Entwurf VSP1

Team: 10 (Antoni Romanski, Benjamin Schröder)

Aufgabenaufteilung:

1. Der Entwurf wurde von beiden Teammitgliedern gemeinsam erstellt.
2. Die Implementation wurde wie folgt aufgeteilt:

Datei	Bearbeitet durch
`server.erl`	Antoni, Benjamin
`cmem.erl`	Benjamin
`hbq.erl`	Antoni
`dlq.erl`	Antoni
`client.erl`	Benjamin

Quellenangaben: Folien und Skript, sowie Vorlesungsmitschriften von Prof. Dr. Christoph Klauck.

Bearbeitungszeitraum:
Gemeinsam: 17.04.2023 (3 Stunden), 19.04.2023 (1 Stunde 30 Minuten), 21.04.2023 (4 Stunden), 24.04.2023 (4 Stunden 30 Minuten), 25.04.2023 (5 Stunden 30 Minuten), 26.04.2023 (3 Stunden 30 Minuten), 27.04.2023 (9 Stunden), 29.04.2023 (7 Stunden), 30.04.2023 (2 Stunden), 03.05.2023 (14 Stunden 30 Minuten), 04.05.2023 (3 Stunden 30 Minuten)= 58 Stunden (*2)

Antoni: 01.05.2023 (4 Stunden), 02.05.2023 (12 Stunden)

Benjamin: 01.05.2023 (6 Stunden), 02.05.2023 (10 Stunden)

Projektstruktur

Datei	Beschreibung
`server.erl`	Server, welcher Nachrichten empfängt und verarbeitet
`server.cfg`	Konfigurationsdatei für den Server
`cmem.erl`	Nachrichtennummerspeicher des Servers
`hbq.erl`	Warteschlange, welche Nachrichten des Servers verwaltet
`dlq.erl`	Warteschlange, welche Nachrichten an Clients zurückschickt
`client.erl`	Client, welcher zugleich Nachrichten an den Server sendet und Nachrichten vom Server liest
`client.cfg`	Konfigurationsdatei für den Client

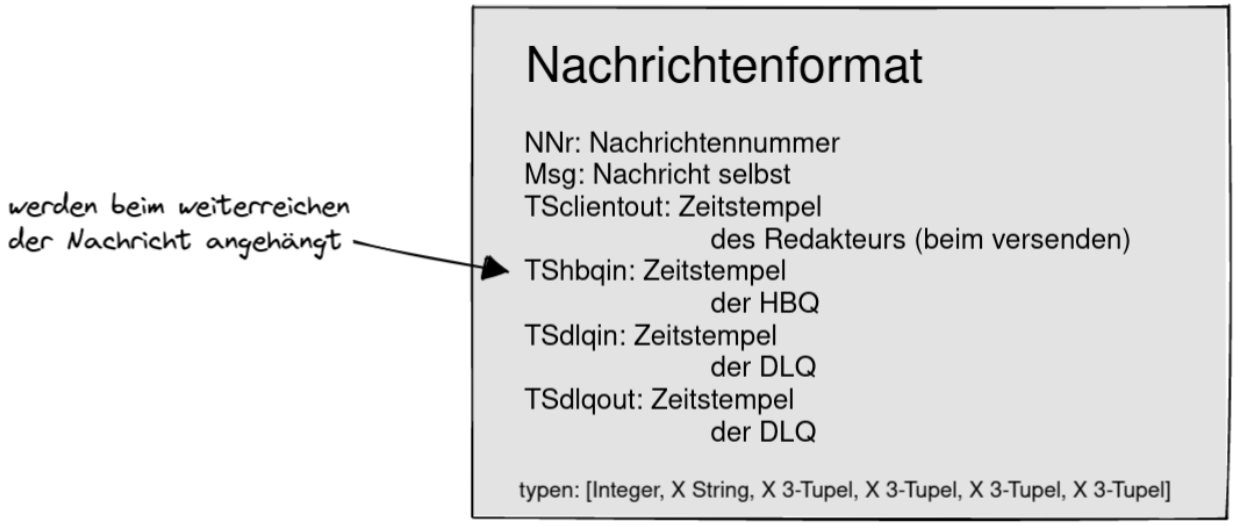
Projektbeschreibung

Dieses Projekt implementiert eine Client-Server-Architektur, welche einen Nachrichtendienst simuliert.

Dabei gibt es mehrere Clients, welche als Redakteur oder Leser agieren, und einen Server, welcher Nachrichten zwischen den Clients vermittelt.

Nachricht/Message

Das Nachrichtenformat, welches vom Server und Client benutzt wird, kann wie folgt beschrieben werden:



Entwurf

Server

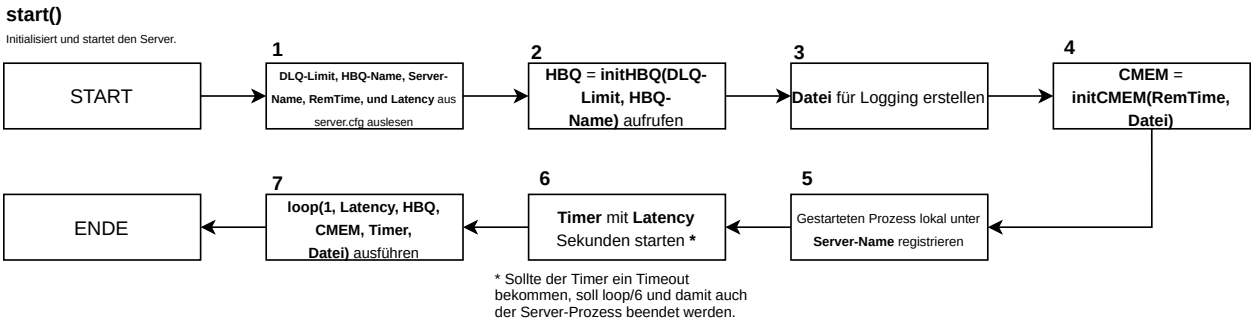
In einer Datei `server.erl` zu finden. Die Konfiguration des Servers ist in der Datei `server.cfg` möglich:

Parameter	Beschreibung
`latency`	Zeit bis zum automatischen Beenden des Servers nach der letzten Interaktion mit einem Client.
`clientlifetime`	Zeit bis zum Vergessen eines Clients im CMEM (entspricht der dortigen `RemTime` bzw. `ErinnerungsZeit`).
`servername`	Name des Server-Prozesses im lokalen Namensdienst.
`hbqname`	Name des HBQ-Prozesses im lokalen Namensdienst.
`hbqnode`	Name der Node, auf welcher der HBQ-Prozess läuft.
`dlqlimit`	Kapazität der DLQ.

Der Server empfängt Nachrichten vom Client und schickt diese auch wieder an Clients zurück.

Er beinhaltet außerdem das `CMEM`, mit welchem er sich die Nachrichtennummern der Clients merkt.

Die Nachrichten selbst verwaltet der Server in der `HBQ` bzw. `DLQ`, welche in einem eigenen HBQ-Prozess laufen und über dessen Schnittstellen angesprochen werden.



loop(NNrCounter, Latency, HBQ, CMEM, Timer, Datei)

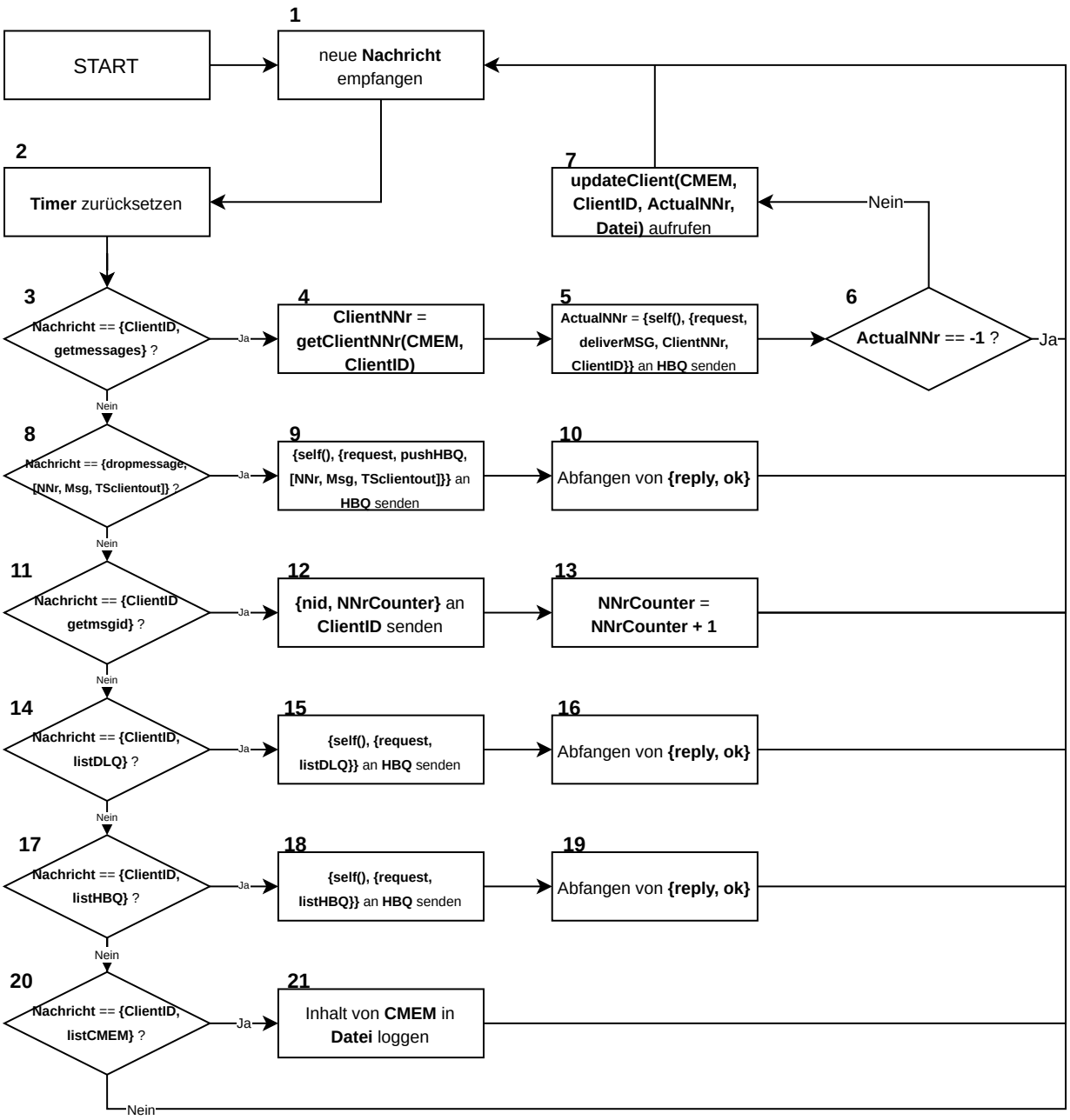
Empfängt und verarbeitet kontinuierlich Nachrichten im Server-Prozess, bis dieser automatisch nach einer Zeit von Latency beendet wird.
HBQ ist hier die PID des HBQ-Prozesses. Es gibt die folgenden Schnittstellen:

- getmessages: liefert eine weitere Nachricht an den Client aus und updated anschließend das CMEM

- dropmessage: speichert die mitgelieferte Nachricht in der HBQ ab, dazu wird die Anfrage an den HBQ-Prozess weitergeleitet

- getmsgid: sendet dem Client die nächste eindeutige Nachrichtennummer zurück und erhöht anschließend den NNr-Counter

- listDLQ/HBQ/CMEM: bewirkt ein Logging der DLQ/HBQ/CMEM in einer Datei



CMEM

In einer Datei `cmem.erl` zu finden.

In der CMEM (ClientMemory) merkt sich der Server, wann er zuletzt mit einem Client kommuniziert hat und welche seine letzte Nachrichtennummer war.

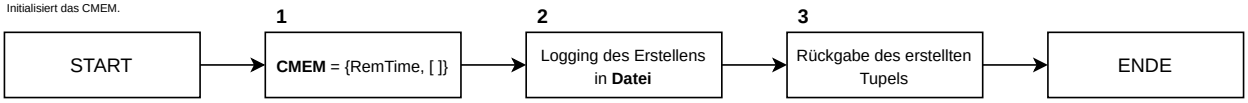
Nach einer gewissen Zeit (`ErinnerungsZeit`) vergisst er einen Client wieder, damit das CMEM nicht unendlich groß wird.

CMEM: `[<ErinnerungsZeit>, [{<Client-ID>, <LetzteInteraktion>, <Nachrichtennummer>}, …]]`

⇒ LetzeInteraktion = Zeitstempel

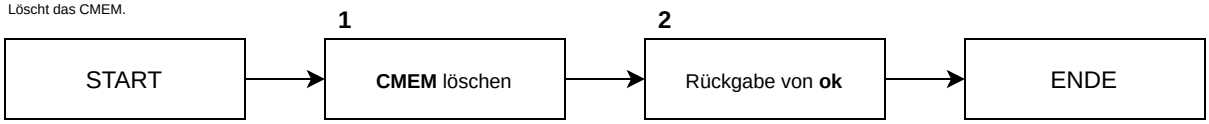
initCMEM(RemTime, Datei)

Initialisiert das CMEM.



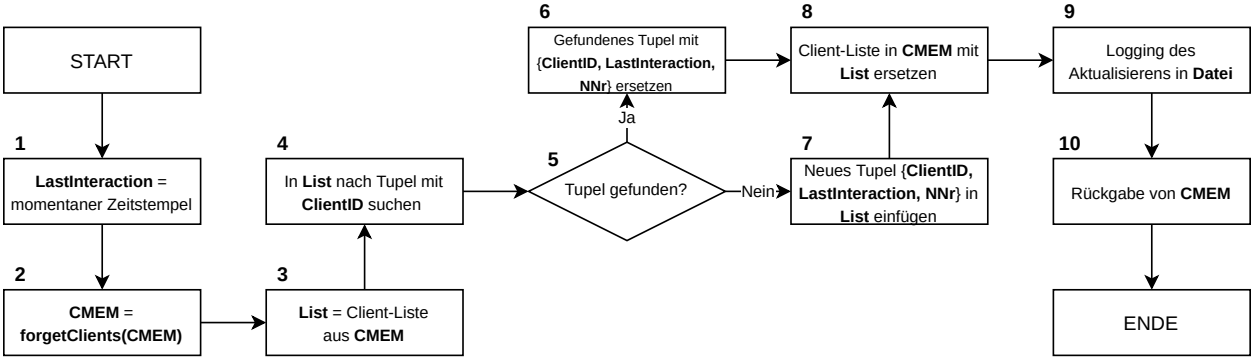
delCMEM(CMEM)

Löscht das CMEM.



