Team: 15, Adrian Helberg

Aufgabenaufteilung: 1er Team, keine Aufteilung

Quellenangaben:

- Aufgabenstellung: http://users.informatik.haw-hamburg.de/~klauck/VerteilteSysteme/aufg1.html
- Entity-Relationship-Modell und Sequenzdiagramm erstellt mit „Draw.io“ : https://www.draw.io/
- Erstellen eines Entwurfs: http://users.informatik.haw-hamburg.de/~klauck/VerteilteSysteme/Entwurf.pdf

Bearbeitungszeitraum: 05.10.2018 – 11.10.2018

Aktueller Stand: Finaler Entwurf

Änderungen des Entwurfs: Version 2

Entwurf: Ab Seite 2

Inhalt

Rahmenbedingungen

Zu implementieren sind die Server-Komponenten CMEM, HBQ und DLQ

Zu verwenden ist die Open Source Sprache Erlang/OTP. Entwickelt wird in der Entwicklungsumgebung „IntelliJ IDEA Ultimate“ (Version 2018.*) mit dem Plugin „Erlang“ von Sergey Ignatov (http://ignatov.github.io/intellij-erlang/). Dieses bietet unter anderem folgende Eigenschaften:

- Codevervollständigung
- Hervorheben von Compiler-Fehlern und der Syntax
- Code-Inspektion

Auf einem Windows-System wird das Programm erstellt, getestet und ausgeführt. Zum Testen der Kommunikation des Programms wird eine virtuelle Maschine mit einem Linux-System verwendet, um so eine sich vom Entwicklungssystem unterscheidende IP-Adresse als Host für einen Erlang Node verwenden zu können.

Das Übergeben komplexer Datenstrukturen ist nicht gestattet. Deshalb wird intern mit den Basis-Strukturen Liste (lists) und Tupel (tuple) gearbeitet und Operationen (Algorithmen) auf diesen selbst implementiert.

Architektur

Die Aufgabe wird in einer Client-Server-Architektur in einem verteilten System umgesetzt. Dabei bietet der Server Dienste an, die der Client auf Wunsch anfordern kann.

Die Komponenten des Systems sollen austauschbar sein. Die Server-Komponente ist in folgende Sub-Komponenten unterteilt:

- HBQ
  - Hold-Back-Queue
  - Hält Nachrichten, die nicht ausgeliefert werden
  - Wird als lokaler Erlang Node ausgeführt
  - Läuft als Prozess, der mittels erlang:spawn/2 gestartet wird
- DLQ
  - Deliveryqueue
  - Hält Nachrichten, die an den Leser ausgeliefert werden können
  - Wird als globaler Erlang Node ausgeführt
  - Läuft als Prozess, der mittels erlang:spawn/2 gestartet wird
- CMEM
  - Client Memory
  - Gedächtnis für die Leser
  - Wird als lokaler Erlang Node ausgeführt
  - Läuft als Prozess, der mittels erlang:spawn/2 gestartet wird

Rahmenwerke und Bibliotheken

Um den Erlang-eigenen Observer, Debugger und ein erweitertes Kommandozeilenprogramm nutzen zu können, werden die Nodes mit dem Schlüsselwort werl gestartet.

Schnittstellen

Die einzelnen Komponenten werden mittels erlang:register/2 entweder im lokalen oder globalen Namensraum registriert. Die Kommunikation im verteilten System wird von der Erlang-Umgebung übernommen. Server-Komponenten und deren Dienste werden über den Namen, der beim Erstellen der Node gewählt wird, und dem Funktionsnamen angesprochen.

