Team: 6, Mert Siginc, Michael Müller
Aufgabenaufteilung:
1. Client.erl und Server.erl
2. Hbq.erl, Dlq.erl und CMEM.elr

Quellenangaben:
- http://erlang.org/doc/apps/stdlib/index.html
- http://users.informatik.haw-hamburg.de/~klauck/verteiltesysteme.html

Bearbeitungszeitraum: <Datum und Dauer der Bearbeitung an der Aufgabe von allen Teammitgliedern>

Aktueller Stand:
Entwurf fertig, einzelne Teile der Komponenten fertig, einzelne Tests erstellt und „laufen".

Änderungen des Entwurfs: <Vor dem Praktikum auszufüllen: Welche Änderungen sind bzgl. des Vorentwurfs vorgenommen worden.>

Entwurf:

Allgemeine Funktion der Anwendung:
Die Anwendung realisiert eine Client/Server-Umgebung, in der von Clients zugeschickte Nachrichten vom Server verwaltet werden. Die Clients nehmen zwei Rollen an.
Der Redakteur-Client versendet eindeutig nummerierte Nachrichten an den Server. Der Leser-Client fragt in regelmäßigen Abständen den Server, ob es neue Nachrichten zu lesen gibt. Dabei ist es wichtig, dass der Leser nicht immer wieder alle Nachrichten erhält sondern, dass der Server sich an die Leser erinnert und ihm nur die Nachrichten übermittelt, die der Client noch nicht erhalten hat. Wenn ein Client einige Zeit keine neuen Nachrichten abfragt, wird dieser vom Server vergessen, liest dieser hiernach erneut, wird dieser Client als neuer Client behandelt. Der Server hat zudem die Aufgabe, den Clients die Nachrichten in der richtigen Reihenfolge zu übermitteln.

Nachrichtenformat:
Eine Nachricht besteht aus einer Liste mit folgendem Inhalt:
- Nachrichtennummer (einmalig im System)
- Text, bestehend aus:
  - Hostname
  - Praktikumsgruppe
  - Teamnummer
  - Zeitpunkt der Erstellung
- Zeitpunkt der Erstellung
- Zeitpunkt des Empfangs in der HBQ
- Zeitpunkt des Empfangs in der DLQ
- Zeitpunkt des Verlassens der DLQ

Die letzten 3 Zeitstempel kommen erst hinzu, wenn das jeweilige Event auch passiert ist (wenn die Nachrichte in der HBQ ankam, wenn sie in der DLQ ankam, wenn sie die DLQ verlassen hat).

Eine Fehlernachricht wird in der DLQ erstellt und ist äußerlich von einer normalen Nachricht nicht zu unterscheiden. Sie unterscheidet sich inhaltlich darin, dass:
- die NNr = 0 ist
- der Text „Angeforderte Nachricht nicht vorhanden." ist
- und (wenn man es vergleichen würde) alle Zeitstempel gleich sind.

Weitere Variablen:

LogDatei	= ist der Dateinamen der Log Datei
NNr	= Die eindeutige Nachrichtenummer
NNrListe	= (In Aktivitätsdiagrammen ausversehen NachrichtenListe genannt) beinhaltet alle Nachrichtennummern die vom Redakteur einer Clients geschrieben wurden
TSClientout	= Timestamp vom Moment wenn die Nachricht den Client(Redakteur) verlässt
TSHBQIn	= Timestamp vom Moment wenn die Nachricht die HBQ erreicht
TSDLQIn	= Timestamp vom Moment wenn die Nachricht die DLQ erreicht
TSDLQOut	= Timestamp von dem Moment wenn die Nachricht die DLQ verlässt
ClientPID	= Pid des jeweiligen Clients

Client:

Beschreibung:

Der Client ist dafür da, die angegebene Anzahl an Clients zu starten und diese in den Rollen Redakteur und Leser hin und her zu wechseln bis dessen Lebenszeit verstreicht. Dies gestaltet sich so, dass es einen „Mainclient" (auch Client0 genannt) gibt, der alle Unterclients startet, sich ihre PID speichert und nach Ablauf eines Timers diese terminiert. Zu dem pingt der Mainclient den Server an damit die Nodes sich kennen, alle Unterclients werden in der selben Node wie der Mainclient gestartet und brauchen somit diesen Ping nicht mehr zu machen.
Die Unterclients wissen von keinem Timer und ggf. laufen unendlich lange bis sie vom Mainclient terminiert werden.

Die Implementation des Redakteurs und des Lesers beruht dann auf zwei unterschiedlichen Loops (Redakteur_loop / Leser_loop) mit bestimmten Inputparametern. Wenn ein Wechsel vollzogen wird, geht man dementsprechend nach einem loop in den anderen loop über. Dabei wird der über die Zeit errechnete Intervall des Redakteurs nicht verworfen sondern auch im Leser loop immer mitgegeben damit er dann bei einem Wechsel dem Redakteur wieder zur Verfügung steht.