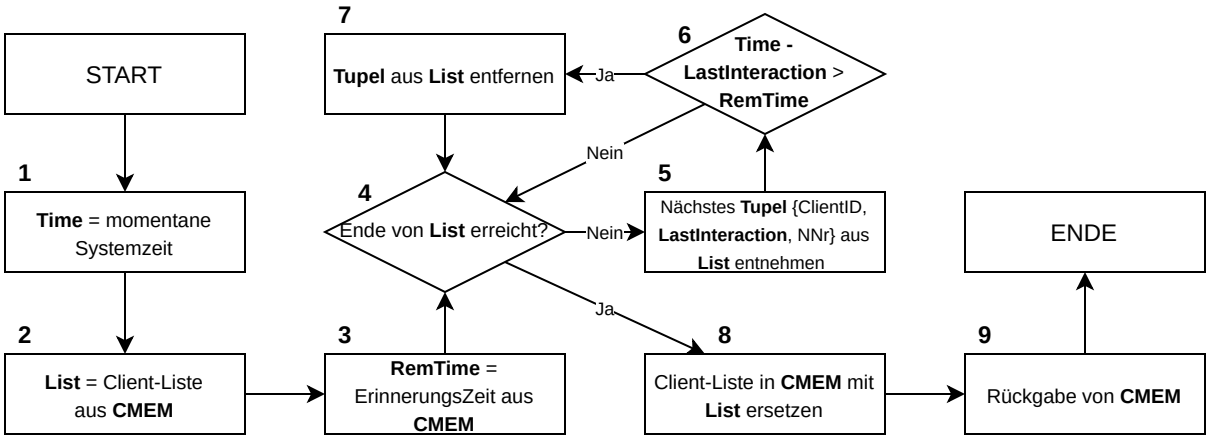
updateClient(CMEM, ClientID, NNr, Datei)

Aktualisiert einen Eintrag im CMEM mit der übergebenen Nachrichtennummer. Gibt es für den angegebenen Client noch keinen Eintrag, wird dieser erstellt, wobei ihm die Nachrichtennummer 1 zugewiesen wird.



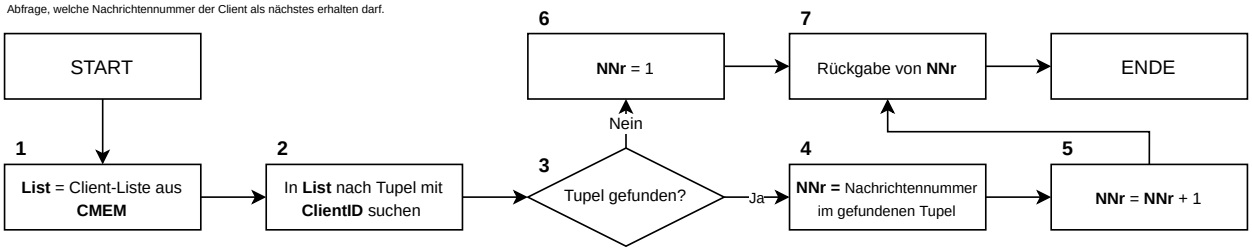
forgetClients(CMEM)

Entfernt Einträge von allen Clients, die sich zu lang nicht beim Server gemeldet haben (länger als RemTime), aus dem CMEM.



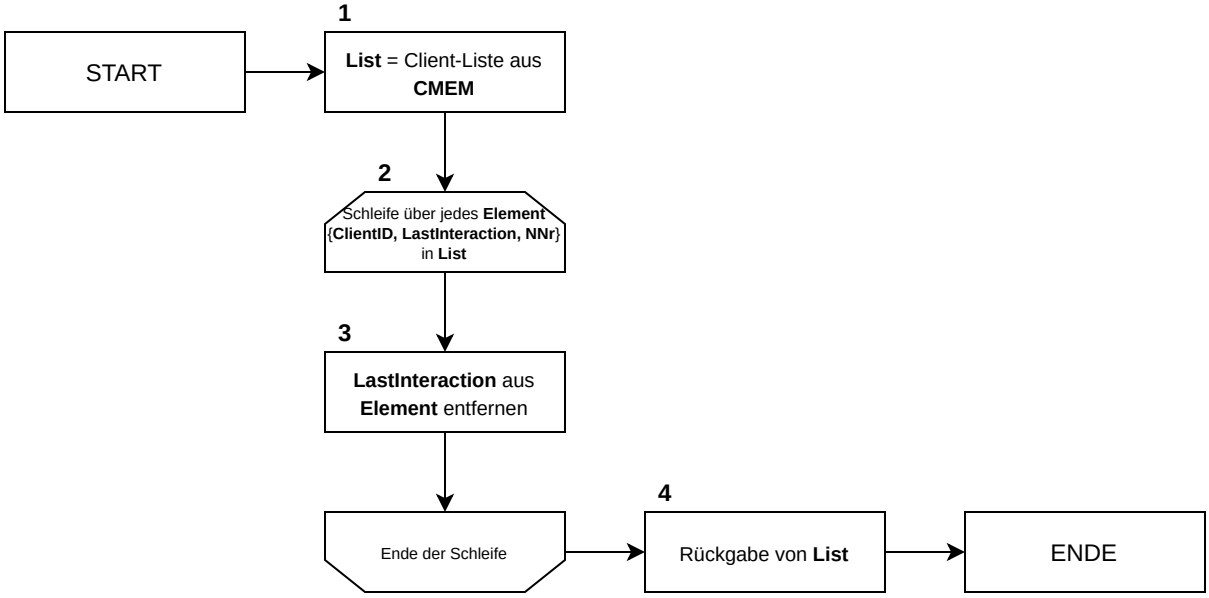
getClientNNr(CMEM, ClientID)

Abfrage, welche Nachrichtennummer der Client als nächstes erhalten darf.



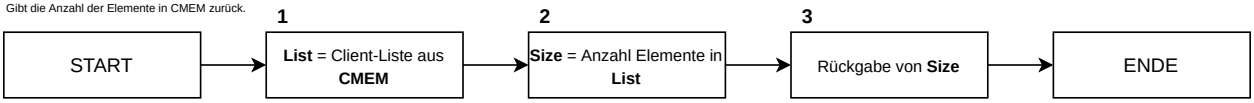
listCMEM(CMEM)

Gibt eine Liste aller ClientIDs und den dazugehörigen Nachrichtennummern im CMEM zurück.



lengthCMEM(CMEM)

Gibt die Anzahl der Elemente in CMEM zurück.