Anforderungen

Funktionale Anforderungen

- Server
    - Funktionalität: Alle beschriebenen Dienste funktionieren korrekt
    - Nachrichtenformat
        - MSG_List := [NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout]:
            - [Integer X String X 3-Tupel X 3-Tupel X 3-Tupel X 3-Tupel]
    - DLQ halt maximal ?Xdlq viele Nachrichten
    - Übertragungszeiten werden mit **erlang:timestamp()** getrackt
    - Sind keine Nachrichten beim Server vorhanden, wird eine Dummy-Nachricht versendet
    - Leser ohne Anfragen werden nach ?Xleser Sekunden beim Server abgemeldet
    - Besteht zwischen HBQ und DLQ zu 2/3 der Nachrichten eine Inkonsistenz, wird diese mit genau einer Fehlernachricht geschlossen
    - Nach einer gewissen Wartezeit ohne Anfragen, terminiert der Server
    - Ausgaben werden in Dateien Server<Node>.log und HB-DLQ<Node>.log geschrieben

Nichtfunktionale Anforderungen

- Zuverlässigkeit
    - Korrekte Auslieferung von Nachrichten an den Leser → Korrekte Nummerierung
- Effizienz
    - Möglichst optimaler Durchsatz an Nachrichten
        - Client → Server
        - Server → Client
- Benutzbarkeit
    - Anlegen der Erlang Nodes über ein Skript (z.B. BAT)
- Austauschbarkeit
    - Die ADTs (HBQ, DLQ, CMEM) müssen austauschbar sein
- Analysierbarkeit
    - Logging

User Stories

- Epics
    - Verschiedene Redakteure versenden Nachrichten („Nachrichten des Tages“, Textzeilen und Verwaltungsinformationen) an einen Server
    - Ein Server verwaltet die Nachrichten
        - Vergeben von IDs für Nachrichten
        - Verwalten von Client-Anfragen
        - Merken der Clients
    - Verschiedene Leser (Client) fragen die Nachrichten vom Server ab

Modularisierung

Paketstruktur

Jeder Node wird in einer eigenen Datei „<Node>.erl“ realisiert

Beschreibung

Die in Klammern stehenden Namen sind die Funktionsnamen der Dienste

- HBQ „hbq.erl“
    - Initialisierung (initHBQ)
        - Prozess spawnen und Prozess-ID zurückgeben
    - Terminierung (dellHBQ)
        - Prozess terminieren (exit)
    - Speichern einer Nachricht (pushHBQ)
        - Schreiben der Nachricht in interne Liste
    - Abfrage einer Nachricht (deliverMSG)
        - Zurückgeben einer Nachricht über die ID
    - Logging (listDLQ, listHBQ)
        - Schreiben des aktuellen Stands in eine Logging-Datei
- DLQ „dlq.erl“
    - Initialisierung (initDLQ)
        - Prozess spawnen und Prozess-ID zurückgeben
    - Terminierung (delDLQ)
        - Prozess terminieren (exit)
    - Speichern einer Nachricht (push2DLQ)
        - Schreiben der Nachricht in interne Liste
    - Abfrage welche Nachrichtennummer in der DLQ gespeichert werden kann (expectedNr)
        - Zurückgeben einer ID
    - Ausliefern einer Nachricht an einen Leser-Client (deliverMSG)
        - Zurückgeben einer Nachricht
    - Abfrage einer Liste aller Nachrichtennummern (listDLQ)
        - Gibt eine Liste der Nachrichtennummern zurück
    - Abfragen der Größe der DLQ (lengthDLQ)
        - Gibt die Größe der DLQ zurück

