es erst einmmal notwendig die Libraries zu kompilieren.

2.1 Kompilieren von Sobjectizer

Für das kompilieren von Sobjectizer wird von den Herstellern Stiffstream CMake vorausgesetzt. Das builden sieht unter Linux folgendermaßen aus:

```
git clone https://github.com/stiffstream/sobjectizer
cd sobjectizer
mkdir cmake_build
cmake -DCMAKE_INSTALL_PREFIX=target -DCMAKE_BUILD_TYPE=Release ../dev
cmake --build . --config Release
cmake --build . --config Release --target install
```

Nun sollten sich in einem neu erstellten target-Folder zwei Ordner befinden. Einmal include und einmal lib. In dem lib-Folder befinden sich die kompilierte Static- und Dynamic-Library. Im include-Folder können alle notwendigen header-Files gefunden werden.

2.2 Setup zum Verwenden des Projekts

Um das Projekt verwenden zu können muss ersteinmal die richtige Arbeitsumgebung geschaffen werden. Im include-Verzeichnis des Projekts befindet sich eine vorkompilierte Version von Sobjectizer. Dafür muss folgendes gemacht werden:

- Kompilieren wie oben beschrieben oder vorkompiliertes Material verwenden
- Shared Library in /usr/lib verschieben
- Ordner so_5 in /usr/lib anlegen
- Den Ordner so_5 aus target/include in /usr/lib/so_5 verschieben

Meson kann leider mit der find_library-Funktion keine relativen Pfade verwenden weshalb dieser Umweg notwendig ist. Ein Pfad der bei jedem Linux-User gleich ist ist nunmal /usr/lib.

Kapitel 3

Benutzte SObjektizer-Funktionen

SObjectizer besitzt an sich eigentlich eine Dokumentation. Leider ist diese nicht sonderlich detailreich und in manchen Fällen auch nicht aufschlussreich. Der einzige Weg herauszufinden wie etwas funktioniert war es in einem Beispiel auszuprobieren.

3.1 Funktionen

3.1.1 so_5::launch(*[](so_5::environment_t&){}*)

Diese Funktion ist essentiell für die Verwaltung von Agents. Prinzipiell wird nur ein Argument übernommen. Dieses ist eine Funktion welche eine so_5::environment_t Instanz verwaltet. Auf Basis dieser Instanz werden die Agents angelegt und verwaltet. Erst wenn diese Umgebung beendet wird wird die launch-Funktion beendet.

3.1.2 introduce_coop(*[](so_5::coop_t&){}*)

introduce_coop() ist eine Member-Function von so_5::environment_t. Diese Funktion wird in der Dokumentation von SObjectizer nicht erklärt. Sie wird einfach als Code-Snippet als notwendiges Übel bereitgestellt. Der einzige Parameter ist wieder wie in so_5::launch() eine Funktion welche sich wiederum um das eigentliche Anlegen der Agents kümmert. Der Parameter der übergebenen Funktion ist ein so_5::coop_t Objekt.

3.1.3 make_agent<*Klasse*>(*Paramliste des Konstruktors der Klasse*)

Die verwendete Klasse muss immer von so_5::agent_t abgeleitet sein um make_agent verwenden zu können. Es wird ein Zeiger auf die Klasse angelegt. Der einfachste Weg das Resultat zu speichern ist "auto" zu verwenden.

3.1.4 so_direct_mbox()

Im Kontext zu meinem Programm ist dies die wichtigste Funktion. Mit ihr wird die Inbox des erstellten Agents zurückgeliefert. Mithilfe dieser können wie bereits erwähnt die einzelnen Agents miteinander Kommunizieren. Wenn der Agent mittels make_agent aufgerufen wurde ist es notwendig darauf zu achten, dass das erstellte Objekt ein Pointer ist.

3.1.5 so_5::send<*Klasse/Struct*>(so_5::mbox_t, *Paramliste des Konstruktors der Klasse*)

Der erste Parameter ist immer die Inbox des Agents den man ansprechen möchte. Alle weiteren Parameter werden dazu genutzt die Klasse zu bilden. Die Funktion kann sowohl innerhalb von launch als auch in diversen Agents verwendet werden. Mithilfe dieser Funktion wird die Kommunikation der Agents ermöglicht. Die Klasse wird als "Brief" verwendet.