Vor jeder Nachricht die der Client verschickt, fragt es vorher nach der nächsten Nachrichtenummer und wartet nach einer verschickten Nachricht eine gewisse Zeit. Diese Wartezeit wird nach dem Senden von 5 Textzeilen jeweils um ca. 50% per Zufall vergrößert oder verkleinert. Die Wartezeit darf nicht unter 2 Sekunde rutschen. Initial ist die Wartezeit 5 Sekunden lang.
Hat der Client 4 Nachrichten verschickt, fragt er noch einmal nach einer Nachrichtenummer, vergisst aber diese Nachricht zu senden. Dies wird dann dementsprechend geloggt („NNr N, vergessen zu senden").

Die fünf Nachrichtennummern die der Redakteur verwendet hat, übergibt er in einer Liste an den Leser loop.
Somit wechselt der Client zum Leser und fragt den Server, ob es Nachrichten zu lesen gibt. Dies macht er solange bis alle Nachrichten auf dem Server gelesen wurden und der Client wieder in die Redakteursrolle übergeht. Dass keine weiteren Nachrichten vorhanden sind erkennt er am TerminatedFlag, gleich True heißt, es gibt keine weiteren Nachrichten, gleich False heißt, es gibt noch mindestens eine weitere Nachrichten.

Eingehende Nachrichten des eigenen Redakteurs (sprich deren NNr in der vom Redakteur übergebenen Liste enthalten sind) werden im Log besonders markiert. Ebenso Nachrichten die aus der DLQ heraus aus der Zukunft zu kommen scheinen, hier wird zusätzlich die Zeitliche Differenz vom Zeitstempel des Verlassens der DLQ mit Jetzt verglichen und die Differenz ausgegeben.
Log Beispiel: („Empfangene NNr N, ist von meinem Redakteur und aus der Zukunft um DIFFERENZ")

Anders dargestellt sind es folgende Abläufe:

- Redakteur:
  - Fragt nach neuer NNr beim Server
  - Mit NNr Nachricht erstellen
  - Die Intervallzeit warten
  - NNr in NNrListe hinzufügen
  - Nachricht checken
    - Ist die NNrListe 5 Elemente groß: Nicht senden und dementsprechend logen
    - Ist sie kleiner als 5: Nachricht an Server senden
  - Neue Intervallzeit wird errechnet
  - Wechsel checken
    - Ist die NNrListe 5 Elemente groß wird zum Leser gewechselt, dem Leser wird als Parameter die NNrListe und das neue Intervall mitgegeben
    - Ist sie kleiner als 5 wird der Redakteursloop von vorne begonnen, mit der aktuellen NNrListe und dem neuen Intervall
- Leser:
  - Fragt nach neuer Nachricht beim Server
  - Empfangene Nachricht wird geloggt:
    - Loggt wenn, die Nachricht aus der Zukunft ist
    - Loggt wenn, die Nachricht von meinem Redakteur (wenn NNr in übergebener NNrListe enthalten) ist.
  - Terminated Flag checken:
    - = True: Zu Redakteur mit Intervall und leerer NNrListe wechseln
    - = False: Weiterer Leser Loop mit unverändertem Intervall und unveränderter NNrListe

Server:

Beschreibung:

Der Server agiert als Schnittstelle für die Clients und verwaltet die eingehenden Nachrichten. Der Server arbeitet mit der HBQ und der CMEM zusammen, um die Anfragen vom Client zu bewerkstelligen.
Der Server besteht hauptsächlich aus seinem Loop. Als Parameter erwartet er die aktuelle CMEM sowie die nächste NNr für einen Client. Die HBQ ist mit Registrierten Namen und Node bekannt und wird darüber angesprochen.
Der Server hat zudem die Aufgabe, eindeutige Nachrichtennummern zu verteilen.
Der Server terminiert sobald der letzte Kontakt mit einem Client länger ist als die Wartezeit. Daraufhin wird die CMEM und HBQ gelöscht.
Der Server muss eine Aufgabe (hier aufgeführt als Schnittstellen) zuerst beenden bevor er sich der Nächsten widmen kann, so wird vermieden, dass 2 Clients die gleiche NNr bekämen.

Schnittstellen:

Abfragen der eindeutigen Nachrichtennummer:
Server ! {self(),getmsgid}
receive {nid, Number}
Hierüber wird eine neue (im System einmalige) Nachrichtennummer an den Client gesendet. Initial ist sie 1. Die im Loop bekannte NNr für Clients wird um 1 erhöht und der Loop wird von vorn gestartet.

