Publish/Subscribe Broker

Lorenz Killer

10. März 2019

Inhaltsverzeichnis

1 Lizenz

2 Themenbeschreibung

Unter einem Publish/Subscribe Broker versteht man einen Server, der das Lesen und Veröffentlichen von Nachrichten für beliebig viele Nutzer ermöglicht. Clients können dabei sogenannte *Topics* oder *Subjects* subskribieren um alle damit verbundenen Nachrichten zu erhalten oder Nachrichten zu beliebigen Topics veröffentlichen um sie subskribierten Nutzern zur Verfügung zu stellen.

Ein gängiges Protokoll, das für diese Art der Kommunikation oft verwendet wird, ist das *Message Queuing Telemetry Transport*, oder kurz *MQTT* Protokoll. Dieses Protokoll wird häufig im Kontext des *Internet of Things* angewendet, um beispielsweise verschiedensten Anwendungen Zugriff auf die Informationen von Sensoren zu geben. Der im Kontext dieses Projektes programmierte Publish/Subscribe Broker orientiert sich in einigen Aspekten an dem MQTT Protokoll, stellt jedoch keine korrekte Implementierung dieses dar.

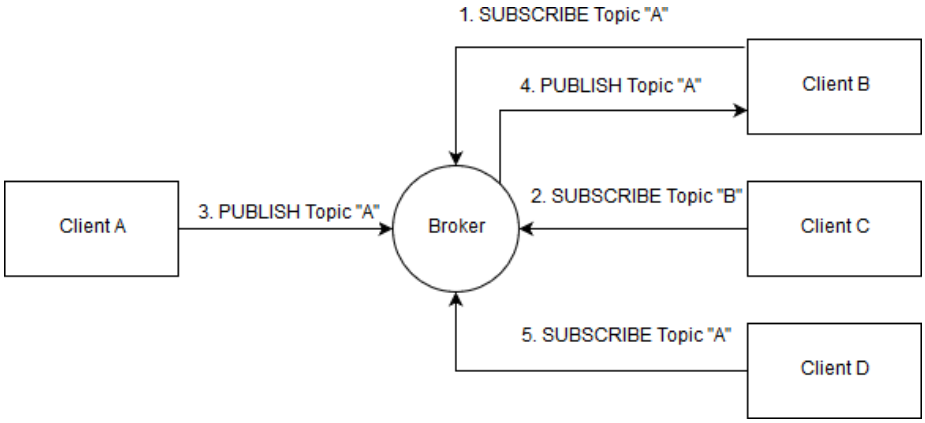

Abbildung 1: Darstellung eines theoretischen Publish/Subscribe Broker mit vier gleichzeitigen Clients. Die Zahlen vor den jeweiligen Nachrichten stehen für die Reihenfolge der Kommunikation.
Achtung! Diese Abbildung stellt kein korrektes Kommunikationsprotokoll dar, da in Wirklichkeit weitere Nachrichten geschickt werden müssten.
(Zum Beispiel: CONNECT, ACKNOWLEDGE, ...)

Obige Abbildung dient zur besseren Vorstellung des Themas. Client A veröffentlicht in diesem Szenario eine Nachricht zu dem Topic ,,A". Erkennbar ist, dass nur jener Client die Nachricht erhält, der zu dem Zeitpunkt zu dem der Broker die Nachricht von Client A erhält, das Topic ,,A" subskribiert hat (Client B).

3 Verwendung

3.1 Broker

3.1.1 Kommandozeilenparameter

Folgende Optionen können dem Broker beim Starten mitgegeben werden:

```
Usage: ./Broker [OPTIONS]

Options:
  -h,--help         Print this help message and exit
  -p,--port UINT    The Port the Broker will listen on      (Default:  6666)
  -n,--name TEXT    The Name of the Broker                  (Default: 'Broker')
  -c,--config TEXT  The Path to the JSON Configuration File (Default: None)
  -l,--log TEXT     The Path to the Logfile                 (Default: None)
```