3.1.6 on_enter(*Funktion*)

Diese Funktion kann in Kombination von States verwendet werden. Wenn der State eines Agent zum spezifizierten State geändert wird, wird sofort der Code aus der übergebenen Funktion ausgeführt. Das kann benutzt werden um Variablen zu initialisieren oder um anderen Agents mitzuteilen, dass sich dieser jetzt in jenem State befindet.

```
class test final : public so_5::agent_t{
public:

  void so_define_agent() override {

    off.on_enter([this]{
      cout << "Dieser Agent ist gerade inaktiv geworden!!!" << endl;
    });

    on.on_enter([this]{
      cout << "Dieser Agent ist gerade aktiv geworden!!!" << endl;
    });

  }

private:
  state_t server_state,
  on{substate_of(server_state)},
  off{initial_substate_of(server_state)};
}
```

Als Eklärung: "so_define_agent()" ist eine Funktion, mit welcher das Verhalten eines Agents definiert wird. Diese Funktion ist von der Parent-Class "agent_t" vererbt worden und muss überschrieben werden.

3.1.7 on_exit(*Funktion*)

Ist das Gegenstück zu "on_enter" und wird aufgerufen wenn der State verlassen wird und kann somit als Destruktor gesehen werden.

```
class test final : public so_5::agent_t{
public:

  void so_define_agent() override {

    off.on_exit([this]{
      cout << "Dieser Agent ist gerade dabei in den Dienst zu wechseln!!!" << endl;
    });

    on.on_exit([this]{
      cout << "Dieser Agent beendet gerade seinen Dienst!!!" << endl;
    });

  }

private:
  state_t server_state,
  on{substate_of(server_state)},
  off{initial_substate_of(server_state)};
}
```

3.1.8 event(*Funktion*)

Events bilden das Herzstück von SObjectizer-States. Wenn ein Element in der Inbox ankommt was mit dem Typ übereinstimmt, welches die übergebene Funktion als Parameter besitzt wird dieses Event aktiviert.

```
struct msg_container {
  int sender_id;
  string msg;
}

class test final : public so_5::agent_t{
public:

  void so_define_agent() override {

    server_state.event(&event_handler);

  }
```

```
private:
  state_t server_state,

  void event_handler(mhood_t<msg_container> msg_c) {
    cout << "Nachricht von " << msg_c->sender_id << ": " << msg_c->msg << endl;
  }
}
```

3.1.9 time_limit(*chrono::milliseconds*, *state_t*

Diese Funktion ist wieder den States vorbehalten. Sie steuert den Übergang nach einer gewissen Zeit von einem Zustand in den anderen. Wird time_limit innerhalb des definierten Zeitintervalls nochmals im selben State aufgerufen wird der vorhergehende Aufruf "aufgehoben"/ "aktualisiert".

```
class test final : public so_5::agent_t{
public:

  void so_define_agent() override {

    off.on_exit([this]{
      cout << "Dieser Agent ist gerade dabei in den Dienst zu wechseln!!!" << endl;
    });

    on.on_exit([this]{
      cout << "Dieser Agent beendet gerade seinen Dienst!!!" << endl;
    });

    //sorgt dafür das von State on nach 100ms in State off gewechselt wird
    on.time_limit(chrono::milliseconds{100}, off);

    //hier ist das Gegenteil der Fall -> Fertig ist der Ping-Pong-Agent
    off.time_limit(chrono::milliseconds{100}, on);

  }

private:
  state_t server_state,
  on{substate_of(server_state)},
  off{initial_substate_of(server_state)};
}
```

3.2 Typen/Klassen

3.2.1 so_5::environment_t

Stellt die Umgebung bereit in welcher die Agents agieren können.

3.2.2 so_5::coop_t

so_5::coop_t stellt den Kontext bereit, mithilfe dessen Agents erstellt werden können.