Senden einer Nachricht:
Server ! {dropmessage,[INNr,Msg,TSclientout]}
Sobald ein eine Nachricht beim Server ankommt, wird diese zur Speicherung an die HBQ weitergeleitet. Es wird ein {reply, ok} zurückerwartet, damit der Server weiß, dass alles gut lief. Kommt binnen 5 Sekunden dies jedoch nicht wird geloggt, dass die Nachricht wohl nicht erfolgreich an die HBQ gesendet und eingefügt werden konnte. Die Nachricht wird dann kein zweites Mal zur HBQ gesendet. Der Server Loop beginnt dann mit unveränderten Parametern von vorn.

Abfragen einer Nachricht:
Server ! {self(), getmessages}
receive {reply,[NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout],Terminated} }
Sobald der Client Lesen möchte, ruft der Server aus der CMEM die nächste Nachrichtennummer für diesen Client ab (CMEM:getClientNNR) und beauftragt die HBQ, diese Nachricht an den Client zu verschicken ({self(), {request, deliverMSG, ZuSendendeNNr, LeserPid}}).
Daraufhin wartet der Server bis er {reply, GesendeteNNr} zurückbekommt. Der Eintrags des Lesers wird in der CMEM dann mit der Gesendeten NNr geupdated (CMEM:updateClient).
Die neue CMEM wird zurück gegeben und der Server loop beginnt mit neuer CMEM von vorn.

HBQ:

Beschreibung:
Die HBQ wird zur Verwaltung der einkommenden Nachrichten der Clients genutzt. Nachrichten werden entweder in der HBQ oder der DLQ gespeichert. Will der Client lesen, so wird die DLQ von der HBQ beauftragt eine Nachricht an den Client zu schicken. Nur die HBQ greift auf die DLQ zu.
Die HBQ an sich besteht aus einer Liste, diese enthält nur Nachrichten und ist, anhand der NNr, aufsteigend sortiert.
Nach dem starten der Node und darin dem ausführen von hbq:start(), wartet die HBQ auf die Initialisierung. Danach wechselt sie in einen Loop der ankommende Nachrichten sequenziell abarbeitet und als Parameter die aktuelle Nachrichtenliste und die aktuelle DLQ erwartet.

Schnittstellen:
Initialisieren der HBQ:
HBQ ! {self(), {request,initHBQ}}
receive {reply, ok}
Beim Starten des Servers wird die HBQ (leere Liste) und DLQ initialisiert (DLQ:initDLQ) und wechselt in den Loop.

Terminierung der HBQ:
HBQ ! {self(), {request,dellHBQ}}
receive {reply, ok}
Sobald der Server terminiert, wird diese Schnittstelle vom Server angesprochen um die HBQ ebenfalls zu terminieren. Hier wird dann auch die DLQ gelöscht (DLQ:delDLQ).

Speichern einer Nachricht:
HBQ ! {self(), {request,pushHBQ,[NNr,Msg,TSclientout]}}
receive {reply, ok}
Sobald eine Nachricht vom Client über den Server erhalten wurde wird hierfür ein Zeitstempel an der Nachricht angehängt. Danach wird geprüft, ob die Nachrichtennummer der eingegangenen Nachricht, mit der erwarteten Nummer der DLQ übereinstimmt.
Ist dies der Fall, wird die Nachricht in der DLQ gespeichert, falls nicht wird die Nachricht in der HBQ gespeichert. Hierbei ist auf die aufsteigende Sortierung (anhand der NNr) zu achten!
Ist die neue Nachricht in der HBQ eingefügt wird geprüft ob die HBQ bereits mehr Nachrichten hält als 2/3 der DLQ Größe. Ist dies True wird die Lücke geschlossen. Diese Lücke zwischen DLQ und HBQ wird mit genau einer Fehlernachricht geschlossen, etwa: "***Fehlernachricht fuer Nachrichtennummern 11 bis 17 um 16.05 18:01:30,580", indem diese Fehlernachricht in die DLQ eingetragen wird und als Nachrichten-ID die größte fehlende ID der Lücke erhält (im Beispiel also 17). Es werden zunächst keine weiteren Lücken innerhalb der HBQ behandelt, da das System nach Generierung der Fehlernachricht zunächst in den normalen Zustand zurückkehrt.
Mit neuer Nachrichtenliste und DLQ wird der Loop von vorn begonnen.

Abfrage einer Nachricht:
HBQ ! {self(), {request,deliverMSG,NNr,ToClient}}
receive {reply, SendNNr}
Die HBQ leitet die Anfrage an die DLQ weiter (DLQ:deliverMSG), wo diese behandelt und beantwortet wird. Als Rückmeldung erhält die HBQ die gesendete Nachrichtennummer ({reply, GesendeteNNr}), die die HBQ wiederum an den Server schicken wird. Unverändert beginnt dann der Loop von vorn.