- **-h, –help:** Zeigt Informationen zur Verwendung des Programms wie oben dargestellt an.
- **-p, –port:** Gibt den Port an, auf dem der Broker lauscht. Wird diese Option nicht angegeben, so wird der Port 6666 verwendet.
- **-n, –name:** Der Name des Brokers. Dieser wird lediglich beim Loggen verwendet.
- **-c, –config:** Definiert den Pfad zu einer JSON Konfigurationsdatei, die die Topics festlegt, welche auf dem Broker zum Subskribieren und Veröffentlichen von Nachrichten zur Verfügung stehen. Wird diese Option nicht angegeben, so werden die Topics wie in MQTT dynamisch generiert.
- **-l, –log:** Definiert den Pfad zu einem Logfile, in dem die Nachrichten zu allen Topics gespeichert werden. Wird diese Option nicht angegeben, so werden diese Informationen lediglich auf der Kommandozeile geloggt.

Simpelster Aufruf eines Brokers:

```
./Broker
```

Beispielaufruf eines Brokers mit allen Optionen:

```
./Broker -p 8888 -n SmartHome -c ./config.json -l ./log.txt
```

3.1.2 JSON Konfigurationsdatei

Die Konfigurationsdatei für den Broker dient lediglich zur Definition der Topics, die zur Verfügung stehen sollen. Aus diesem Grund besteht das JSON Dokument lediglich aus einem Array, welches eine beliebige Anzahl an Strings enthält. Ein Beispiel für solch ein Dokument wäre folgender Ausdruck:

```
[
"house/kitchen/lights/main",
"house/kitchen/ligths/counter",
"house/kitchen/temperature",
"house/bedroom/lights/main",
"house/bedroom/lights/bed",
"house/bedroom/temperature"
]
```

3.1.3 Topics

Topics *können* wie im vorherigen Abschnitt hierarchisch angeordnet sein, *müssen* dies aber nicht. ,,kitchen-light" oder ,,kitchen-temperature" wären demnach ebenfalls gültige Topics.

Die beiden Wildcards ,,+" und ,,#" haben die selbe Bedeutung wie in MQTT und können im Zuge eines SUBSCRIBE oder UNSUBSCRIBE Requests, nicht jedoch eines PUBLISH Requests oder in der Konfigurationsdatei verwendet werden.

3.2 Client

3.2.1 Kommandozeilenparameter

Folgende Optionen können dem Client beim Starten mitgegeben werden:

```
Usage: ./Client [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  -s,--server TEXT            The Hostname the Client will connect to (Default:  'localhost')
  -p,--port UINT              The Port the Client will connect to     (Default:  6666)
  -n,--name TEXT              The Name of the Client                  (Default: 'Client')
  -c,--config TEXT REQUIRED   The Path to the Config File             (Required)
```

- **-h, –help:** Zeigt Informationen zur Verwendung des Programms wie oben dargestellt an.
- **-s, –server:** Gibt den Hostname an, zu dem sich der Client verbindet. Wird diese Option nicht angegeben, so wird 'localhost' verwendet.
- **-p, –port:** Gibt den Port an, zu dem sich der Client verbindet. Wird diese Option nicht angegeben, so wird der der Port 6666 verwendet.

- **-n, –name:** Der Name des Clients. Dieser wird lediglich beim Loggen verwendet.
- **-c, –config:** Definiert den Pfad zu einer JSON Konfigurationsdatei, die die Aktionen enthält, die der Client durchführen soll. Diese Option muss zwingend angegeben werden.

Simpelster Aufruf eines Clients:

```
./Client -c ./config.json
```

Beispielaufruf eines Clients mit allen Optionen:

```
./Client -s 8.8.8.8 -p 8888 -n DoorWatcher -c ./config.json
```

3.2.2 JSON Konfigurationsdatei

Die Konfigurationsdatei des Clients dient dazu, dem Client Anweisungen zu übergeben die dieser ausführen soll. Dies dient hauptsächlich dem Testen, da so das Verhalten der Clients schnell angepasst werden kann, ohne jedes mal die Implementierung zu ändern.

Der Aufbau des JSON Dokuments kann folgenderweise beschrieben werden.