HBQ

In einer Datei `hbq.erl` zu finden.

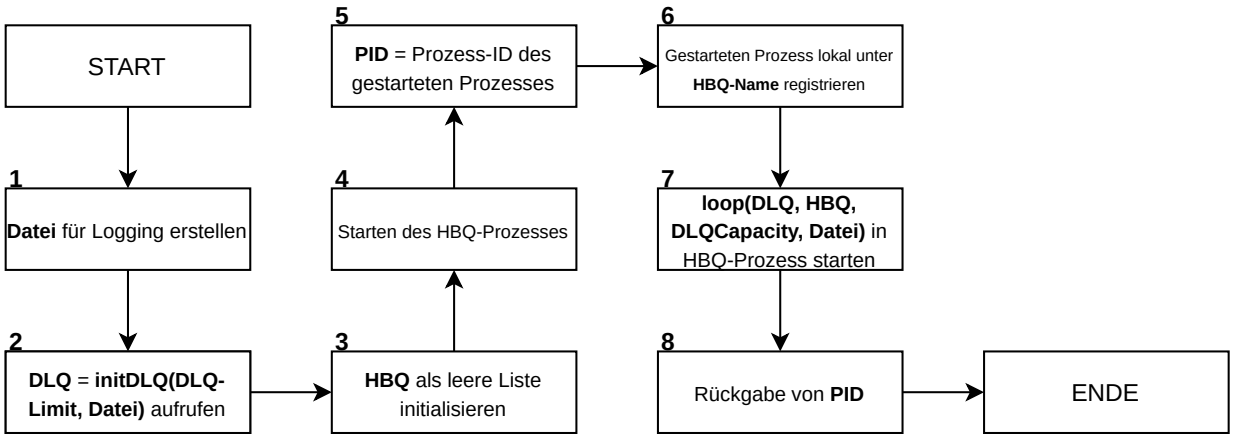
Enthält alle Nachrichten, welche vom Client erhalten wurden.

HBQ: `[<Nachricht1>, <Nachricht2>, …]`

⇒ Nachrichten aufsteigend sortiert nach Nachrichtennummer

initHBQ(DLQ-Limit, HBQ-Name)

Startet den HBQ-Prozess und initialisiert die HBQ sowie die DLQ.



loop(DLQ, DLQCapacity, HBQ, Datei)

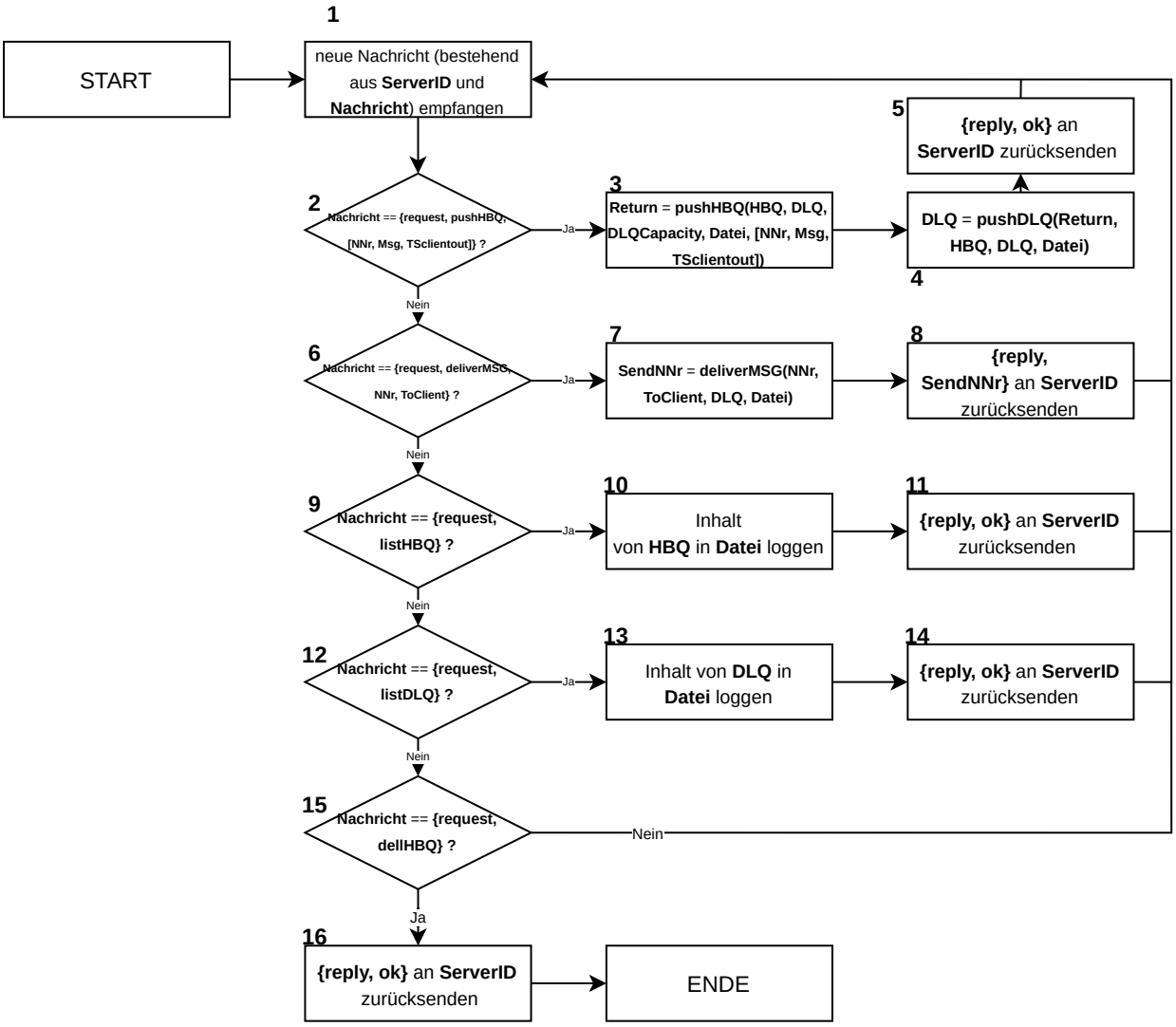
Empfängt und verarbeitet kontinuierlich Befehle vom Server im HBQ-Prozess, bis dieser beendet wird.
Es gibt die folgenden Schnittstellen:

- pushHBQ: fügt eine mitgesendete Nachricht in die HBQ ein und updatet ggf. die DLQ

- deliverMSG: leitet eine Anfrage zum Ausliefern einer Nachricht an die DLQ weiter und sendet im Anschluss die Nachrichtennummer der tatsächlich versendeten Nachricht an den Server zurück

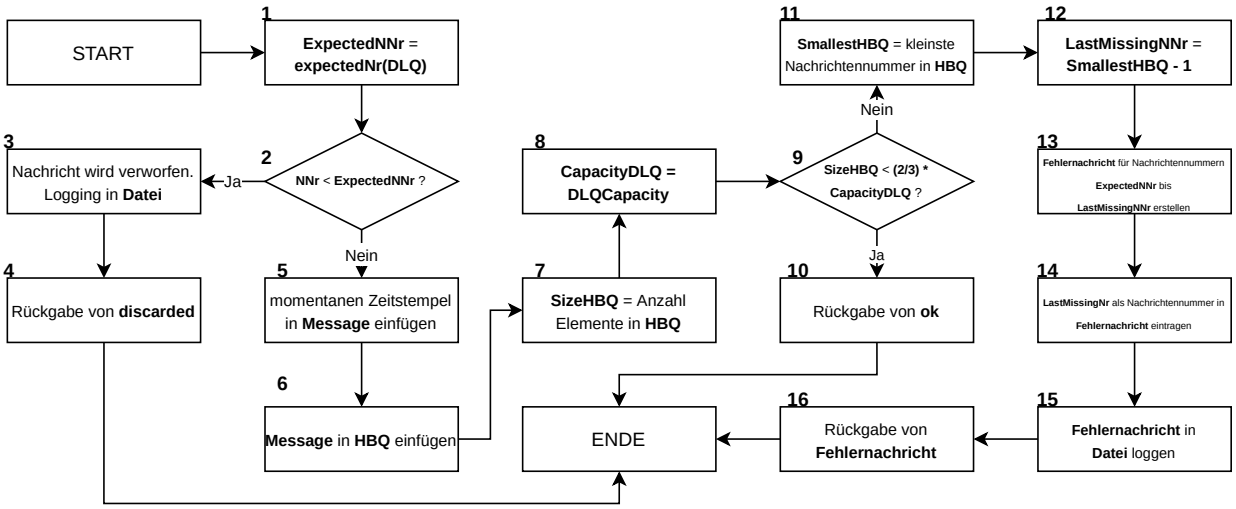
- listHBQ/listDLQ: bewirkt ein Logging der HBQ/DLQ in einer Datei

- delHBQ: beendet den HBQ-Prozess



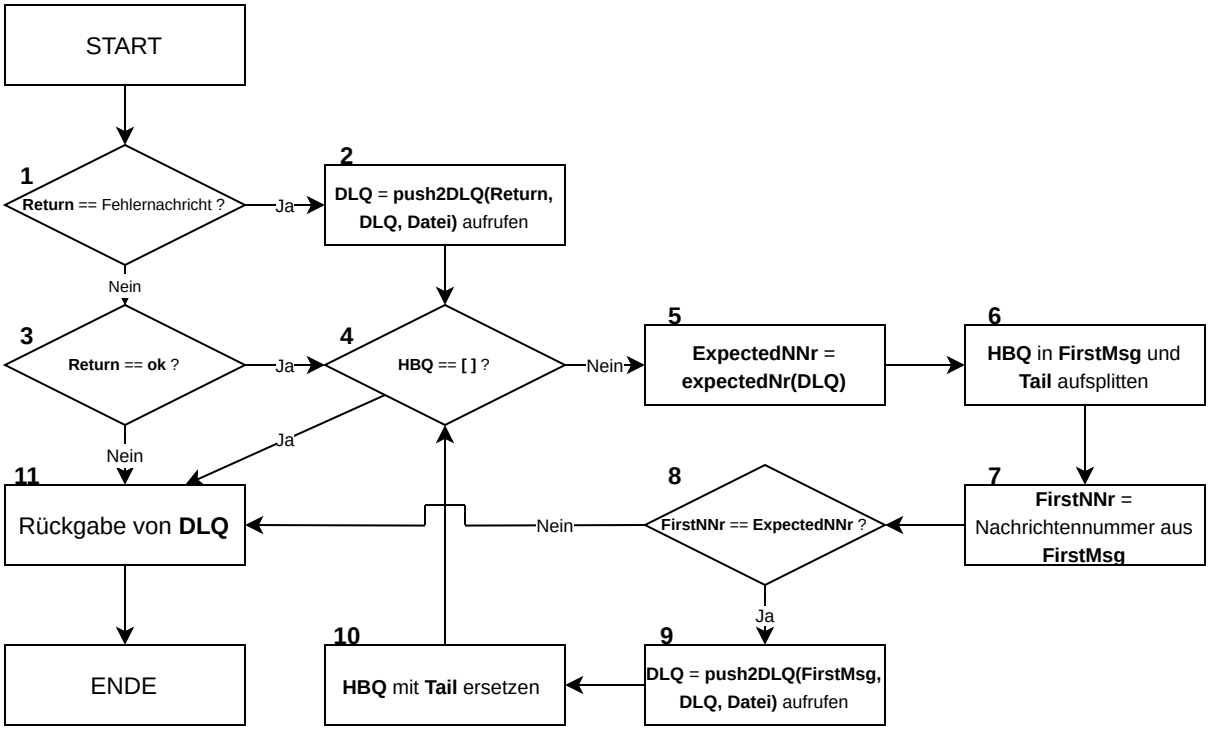
pushHBQ(HBQ, DLQ, DLQCapacity, Datei, Message = [NNr, Msg, TSclientout])

Fügt eine Nachricht in die HBQ ein. Verwirft die Nachricht, wenn ihre Nachrichtennummer bereits in der DLQ war.
Außerdem werden hier ggf. Fehlernachrichten erstellt und in die DLQ eingefügt, sollten sich zu viele Nachrichten in der HBQ befinden.



pushDLQ(Return, HBQ, DLQ, Datei)

Verschiebt so viele Nachrichten wie möglich aus der HBQ in die DLQ. Dies führt zu einem Leeren der HBQ, solange es dort keine Lücken gibt.



DLQ

In einer Datei `dlq.erl` zu finden.

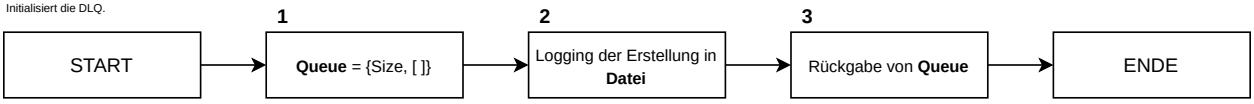
Enthält alle Nachrichten, in korrekter Reihenfolge, welche an den Client geschickt werden sollen.

DLQ: `{<Kapazität>, [<Nachricht1>, <Nachricht2>, …]}`

⇒ Nachrichten aufsteigend sortiert nach Nachrichtennummer

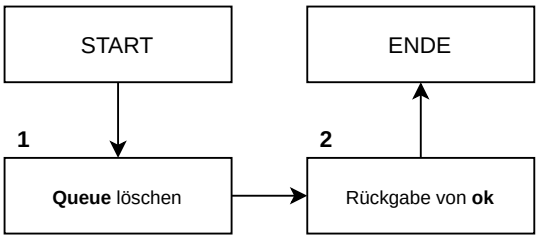
initDLQ(Size, Datei)

Initialisiert die DLQ.



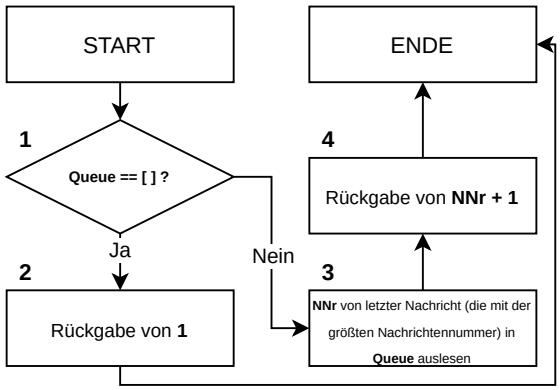
delDLQ(Queue)

Löscht die DLQ.



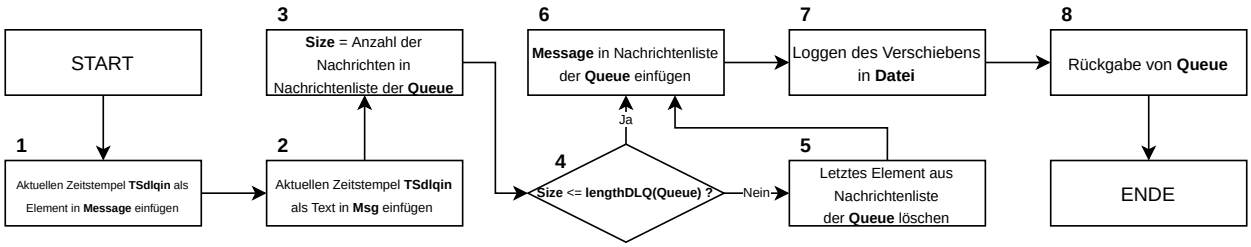
expectedNr(Queue)

Ermittelt die als nächstes erwartete Nachrichtennummer der DLQ.



push2DLQ(Message = [NNr, Msg, TSclientout, TShbqin], Queue, Datei)

Verschiebt eine Nachricht in die DLQ und ergänzt diese um einen Zeitstempel. Loggt das Verschieben in eine Datei.