CMEM:

Beschreibung:
Die CMEM wird als Speicher genutzt, um die Clients-Nachrichtennummer Zuordnung zu verwalten. Es merkt sich für jeden Client, welche Nachricht bereits an welchen Client geschickt wurde. Nur der Server greift auf die CMEM zu.
Die CMEM besteht aus einem Tupel:

- Erinnerungszeit in Sekunden
- Liste mit Tupeln, ein Tupel hat wiederum (nachfolgend Tupelliste genannt):
  - ClientPid
  - Letzte gesendete Nummer
  - Zeitstempel des letzten Sendens

Die Tupelliste ist unsortiert.

Funktionen:
delCMEM(CMEM):

Sobald der Server terminiert, wird diese Funktion vom Server aufgerufen um die CMEM gleichen falls zu terminieren.

initCMEM(RemTime,Datei):
Beim Starten des Servers wird die CMEM initialisiert und steht dem Server hiernach zu Verfügung. Initial besteht sie aus einer Liste mit:

- Erinnerungszeit
- Leere Liste

updateClient(CMEM,ClientPID,NNr,Datei):
Sobald ein Client aus dem Server liest, wird die CMEM für diesen Client, zur letzten empfangenen Nachrichtennummer aktualisiert, so dass der Client bei der nächsten Abfrage die nächste Nachricht erhält.
Hierfür läuft der Algorithmus durch die (ggf.) ganze Tupelliste um den richtigen Tupel mit der ClientPid zu finden. Ist kein passender Eintrag vorhanden wird das neue Tupel (ClientPid, NNr und Jetzt-Zeitstempel) hinten angehängt.
Die neue CMEM wird zurückgegeben.

getClientNNr(CMEM,ClientPID):
Diese Funktion gibt zurück welche NNr als nächstes an den Client gesendet werden kann.
Der Algorithmus läuft nun durch (ggf.) die ganze Tupelliste um das richtige Tupel (anhand der ClientPID) zu finden.

- Wird ein passendes Tupel gefunden
  - Wird die Differenz des Tupel Zeitstempels und dem Jetzt-Zeitstempel verglichen.
    - Ist mehr Zeit vergangen als Erinnerungszeit, wird gehandelt als wäre kein passender Eintrag gefunden worden
    - Ist weniger oder gleich Zeit vergangen wird die gespeicherte NNr + 1 zurückgegeben
- Wird kein passendes Tupel gefunden wird mit 1 geantwortet, die Initial NNr

DLQ:

Beschreibung:

Die DLQ wird zur Speicherung der Nachrichten genutzt, die von den Clients gelesen werden können. Sobald ein Client vom Server lesen möchte, wird die DLQ vom Server über die HBQ beauftragt, die nächste Nachricht an den Client zu übermitteln. Die DLQ darf nur von der HBQ angesprochen werden.
Sie besteht aus einem Tupel:

- Größe der DLQ
- Liste mit Nachrichten (anhand der NNr) in absteigender Sortierung

Funktionen:

delDLQ(Queue):
Sobald die HBQ terminiert, wird diese Funktion von der HBQ aufgerufen um die DLQ gleichen falls zu terminieren.

initDLQ(Size,Datei):
Beim initialisieren der HBQ wird die DLQ initialisiert und steht der HBQ hiernach zu Verfügung. Initial wird also ein folgendes Tupel zurückgegeben:

- DLQ Größe,

- Leere Nachrichtenliste

push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei):
Zunächst wird geprüft ob die DLQ voll ist:
- Ist sie es wird die letzte Nachricht verworfen
- Ist sie es nicht wird nichts unternommen

Danach wird die Nachricht mit einem Zeitstempel versehen und ganz vorne in die Nachrichtenliste eingefügt.
Die neue DLQ wird zurückgegeben.

deliverMSG(NNr,ClientPID,Queue,Datei):
Diese Funktion übermittelt eine Nachricht an den Client.

Der Algorithmus arbeitet wie folgt:
- Es wird durch die Nachrichtenliste gegangen um zu sehen ob die Nachricht mit der geforderten NNr vorhanden ist. Es wird von vorn jede Nachricht in der Nachrichtenliste überprüft da man aufgrund der nicht-deterministischen Lücken nicht weiß wo welche Lücke geschlossen wurde. Weshalb man nicht die Position errechnen kann.
  - Ist sie es
    - Wird geprüft ob es noch eine weitere Nachricht (größere NNr) in der Liste gibt, ist sie vorhanden ist das TerminatedFlag = False, sonst True
    - Die Nachricht wird mit der TeminatedFlag und einem neuen Zeitstempel an den Client geschickt
    - Die gesendete NNr wird zurückgegeben
  - Ist sie es nicht
    - Aber es gibt eine nächst Größere NNr so wird diese Nachricht ausgewählt, Zeitstempel angehängt, das TerminatedFlag gesetzt (siehe oben) und beides an den Client geschickt
    - Ist auch keine größere NNr mehr in der Nachrichtenliste vorhanden wird eine Fehlernachricht generiert und deren NNr zurückgegeben, TerminatedFlag ist hier stets True

expectedNr(Queue):
Fragt die DLQ welche Nachricht als nächstes in der DLQ gespeichert werden kann.
Da die DLQ absteigend sortiert ist, nehmen wir hierfür nur die erste Nachricht aus der Nachrichtenliste und geben die NNr + 1 zurück.
Ist die Nachrichtenliste leer wird eine 1 (Initial NNr) zurückgegeben.