- **"keep_alive":** Boolean; Default: True
  Bestimmt ob der Client nach der Ausführung der Kommandos weiterläuft.
- **"actions":** Array; Required
  Beinhaltet die "Blöcke"von Befehlen, die auszuführen sind.
  Ein Block besteht aus:
  - **"repeat":** Integer >= 0; Default: 1
    Bestimmt wie oft die folgenden Befehle ausgeführt werden.
  - **"delay_between":** Integer >= 0; Default: 0
    Gibt eine Verzögerung in Millisekunden zwischen den Befehlen an.
  - **"delay_after":** Integer >= 0; Default: 0
    Gibt eine Verzögerung in Millisekunden an nachdem die Befehle ausgeführt wurden.
  - **"commands":** Array; Required
    Beinhaltet die Befehle dieses Blocks.
    Ein Befehl besteht aus:
    - **"type"** String; Required
      Bestimmt den Typ des Befehls.
      Gültige Werte sind ,,SUBSCRIBE", ,,UNSUBSCRIBE" und ,,PUBLISH"
    - **"topic"** String; Required
      Das Topic des Befehls.
    - **"content"** String; Default: ""
      Dieses Feld wird nur gelesen, wenn der Typ ,,PUBLISH" entspricht. Enthält den Inhalt der veröffentlicht werden soll.

Folgender Ausdruck stellt eine gültige Konfigurationsdatei dar. Ein Client mit dieser Konfiguration wird alle Topics, die in der Hierarchie unter dem Topic ,,house/kitchen" liegen subskribieren. Anschließend wird er für eine Minute warten, die für die selben Topics einen ,,UNSUBSCRIBE" Request senden und sich anschließend beenden.

```
{
    "keep_alive": false,
    "actions": [
        { "delay_after": 60000,
          "commands": [ { "type": "SUBSCRIBE", "topic": "house/
    kitchen/#"} ]
        },
        { "commands": [ { "type": "UNSUBSCRIBE", "topic": "house/
    kitchen/#"} ] }
    ]
}
```

4 Das Programm

Dieses Kapitel dient der genaueren Beschreibung des Aufbaus und Ablaufs Programmes.

4.1 UML-Diagramm

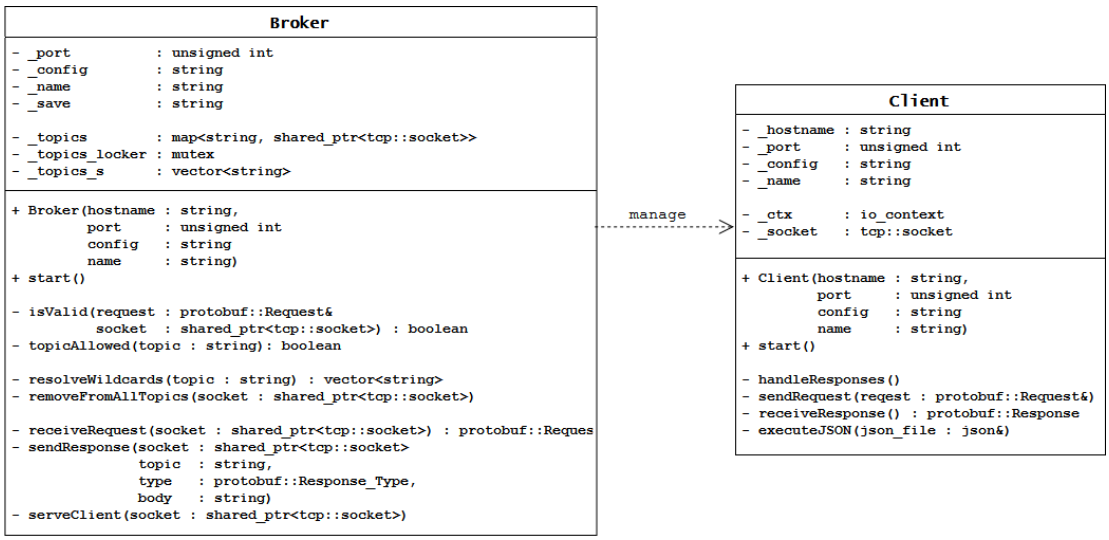

Abbildung 2: UML-Diagramm des Programms. Ein Broker verwaltet n Clients zur selben Zeit über Asio-TCP-Sockets.