3.2.3 so_5::mbox_t

Dieser Typ ist die eigentliche Inbox/Adresse mithilfe der sich die Agents untereinander verständigen.

3.2.4 so_5::agent_t

Von dieser Klasse muss geerbt werden, wenn ein Agent erstellt werden soll.

3.2.5 so_5::signal_t

Ist ein vereinfachtes struct welches ohne Parameter abeschickt werden kann. Wie der Name bereits sagt können zwischen Agents Signale herumgeschickt werden, welche für Events als Indikator dienen können.

3.2.6 state_t

Das Actor-Modell für welches SObjectizer ein Framework ist kennt sogenannte States. Ein Agent verhält sich unterschiedlich je nach dem in welchem State er gerade ist. States können auf folgende weise erstellt werden:

```
state_t server_state{this};
```

Dabei ist State eine Member-Variable der erstellten Agent-Klasse. Um das Event-Handling zu vereinfachen können States auch von States "erben". Das kann mit "substate_of()" und "initial_substate_of()" erreicht werden.

```
state_t server_state{this},
on{substate_of(server_state)},
off{initial_substate_of(server_state)};
```

Der Unterschied zwischen diesen beiden Funktionen ist, dass bei "initial_substate_of()" der erstellte State derjenige ist, in welchem sich der Agent initial befindet.

Kapitel 4

Meine Raft-Implementation

4.1 Server

Wie es in Raft üblich ist bestehen meine "Server" aus drei States: leader, follower und candidate. zusätzlich gibt es einen vierten State, welcher den Ausfall eines Servers simulieren soll.

```
state_t server_state{this},
candidate{substate_of(server_state)},
follower{initial_substate_of(server_state)},
leader{substate_of(server_state)},
inactive{substate_of(server_state)};
```

Für den eigentlichen Nachrichtenaustausch wurden verschiedene Klassen als Container definiert.

4.1.1 ClientRequest

```
struct ClientRequest {
    so_5::mbox_t inbox;
    std::string cmd;
};
```

Der ClientRequest wird vom Client an einen Server geschickt. Dieser Request enthält die Information wohin der Response geschickt werden muss: "inbox". Für die Serverseite nicht weniger wichtig ist aber das Attribut "cmd", welches den Befehl enthält, welcher von diesem verarbeitet werden soll.

4.1.2 ServerResponse

```
struct ServerResponse {
    int status; // 0 or 1 if 0 then request answered by follower
```

```
        std::string resp;
        std::string leader;
    };
```

Der Serverresponse ist das Gegenstück zum ClientRequest. Der Server sendet diesen mit den Informationen wer den Response gesendet hat, was das Ergebnis der Verarbeitung ist und wer der derzeitige Leader des Clusters ist mit. Die Information über den Leader ist deswegen wichtig, da der Client beim ersten Request eine Anfrage an einen zufälligen Server im Cluster sendet, welcher dann den Client über den derzeitigen Leader informieren muss. Um zwischen dem Response eines Leaders und eines Followers zu unterscheiden gibt es das Attribut "status".

4.1.3 SetCluster

```
    struct SetCluster {
        std::unordered_map<std::string, so_5::mbox_t> mboxes;
    };
```

Diese Nachricht ist dafür zuständig die einzelnen Teilnehmer und den Client darüber zu informieren, welcher Server wie zu erreichen ist und stellt somit den Beginn des Algorithmus dar.

4.1.4 deactivate

```
    struct deactivate : public so_5::signal_t{};
```

"deactivate" ist ein Signal, welches den Server einfach nur in einen extrigen State "inactive" versetzt oder herausholt, je nach dem ob der server sich gerade in diesem befindet. Dieser State simuliert den Ausfall eines Servers.