Folgende Config Values können in den genannten Dateiern gesetzt werden:

In client.cfg:
{clients, Zahl}.
Anzahl der zu startenden clients.
{lifetime, Zahl}.
Lebenszeit eines einzelnen clients in Sekunden
{servername, Atom}.
Registrierter Name des Servers.
{servernode, Node}.

Node auf dem der Registrierte Server zu finden ist.

In server:cfg:

{servername, Atom}.
Der Registrierte Servernamen.

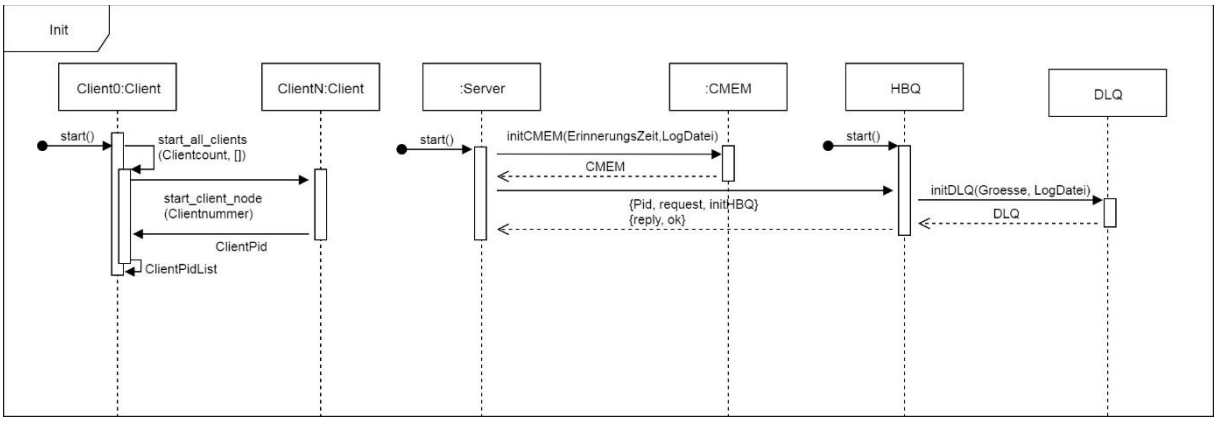
{latency, Zahl}.
Latenz in Sekunden.
Wird verwendet um nach der letzten empfangenen Nachricht und nach der Latenz herunterzufahren.

{clientlifetime, Zahl}.
Erinnerungszeit in Sekunden für die CMEM.
Gibt an wie lang an einen beliebigen aber bestimmten Client in der CMEM gedacht wird.

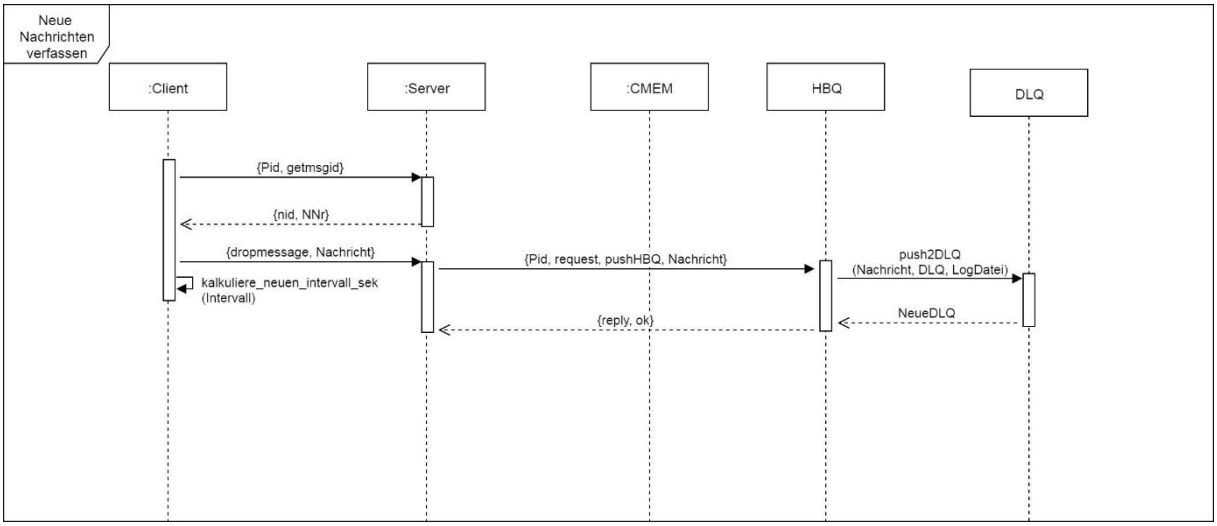
{hbqname, Atom}.
Der Registrierte HBQnamen.