4.2 Broker

Der Broker bekommt beim Erstellen über den Konstruktor die Kommandozeilenparameter übermittelt. Anschließend wird dieser durch die Methode „start" gestartet. Dabei wird wenn angegeben die Konfigurationsdatei gelesen und ein File-Logger erstellt.
Folgender Code dient dem Einlesen der verfügbaren Topics aus der Konfigurationsdatei.

```
try {
    json json_file = json::parse(ifs);

    for (json& topic : json_file){
        _topics_s.push_back(topic);
    }
} catch (json::parse_error& e){
    spdlog::get(_name)->error("Could not find or parse the JSON
    Configuration File!");
    return;
}
```

Anschließend wird in einer Endlosschleife für jeden Client der sich verbindet ein Socket erstellt und dieser in einen Thread verschoben, der sich mithilfe der Funktion „serveClient" um die Requests des Clients kümmern wird. Folgender Codeausschnitt stellt diese Schleife dar.

```
while (true){

    shared_socket sock{make_shared<tcp::socket>(ctx)};

    acceptor.accept(*sock);  // Blocking !!!

    spdlog::get(_name)->info("A new Client connected");

    thread t{&Broker::serveClient, this, move(sock)};
    t.detach();
}
```

Die Funktion „serveClient" ist wieder eine Endlosschleife die einen blockierenden Aufruf startet, der auf einen Request des Clients wartet. Jeder Request wird vorerst auf Fehlerhaftigkeit überprüft, beispielsweise ob ein ungültiger Typ oder ein ungültiges Topic angegeben wurde.
Bei einem Fehlerhaften Request wird der Client dementsprechend benachrichtigt. Ist der Request nicht gültig, so werden je nach Request-Typ die passenden Aktionen durchgeführt und ein Response gesendet.
Folgende Möglichkeiten gibt es dabei:

- **SUBSCRIBE:** Da bei einem SUBSCRIBE-Request Wildcards erlaubt sind, werden diese - falls vorhanden - zuerst ausgewertet.
  Passt das Topic mit Wildcards zu keinem einzigen Topic, so wird dies dem Client anhand eines Responses mitgeteilt.
  Andernfalls wird der Client zu bei jeden passenden Topic als „subscribed" hinzugefügt und bekommt für jedes Topic einen Response.

- **UNSUSCRIBE:** Auch hier wird das Topic des Requests zuerst auf Wildcards durchsucht und der Client informiert, falls keine Topics darauf zutreffen.
  Anschließend wird der Client von jedem der Topics entfernt und bekommt dementsprechend jeweils einen Response. Beinhaltet der Request ein Topic zu dem der Client gar nicht ,,subscribed" ist oder das nicht existiert, so bekommt dieser dennoch einen positiven Request, da das Endergebnis das Selbe ist.

- **PUBLISH:** Bei einem PUBLISH-Request werden alle Clients die bei dem Topic subskribiert sind mit einem UPDATE-Response benachrichtigt. Des weiteren wird die Nachricht - sofern angegeben - in einem Logfile gespeichert und der Client der den Request abgeschickt hat benachrichtigt.

Beendet sich ein Client zu einem beliebigen Zeitpunkt oder schlägt die Kommunikation fehl, so wird dies geloggt und der Client wird von allen Topics entfernt.

4.3 Client

Ebenso wie beim Broker, werden auch beim Client die Kommandozeilenparameter über den Konstruktor übergeben. Genauso wird dieser danach über die Funktion ,,start" gestartet. In der Funktion wird die Verbindung zum Server erstellt und die Konfigurationsdatei gelesen. Sollte dabei ein Fehler auftreten so wird der Benutzer informiert und das Programm beendet.

Es wird außerdem ein Thread gestartet, welcher in einer Endlosschleife die Responses des Servers empfangen wird und dementsprechende Informationen für den Benutzer loggt. Als nächstes werden die Kommandos aus der JSON-Konfigurationsdatei der Reihe nach ausgeführt indem entsprechende Requests an den Server gesendet werden.

Wurden alle Befehle abgearbeitet, kommt es auf die Option ,,keep_alive" aus der Konfigurationsdatei an. Hat diese den Wert ,,true", so wird der Thread, der für die Responses zuständig ist ,,gejoined" um das Programm am Laufen zu halten und weiterhin Updates von subskribierten Topics zu bekommen. Ist der Wert der Option jedoch ,,false", wird das Programm beendet.