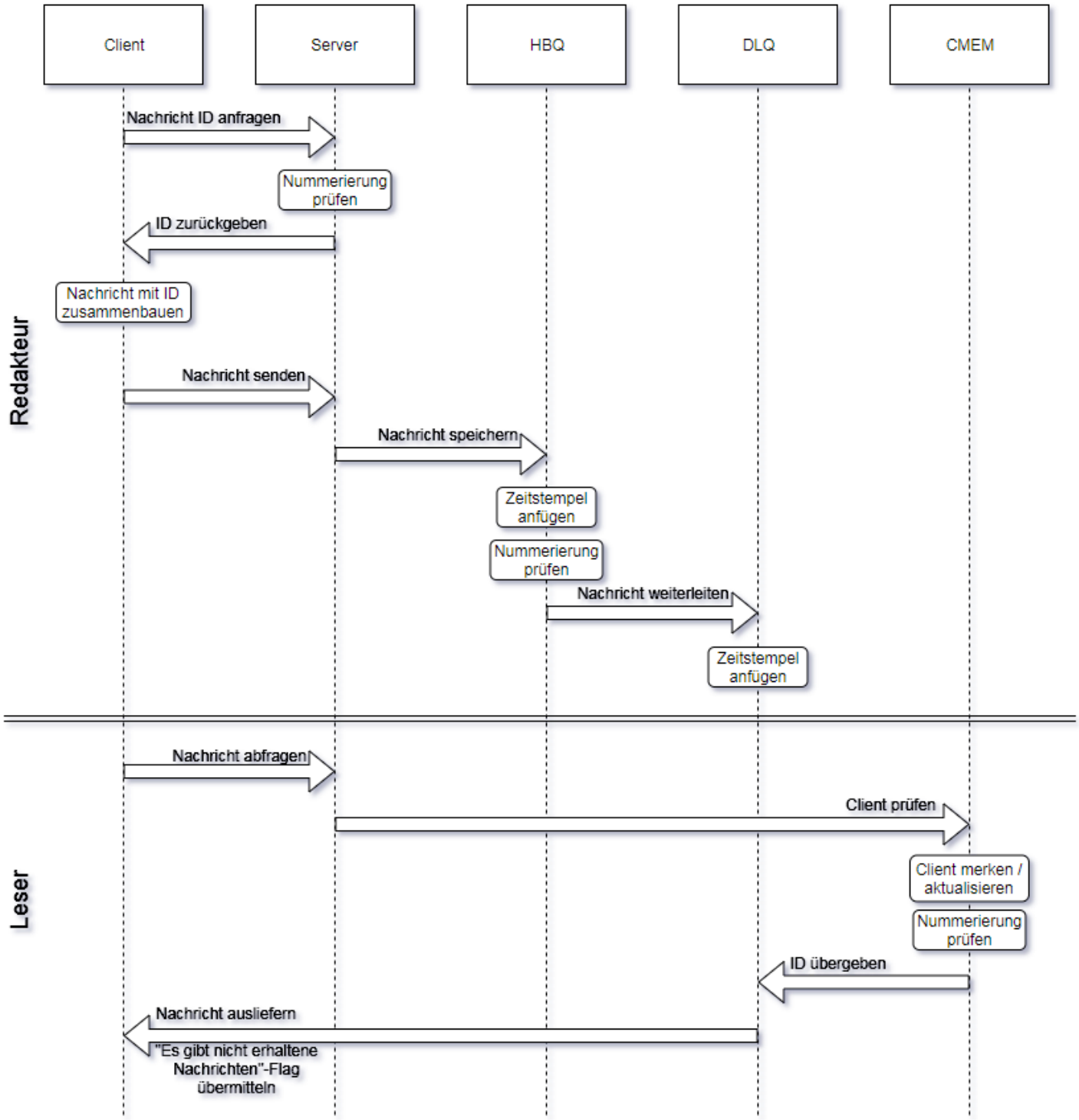
- CMEM „cmem.erl“
  - Initialisierung (initCMEM)
    - Prozess spawnen und Prozess-ID zurückgeben
  - Terminierung (delCMEM)
    - Prozess terminieren (exit)
  - Speichern/Aktualisieren eines Clients (updateClient)
    - Gibt den gespeicherten/aktualisierten Client zurück (PID)
  - Abfrage welche Nachrichtennummer der Client als nächstes erhalten darf (getClientNNr)
    - Gibt die ID zurück
  - Abfrage aller Leser (listCMEM)
    - gibt eine Liste der Leser-PIDs zurück
  - Abfragen der Größe des CMEM (lengthCMEM)
    - Gibt die Größe des CMEM zurück