{hbqnode, Node}.
Node auf dem die Registrierte HBQ zu finden ist.

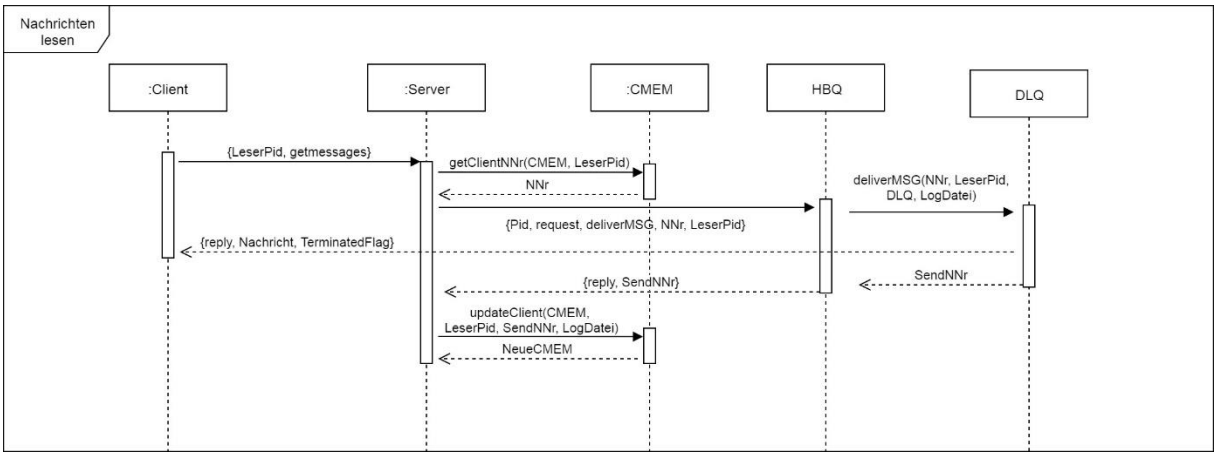
UML Sequenzdiagramme



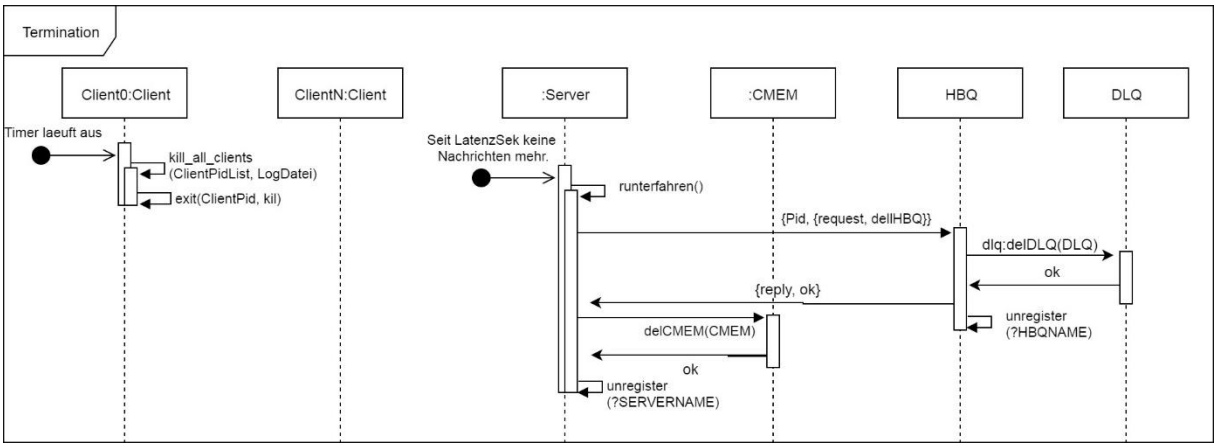
Sequenzdiagramm zur Initiierungsphase



Sequenzdiagramm zum Redakteur / für das erstellen und erfassen neuer Nachrichten im System