4.1.5 RequestVote

```
    struct RequestVote {

        RequestVote(int term_, bool vote_granted_) : term{term_},
        vote_granted{vote_granted_} {}

        RequestVote(int term_, std::string candidate_id_, int lli_, int llt_)
        : term{term_}, candidate_id{candidate_id_}, last_log_index{lli_},
        last_log_term{llt_}, vote_granted{false} {}

        int term;
        std::string candidate_id;
        int last_log_index;
        int last_log_term;
```

```
    bool vote_granted;
};
```

Dieses struct ist so aufgebaut wie es im Raft-Paper spezifiziert ist. Diese Nachricht wird nur aus dem candidate-State verschickt. Wenn mehr als die hälfte der im Cluster befindlichen Server mit einem positiven "vote_granted" wird der Server vom candidate- in den leader-State versetzt.

4.1.6 AppendEntry

```
struct AppendEntry {

    AppendEntry(int term_, bool success_) : term{term_},
    success{success_} {};

    AppendEntry(int term_, std::string leader_id_, int prev_log_index_,
    int prev_log_term_, std::vector<std::tuple<int,std::string>> entries_,
    int leader_commit_) : term{term_}, leader_id{leader_id_},
    prev_log_index{prev_log_index_}, prev_log_term{prev_log_term_},
    entries{entries_}, leader_commit{leader_commit_} {}

    int term;
    std::string leader_id;
    int prev_log_index;
    int prev_log_term;
    std::vector<std::tuple<int,std::string>> entries;
    int leader_commit;
    bool success;
};
```

AppendEntry ist ebenfalls so aufgebaut wie es im Raft-Paper spezifiziert wurde. Da es aber Probleme bei der Umsetzung gab musste eine zusätzlicher Nachricht "AppendEntryResult" hinzugefügt werden. "AppendEntry" dient nicht nur dazu das Log des leaders zu verteilen sondern auch dazu die follower davon abzuhalten ein Timeout zu erfahren. Für diesen "Heartbeat" wird das Attribut "entries" leergelassen.

4.1.7 AppendEntryResult

```
struct AppendEntryResult {
    std::string follower_name;
    int term;
    bool success;
};
```

Wie bereits oben erwähnt gab es Probleme bei der implementierung der Antwort auf AppendEntry-Nachrichten. Deshalb wurde ein struct hinzugefügt, welches das Prozedere vereinfachen sollte.

4.1.8 Heartbeats

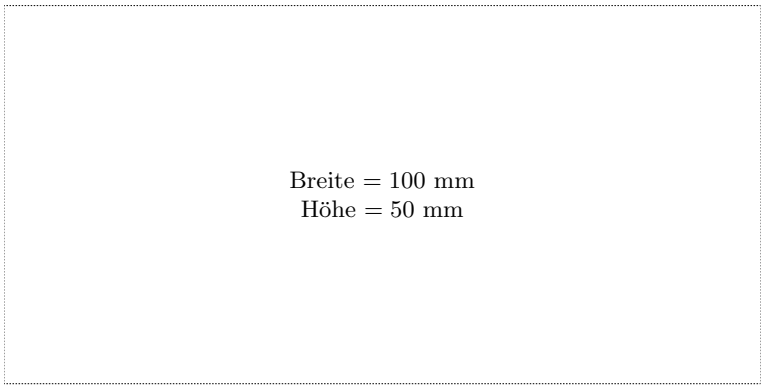
Heartbeats werden vom leader in einem Thread ausgeführt, welcher vom Hauptprogramm getrennt wird(detached). Mithilfe einer Variable kann die darin enthaltene Loop angehalten werden. In der Loop selbst werden in einem bestimmten Zeitraum immer wieder "AppendEntries" als Heartbeat ausgesand, damit die follower kein Timeout erleben.
Auf diese AppendEntries wird mit AppendEntryResults geantwortet. Das Attribut "success" gibt dabei an ob das Log des followers aktuell ist oder nicht.

4.2 Client

Der Client ist ein verkomplizierter Klon des LED-Beispiels in der SObjectizer-Dokumentation. Nach einer bestimmten Zeit wird zwishen zwei States gewechselt. In dem einen wird ein Request an das Cluster gesendet, wohingegen der andere State einfach nur eine Zeit lang wartet. Das Ergebnis welches vom Cluster erhaltern wird, wird derzeit einfach in der Konsole ausgegeben.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —