Message Broker

Seminar Concurrent Programming in C

an der ZHAW am Standort Zürich

Reto Hablützel

10. Juni 2014

Inhaltsverzeichnis

1 Einleitung

Ein Message Broker ist eine zentrale Kommunikationsschnittstelle für mehrere Programme, die in verschiedenen Programmiersprachen implementiert sein können. Nebst der Möglichkeit heterogene Systeme zu verbinden, kann jedoch durch einen Message Broker auch eine verteilte Architektur realisiert werden, indem jeder Client einen Teil der Funktionalität übernimmt.

Damit verschiedenste Programme miteinander kommunizieren können, bieten Message Broker typischerweise Schnittstellen mit verschiedenen Protokollen. Eines davon ist STOMP[1], welches sehr einfach gehalten und spezifisch für dynamische Programmiersprachen entwickelt wurde. Im Rahmen dieses Seminars habe ich mich deshalb für die Implementation von einem Message Broker entschieden. Da das Kernthema Concurrency ist und ich nicht zu viel Zeit mit dem Protokoll verschwenden wollte, habe ich eine abgespeckte Version von STOMP als Protokoll implementiert.

Auf den Umfang der Funktionalität wird im Kapitel 2 näher eingegangen. Das Kapitel 3 beschreibt im wesentlichen das Subset von STOMP, das implementiert wurde, die verschiedenen Komponenten, die im Message Broker existieren, sowie die Speicherstruktur. Im Kapitel 4 finden sich einige Details zur Implementation (insbesondere Locking) gefolgt vom Kapitel 5, welches die Verwendung vom Message Broker erklärt. Abgeschlossen wird die Dokumentation mit einem persönlichen Rückblick im Kapitel 6.

2 Funktionalität

2.1 Verbindungsauf- und abbau

Jede Verbindung zwischen einem Client und dem Message Broker beginnt mit einem kurzen Handshake. Der Client teilt dem Message Broker seinen Namen mit und der Message Broker bestätigt den Empfang. Sobald der Client die Kommunikation beenden will, teilt er dies dem Message Broker mit einer Disconnect Nachricht mit und kriegt wiederum eine Bestätigung.

2.2 Nachrichten empfangen

Ein verbundener Client kann einen Topic abonnieren und kriegt so vom Message Broker sämtliche Nachrichten zu diesem Topic zugestellt. Topics werden dynamisch erstellt, sobald sicher der erste Client dafür interessiert. Ein Client, der Nachrichten von einem Topic abonniert hat, wird Subscriber genannt.

2.3 Nachrichten senden

Ein verbundener Client kann Nachrichten zu einem bestimmten Topic an den Message Broker senden. Ein solcher Client wird als Publisher bezeichnet.

[1] https://stomp.github.io/

3 Konzept

3.1 Protokoll

Für die Umsetzung wurde eine abgespeckte Version vom STOMP Protokoll implementiert. Die implementierten Kommandos werden in den folgenden Kapiteln aufgelistet. Auf sämtliche Kommandos vom Client kann der Server mit ERROR antworten, falls etwas schief gelaufen ist. Jedes Kommando hat die folgende Struktur:

Listing 1: Struktur eines Kommandos

```
COMMAND
key: value
key: value
...

Content

^@
```

Auf der ersten Zeile steht der Name des Kommandos in Grossbuchstaben. Auf folgenden Zeilen stehen durch Doppelpunkt separierte Key-Value Paare, welche Header Werte darstellen. Nach den Header Werten folgt der Content, welcher durch eine Leerzeile vom Header getrennt ist. Das Ende eines Kommandos markiert wiederum eine Leerzeile gefolgt vom Null-Byte (hier ^@):

3.1.1 CONNECT

Wird vom Client an den Message Broker zum Beginn der Verbindung gesendet. Hat einen zwingenden Header 'login', welcher den Client identifiziert. Hat keinen Content. Sobald der Message Broker das Kommando den Client aufgenommen hat, sendet er zur Bestätigung CONNECTED.

3.1.2 CONNECTED

Wird vom Message Broker als Antwort auf das CONNECT Kommando versendet falls die Verbindung erfolgreich erstellt werden konnte. Hat weder Header noch Content.

3.1.3 ERROR

Wird vom Message Broker in verschiedenen Szenarien an den Client gesendet um einen Fehler mitzuteilen. Hat einen zwingenden Header 'message', der die Fehlermeldung enthält. Hat keinen Content.

3.1.4 SEND

Wird von einem Client an den Message Broker gesendet um einen Nachricht zuzustellen. Hat einen zwingenden Header 'topic', der den Topic identifiziert, an den die Nachricht gesendet wird. Im Content steht der eigentliche Inhalt der Nachricht.

3.1.5 SUBSCRIBE

Wird von einem Client an den Message Broker gesendet um einen Topic zu abonnieren. Hat einen zwingenden Header 'destination', welcher den Topic identifiziert. Hat keinen Content.

3.1.6 MESSAGE

Wird vom Message Broker an den Client gesendet um eine Nachricht zuzustellen. Hat einen zwingenden Header 'destination', der den Topic identifiziert. Die Nachricht ist im Content.

3.1.7 DISCONNECT

Wird vom Client an den Message Broker gesendet um die Verbindung zu beenden. Hat weder Header noch Content.

3.1.8 RECEIPT

Wird vom Message Broker an den Client gesendet, sobald der Message Broker den DISCONNECT akzeptiert hat. Hat weder Header noch Content.

3.2 Speicherstruktur

Auf dem Server existieren zwei globale Listen: Die eine enthält alle Topics verknüpft mit einer Liste von Subscribern zum jeweiligen Topic. Die zweite Liste enthält alle Nachrichten, die vom Message Broker empfangen wurden. Zudem enthält jede Nachricht eine Liste von Statistiken. Eine Statistik ist eine Tripel aus Subscriber, Anzahl versuchter Zustellungen und einen Timestamp der letzten erfolglosen Zustellung. Diese wird verwendet um zu bestimmen, welche Subscriber die Nachricht erhalten haben und ob ein erneuter Versuch der Zustellung gemacht werden soll.

Listing 2: Speicherstruktur

```
Topics:
  - topic1
      - subscriber1
      - subscriber2
  - topic2
      - subscriber1
      - subscriber3
      ..
  ..

Messages:
  - message1
      - topic
      - content
      - statistic1
          - last_fail
          - nattempts
          - subscriber1
      - statistic2
          - last_fail
          - nattempts
          - subscriber2
  - message2
      ..
```

3.2.1 Einfügen einer neuen Nachricht

Wenn ein Client eine Nachricht an den Message Broker sendet, kopiert er die Liste von Subscribern von dem Topic, an den die Nachricht versendet wurde. Dann wird ein neuer Eintrag in der Liste der Nachrichten erstellt und die Subscriber werden in eine Liste von Statistiken überführt.

3.3 Komponenten

3.3.1 Übersicht

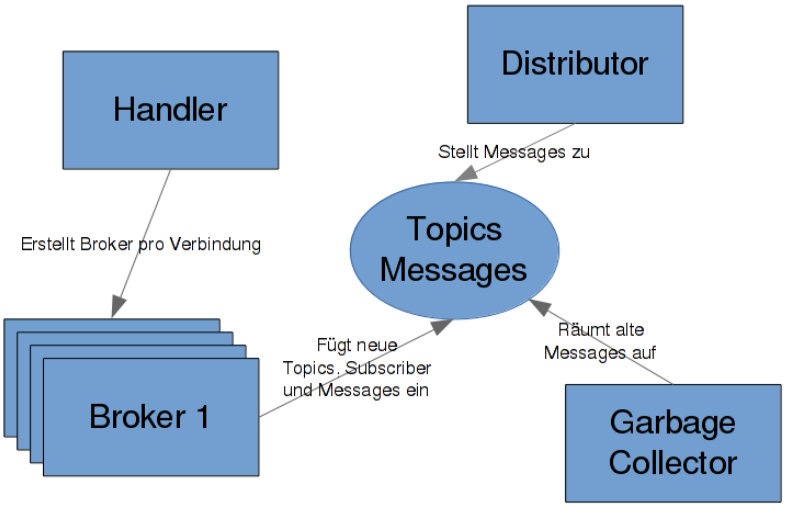

Abbildung 1: Übersicht der Komponenten

3.3.2 Handler

Der Handler erstellt pro Socket - Verbindung einen neuen Thread. Dies ist der Broker.

3.3.3 Broker

Pro verbundenem Client existiert ein Thread, der den Broker repräsentiert. Der Broker hält eine TCP Verbindung aufrecht und ist stets bereit, Kommandos zu lesen. Der Broker fügt den Client zur Liste der Subscriber, wenn er sich für einen Topic interessiert, nimmt eine Nachricht entgegen, wenn er eine sendet und reiht diese in die Liste der Nachrichten.

3.3.4 Distributor

Der Distributor prüft ständig die Liste von Nachrichten nach denjenigen, die zugestellt werden sollen. Jedes Mal, wenn eine Nachricht erfolgreich zugestellt wurde, wird den Zähler 'nattempts' um eins erhöht. Falls die Zustellung fehlschlägt, wird der Timestamp 'last_fail' gesetzt. Mit diesem Timestamp kann der Distributor auch feststellen, ob er bereits einen erneuten Versuch zur unternehmen soll.

3.3.5 Garbage Collector

Der Garbage Collector räumt Statistiken, Nachrichten und Subscriber auf. Mittels bestimmter Regeln entscheidet der Garbage Collector, ob ein Eintrag in der Liste der Statistiken zu einer Nachricht gelöscht werden kann (z.B. Nachricht wurde erfolgreich zugestellt oder Subscriber hat die Verbindung beendet). Sobald eine Nachricht keine Statistiken mehr hat, kann die gesamte Nachricht abgeräumt werden und aus der Liste der Nachrichten gelöscht werden. Wenn ein Subscriber in keiner der Liste der Statistiken mehr auftaucht, kann auch er gelöscht werden.

4 Implementation

4.1 Lock Implementation vs. Globaler Lock

Nach Aufgabenstellung ist es verboten, einen globalen Lock zu verwenden. Aus diesem Grund haben ich mich entschieden, die einzelnen Listen und deren Einträge mit verschiedenen Read- bzw. Write Locks zu schützen.

Die folgenden Tabellen zeigen pro Komponente links die Resourcen, die geschützt werden müssen und in den Spalten 2-n die Aktionen, die diese Resourcen verändern. Mit R, W und M wird dann angedeutet, welcher Lock für eine bestimmte Aktion gehalten werden muss.

- R: Es wurde ein Read Write Lock verwendet und dessen Read Lock muss gehalten werden
- W: Es wurde ein Read Write Lock verwendet und dessen Write Lock muss gehalten werden
- M: Es wurde eine Mutex verwendet, welche gehalten werden muss

Dabei ist es wichtig, dass die Locks top-to-bottom geholt werden. Wenn also eine Aktion zwei oder mehrere Locks für eine Aktion benötigt, muss zuerst der Lock gehalten werden, der oben in der Liste ist. Würde nämlich ein Thread die Locks von oben halten wollen und ein anderer von unten, könnte ein Deadlock auftreten.