Signaturen

```
% HBQ
-export([initHBQ/0, pushHBQ/3, deliverMSG/2, listDLQ/0, listHBQ/0,
dellHBQ/0]).

initHBQ() -> {reply, ok}.
pushHBQ(NNr, Msg, TSclientout) -> {reply, ok}.
deliverMSG(NNr, ToClient) -> {reply, number}.
listDLQ() -> {reply, ok}.
listHBQ() -> {reply, ok}.
dellHBQ() -> {reply, ok}.

% DLQ
-export([initDLQ/2, delDLQ/1, expectedNr/1, push2DLQ/3, deliverMSG/4,
listDLQ/1, lengthDLQ/1]).

initDLQ(Size,Datei) -> {reply, ok}.
delDLQ(Queue) -> {reply, ok}.
expectedNr(Queue) -> {reply, number}.
push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei) -> {reply, ok}.
deliverMSG(MSGNr,ClientPID,Queue,Datei) -> {reply, {number, string, tuple,
tuple}}.
listDLQ(Queue) -> {reply, ok}.
lengthDLQ(Queue) -> {reply, ok}.

% CMEM
-export([initCMEM/2, delCMEM/1, updateClient/4, getClientNNr/2, listCMEM/1,
lengthCMEM/1]).

initCMEM(RemTime,Datei) -> {reply, ok}.
delCMEM(CMEM) -> {reply, ok}.
updateClient(CMEM,ClientID,NNr,Datei)  -> {reply, ok}.
getClientNNr(CMEM,ClientID) -> {reply, number}.
listCMEM(CMEM) -> {reply, [atom]}.
lengthCMEM(CMEM)  -> {reply, number}.
```