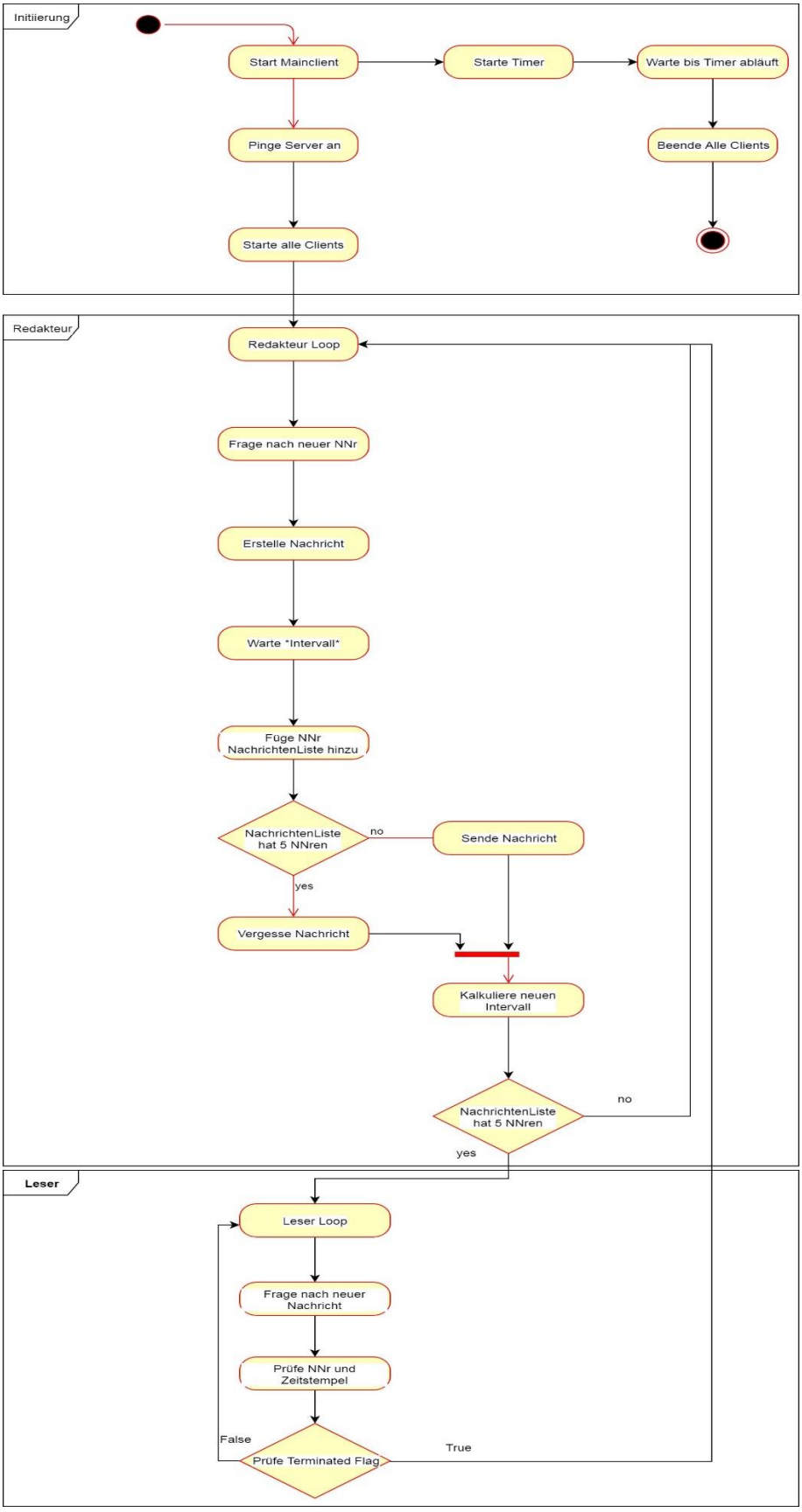


Sequenzdiagramm zum Leser / für das Lesen der vom System erfassten Nachrichten

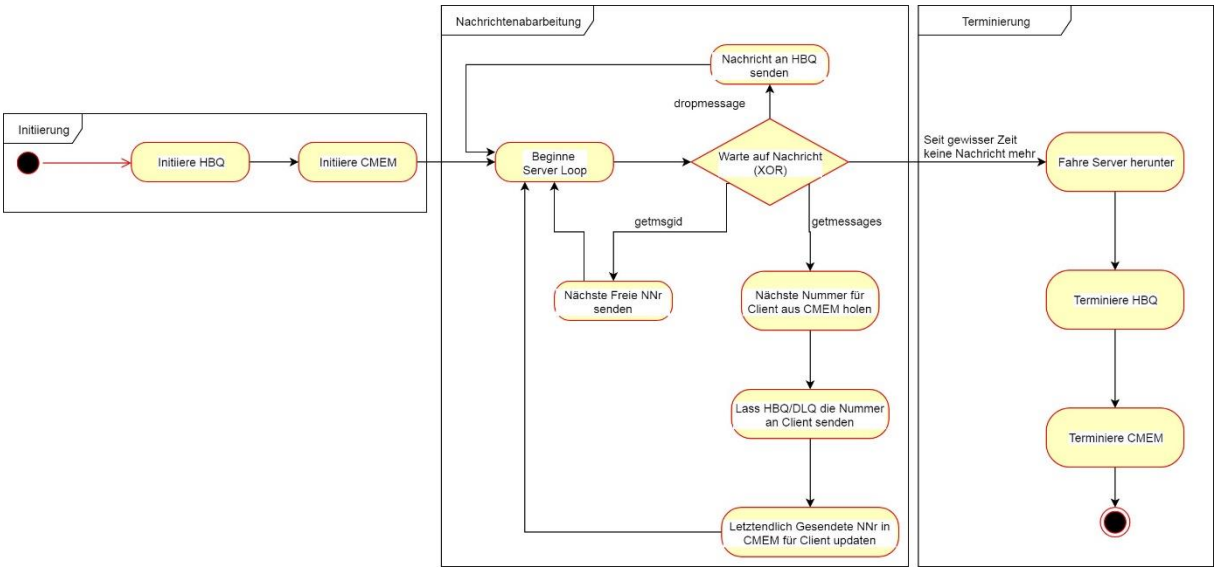