4.1.1 Broker

Resource	Topic erstellen	Subscriber hinzufügen	Nachricht hinzufügen
Topics Liste	W	R	R
Subscriber Liste		W	R
Messages Liste			W
Statistik Liste			
Statistik			

4.1.2 Garbage Collector

Der Garbage Collector löscht Statistiken und Nachrichten nicht direkt, sondern sammelt zu löschende Elemente in einem ersten Schritt in einer Liste und löscht in sie in einem zweiten Durchlauf. Dadurch spart er sich einen Write Lock - welcher exklusiv ist - wenn es nichts zu löschen gibt.

Resource	Topic erstellen	Subscriber hinzufügen	Subscriber entfernen	Statistik löschen prüfen	Statistik löschen	Nachricht löschen prüfen	Nachricht löschen
Topics Liste	W	R / W *	R				
Subscriber Liste		W	W				
Messages Liste				R	R	R	W
Statistik Liste				R	W	R	
Statistik				R			

* Wenn ein Subscriber hinzugefügt werden soll, wird zuerst mit einem Read Lock auf der Liste der Topics geprüft, ob der Topic bereits existiert. Wenn der Topic noch nicht existiert, wird der Write Lock auf der Liste der Topics angefordert und die Existenz wird nochmal geprüft. Falls er noch immer nicht existiert, wird der Topic angelegt.

4.1.3 Distributor

Ähnlich wie der Garbage Collector sammelt der Distributor Nachrichten in einer Liste, welche zugestellt werden können. In einem zweiten Schritt werden diese zugestellt und die zugehörige Statistik wird aktualisiert.

Resource	Nachricht zustellen prüfen	Nachricht zustellen
Messages Liste	R	R
Statistik Liste	R	R
Statstik	R	W

4.1.4 Subscriber / Client

Nebste den diversen Locks für die beiden Listen, existieren noch drei Mutexe pro Client. Je eine wird für das Lese- und Schreib-Ende vom Socket verwendet und die dritte für ein Dead-Flag, welches auf 1 gesetzt wird, falls der Client nicht mehr erreichbar ist. Dies ist für den Garbage Collector auch ein Indikator, dass der Subscriber abgeräumt werden kann, sobald er auch in keiner Statistik mehr auftaucht.

4.2 Dateien

Sämtlicher Produktivcode ist im Verzeichnis src. Es folgt eine Auflistung der Dateien und eine kurze Beschreibung von deren Inhalt.

- server.c: Beinhaltet die main-Funktion vom Message Broker. Hier werden alle Komponenten gestartet und es wird auf neue Clients gewartet.
- broker.c/broker.h: Funktionalität vom Broker. Die Funktion handle_client ist Einstiegspunkt für den Broker pro Client.
- distributor.c/distributor.h: Funktionalität vom Distributor. Die Funktion distributor_main_loop wird dem Distributor - Thread als Startfunktion übergeben und versucht mittels der anderen Funktionen in der Datei die Nachrichten zuzustellen.
- gc.c/gc.h: Funktionalität vom Garbage Collector. Die Funktion gc_main_loop wird dem Garbage Collector - Thread als Startfunktion übergeben und führt die Garbage Collector Funktion kontinuierlich aus.
- list.c/list.h: Eine Implementation einer verketteten Liste und einigen Operationen. Diese Liste wird überall verwendet (z.B. Liste von Topics).
- socket.c/socket.h: Ein High-Level Interface für Sockets. Im Gegensatz zu herkömmlichen Funktionen, die mit Byte-Array Buffern über einen Socket Daten versenden, kann mit diesen Funktionen komfortabel ein ganzes STOMP - Kommando über den Socket versendet bzw. empfangen werden.
- stomp.c/stomp.h: Beinhaltet Funktionen zum Parsen von Char - Arrays in ein STOMP - Kommando (struct) und wieder zurück. Diese Funktionen werden verwendet, um Kommandos über den Socket zu versenden.
- topic.c/topic.h: Beinhaltet die zentralen Strukturen und Funktionen zum operieren mit den beiden zentralen Listen von Nachrichten und Topics. So gibt es zum Beispiel eine Funktion um einen Subscriber einem Topic hinzuzufügen.

4.3 Testing

Das ganze Programm wurde testgetrieben entwickelt und mit den Coverage Tools von GCC wurde kontinuierlich die Testabdeckung überwacht. Somit konnte schlussendlich eine hohe Abdeckung von über 97% erreicht werden. Als Test-Framework wurde cunit[2] verwendet.

[2] http://www.cunit.sourceforge.net

Listing 3: Testabdeckung

```
Summary coverage rate:
  lines.....: 97.9% (967 of 988 lines)
  functions..: 97.2% (69 of 71 functions)
```

4.3.1 Beispiel Testcase

Dass in unixoiden System nahezu alles ein Filedeskriptor ist, hat sich beim Testen als sehr hilfreich herausgestellt. So konnte ich Funktionen, die normalerweise Daten über einen Socket versenden, mittels Pipes testen. Pipes haben jedoch nur ein Lese- und ein Schreibende, weshalb ich für gewisse Tests UNIX Domain Sockets verwendet habe. Mit letzteren kann man nämlich an beiden Enden schreiben und lesen.

Folgendes Beispiel zeigt den Test, der zuerst zu viele Zeichen in die Pipe schreibt und dann erwartet, dass die Funktion socket_read_command den Fehler SOCKET_TOO_MUCH zurückgibt.

Listing 4: Testcase mit Pipe

```
char rawcmd[1025];
memset(rawcmd, 'a', 1024);
rawcmd[1024] = '\0';
pipe(fds);
client.sockfd = fds[0];
write(fds[1], rawcmd, 1025);
ret = socket_read_command(&client, &cmd);
CU_ASSERT_EQUAL_FATAL(SOCKET_TOO_MUCH, ret);
```

5 Verwendung

Dieses Kapitel beschreibt die Anwendung des Programms. Der Hauptteil ist der Message Broker selbst, welcher grundsätzlich durch das oben beschriebene Protokoll verwendet werden kann. Im Rahmen dieser Seminararbeit wurde jedoch auch ein Testclient implementiert, der die Funktionalität demonstriert.

5.1 Kompilieren und Tests ausführen

Als Buildsystem wurde make verwendet und die Tests basieren auf cunit. Letzteres muss installiert sein, um die Tests auszuführen.

Um den Code für die Tests zu kompilieren muss folgender Befehl ausgeführt werden. Dieser wird die Tests auch direkt ausführen und einen Report ausgeben.

Listing 5: Tests kompilieren und ausführen

```
message-broker$ make test
```

Der Produktivcode wird mit folgendem Befehl erzeugt.

Listing 6: Produktivecode kompilieren

```
message-broker$ make
```

Dies erstellt die beiden Binaries run und test im Hauptverzeichnis.

5.2 Message Broker starten

Mit folgendem Befehl kann der Message Broker auf dem Standard Port gestartet werden. Welcher das ist, wird auf der Konsole ausgegeben.

Listing 7: Message Broker auf Standard Port

```
message-broker$ ./run
```

Falls ein anderer Port verwendet werden soll, kann dieser als Argument mitgegeben werden.

Listing 8: Message Broker mit bestimmten Port

```
message-broker$ ./run 44554
```

5.3 Testclient starten

Auch der Testclient kann ohne Argumente aufgerufen werden und wird sich dann mit dem Standard Port vom Message Broker verbinden.

Listing 9: Testclient mit Standard Argumenten

```
message-broker$ ./test
```

Zusätzlich bietet der Testclient einige Argumente um das Verhalten zu Steuern. Wichtig ist, dass alle Argumente in dieser Reihenfolge angegeben werden.

- hostname: Der Hostname vom Message Broker
- port: Der Port vom Message Broker
- name: Der Name, der für das Login verwendet werden soll.

- nmsgs: Die Anzahl Nachrichten, die an den Broker versendet werden sollen.

Listing 10: Testclient mit bestimmten Argumenten

```
message-broker$ ./test localhost 44554 hmuster 33
```

6 Rückblick

6.1 Test Driven Development

Ich wollte dieses Projekt verwenden, um die testgetriebene Entwicklung auszuprobieren. Deshalb habe ich schon von beginn an immer zuerst Unit Tests geschrieben. Zugegebenermassen habe ich jedoch bei den ersten Funktionen (vor allem stomp.c) zuerst die Header-Datei geschrieben und dann dagegen getestet. Als ich bei späteren Dateien (z.B. distributor.c) jedoch überhaupt kein Interface hatte, gegen das ich testen konnte, habe ich gemerkt, wie sich auch die Signatur der Methoden verbessert hat. So habe ich Zeit gespart, indem ich beim Implementieren der Funktionen bereits wusste, was diese tun sollen und sehr wenig unnötige Funktionalität hinzukam.

6.2 C Programmierung

Da ich zusammen mit einem Kommilitonen bereits in einem früheren Semester eine Implementation von MapReduce[3] in Java implementiert habe, war für mich Concurrency an sich nichts Neues. In diesem Projekt kamen jedoch die Finessen von C hinzu und es hat sich herausgestellt, dass dies weit mehr ist als das 'der Programmierer ist für die Speicherverwaltung verantwortlich'. So habe ich zu Beginn einige Male Speicher auf dem Stack alloziert und das hat im Unit Test auch so funktioniert. Erst als dann die Funktion in einem komplexeren Szenario verwendet wurde, war die Speicheradresse plötzlich ungültig. Zudem musste ich auf dem harten Weg erfahren, dass der Speicher, der via malloc alloziert wird, nicht zwingend mit Nullen gefüllt ist. Auch hier hat eine Funktion im Unit Test funktioniert, da der Speicher angeblich initial tatsächlich überall Nullen hat, aber in einem komplexeren Szenario hatte die malloc Funktion nicht mehr frisch allozierten Speicher geliefert, sondern jenen, der zuvor vom gleichen Prozess freigegeben wurde. Wie ich dann erfahren habe, ist gerade in diesen Fällen der Speicher nicht mit Nullen gefüllt. Diesen Bug konnte ich schlussendlich mit folgendem Code auch in einem Unit Test nachstellen.

[3] http://en.wikipedia.org/wiki/MapReduce

Listing 11: Testsetup für Bug, der nur auftritt, wenn malloc keine Nullen zurückgibt (test/stomp-test.c test_create_command_strcatbug)

```
int *dummy = malloc(1024 * sizeof(int));
for (int i = 0; i < 1024; i++)
    dummy[i] = 1;
free(dummy);

// eigentlicher Testcode
```

6.3 Fazit

Dieses Seminar hat mich weit mehr Zeit gekostet als vorgesehen. Einerseits war das absehbar dadurch, dass ich die Aufgabestellung selbst definiert und umfangreicher als die vorgegebenen war. Andererseits habe ich jedoch auch vor allem Anfang viele Fehler programmiert, weil ich in C nicht so fit bin. Mit der Zeit hat sich dies jedoch gebessert und ich wurde immer produktiver und mein Code wurde - auch durch die konsequentere Umsetzung von TDD - immer robuster.

Alles in Allem hat es mir Spass gemacht diese Arbeit zu schreiben und ich bin mit meinem Message Broker zufrieden.

7 Anhang

7.1 Begriffe

Begriff	Erklärung
Topic	Nachrichten werden in Themen gruppiert. Clients abonnieren Themen und kriegen so die Nachrichten zugestellt, die an dieses Thema gesendet werden.
Publisher	Ein Client, der Nachrichten sendet
Subscriber	Ein Client, der an Nachrichten von einem Topic interessiert ist
STOMP	Simple Text Oriented Messaging Protocol
TDD	Test Driven Development / testgetriebene Entwicklung