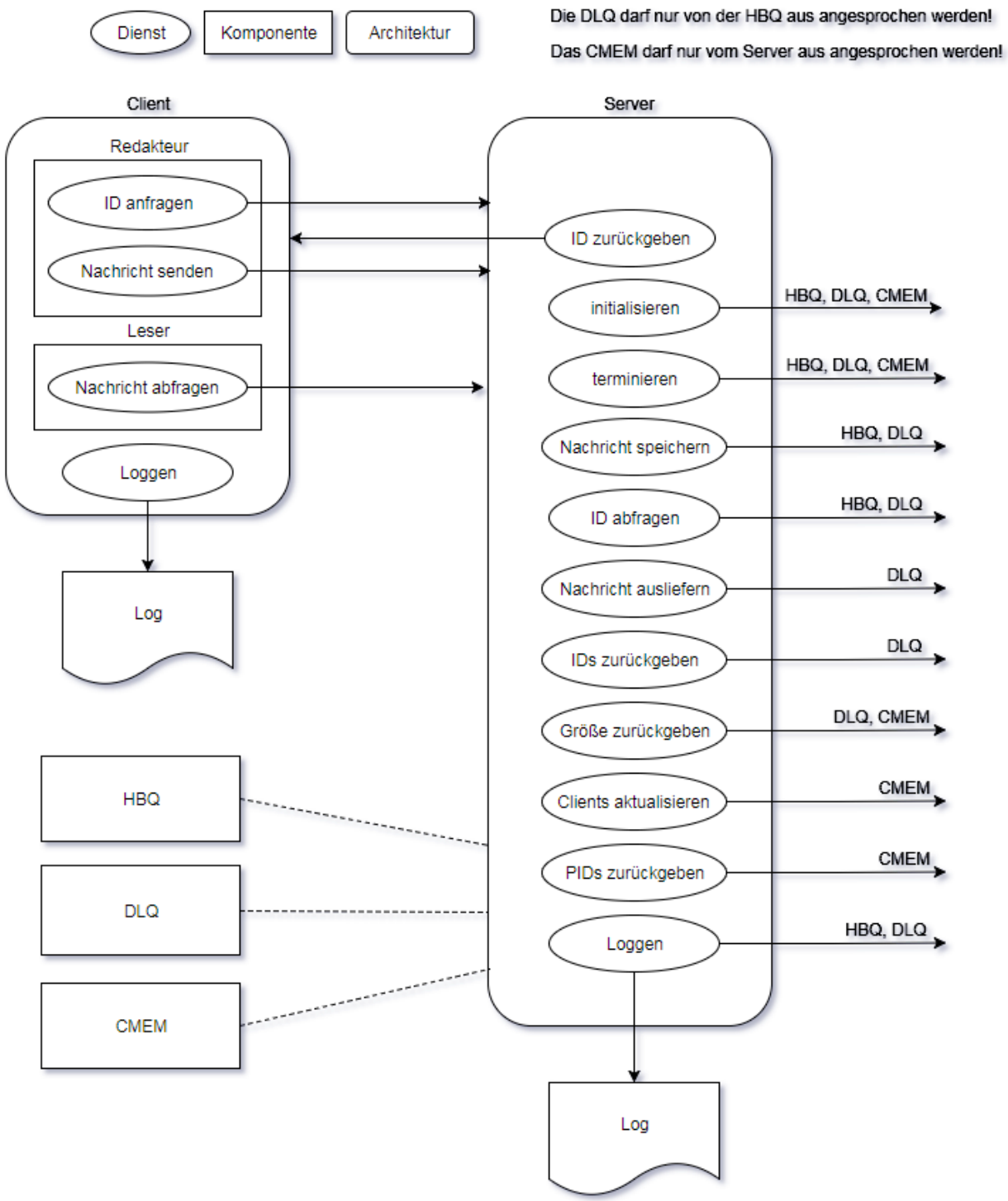
Kommunikation

Sequenzdiagramm



Datenstrukturen

Entity-Relation-Model



HBQ

Um die Daten in der HBQ zu halten wird intern eine Liste aus Tupeln genutzt. Diese Tupel beschreiben die Nachricht (NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout]: [Integer X String X 3-Tupel X 3-Tupel X 3-Tupel X 3-Tupel), die auch den Zeitstempel des Eintritt in die Queue enthält.

Die Organisation der Daten erfolgt über folgende Funktionen:

- Lesen und zurückgeben
- Schreiben
- Suchen und zurückgeben
- Sortieren
- Löschen

Wenn in der HBQ mehr als 2/3-tel der maximalen Größe der Liste der DLQ sind, wird, sofern eine Lücke besteht, diese Lücke zwischen DLQ und HBQ mit genau einer Fehlernachricht geschlossen. Daraufhin kehrt das System in den normalen Zustand zurück.

DLQ

Um die Daten in der DLQ zu halten wird intern eine Liste aus Tupeln genutzt. Diese Tupel beschreiben die Nachricht (NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout]: [Integer X String X 3-Tupel X 3-Tupel X 3-Tupel X 3-Tupel)), die auch den Zeitstempel des Eintritt in die Queue enthält. Die Größe der Liste ist auf den Wert „?Xdlq“ gesetzt.

Die Organisation der Daten erfolgt über folgende Funktionen:

- Lesen und zurückgeben
- Schreiben
- Suchen und zurückgeben
- Sortieren
- Löschen

CMEM

Um die Clients im CMEM zu speichern wird eine Liste aus Tupeln genutzt, die alle nötigen Daten über die Clients hält:

- Client ID
- Nachrichten ID der letzten ausgelieferten Nachricht an den Client
- Zeit der letzten Anfrage des Clients an den Server

Die Organisation der Client-Daten erfolgt über folgende Funktionen:

- Lesen und zurückgeben
- Schreiben
- Suchen und zurückgeben
- Löschen

Ein Leser, der seit „?Xleser“ Sekunden keine Abfrage mehr gemacht hat, wird im CMEM gelöscht.

Schnittstellenbeschreibungen

Dateischnittstellen

Ausgaben werden in Log-Dateien geschrieben

- HBQ: HB-DLQ<Node>.log

API-Schnittstellen

Um auf die Funktionen der Module zugreifen zu können, müssen diese exportiert werden

- z.B. „-export([fact/1]).“ , um eine Funktion fact, die einen Parameter entgegennimmt, zu exportieren