Sequenzdiagramm zur Termination des Systems

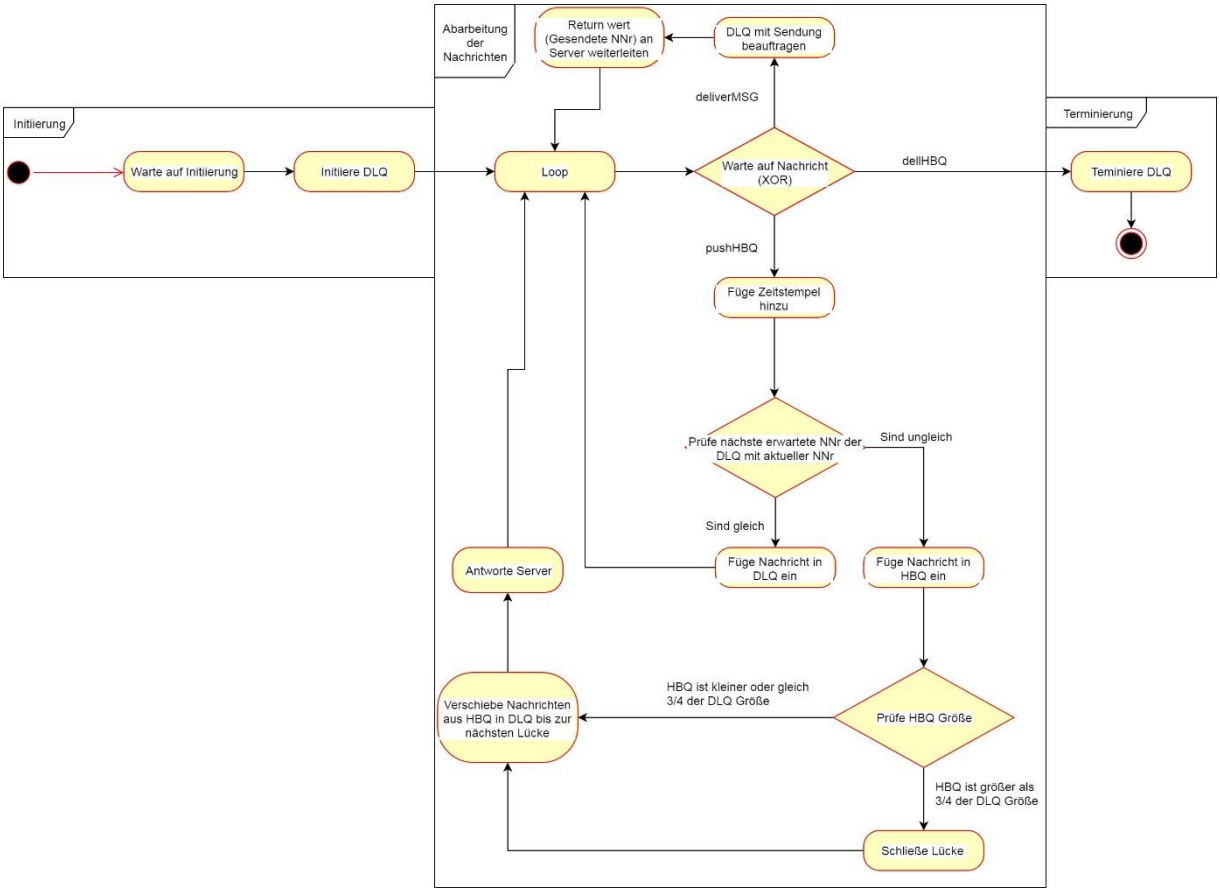
UML Aktivitätsdiagramme



Client Aktivität



Server Aktivität



HBQ Aktivität