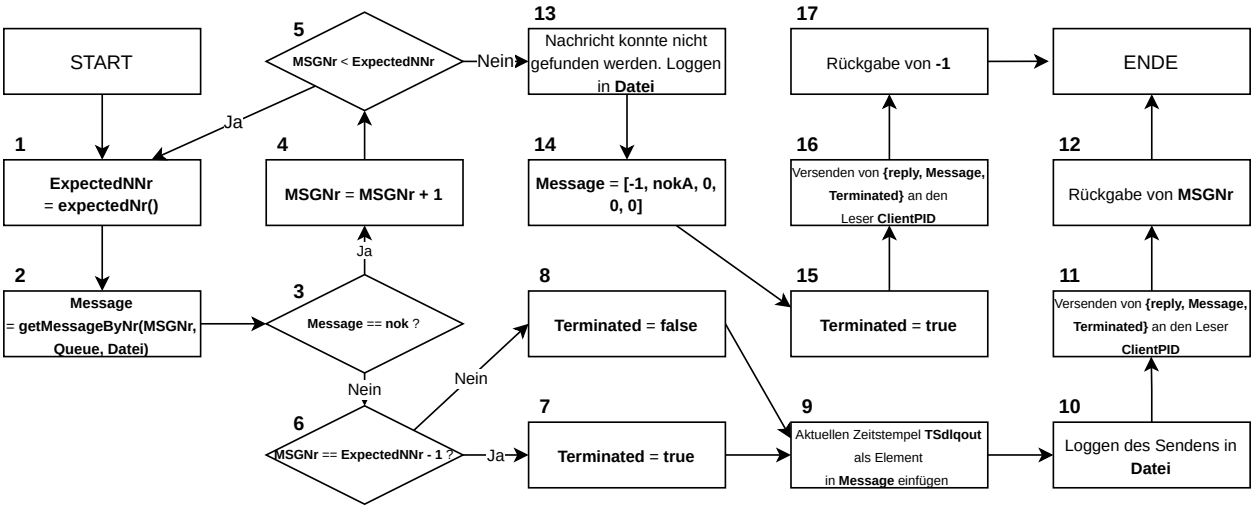


deliverMSG(MSGNr, ClientPID, Queue, Datei)

Versendet eine Nachricht an einen Leser. Ist die angefragte Nachrichtennummer nicht in der DLQ enthalten, so wird die Nachricht mit der nächstgrößeren Nummer versendet. Sollte diese ebenfalls nicht existieren, wird stattdessen eine spezielle Fehlernachricht versendet. Das passiert gdw. eine Nachricht angefragt wird, aber es garkeine Nachrichten für den Leser mehr gab.

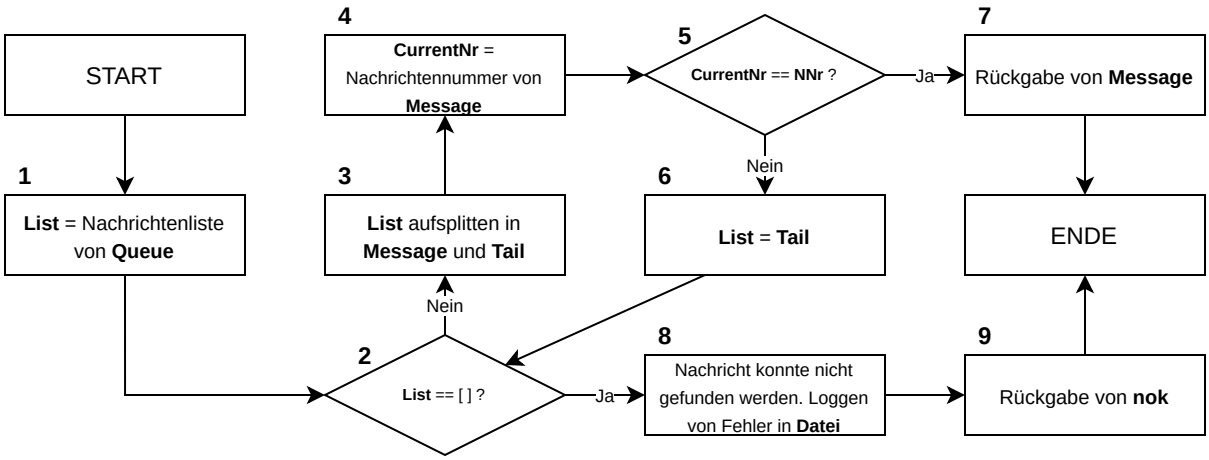
Neben dem ursprünglichen Nachrichteninhalt werden außerdem ein Ausgangszeitstempel und eine Mitteilung, ob es noch weitere Nachrichten für den Leser gibt (Terminated = true, wenn es keine Nachrichten mehr gibt) mit in die versendete Nachricht eingefügt.

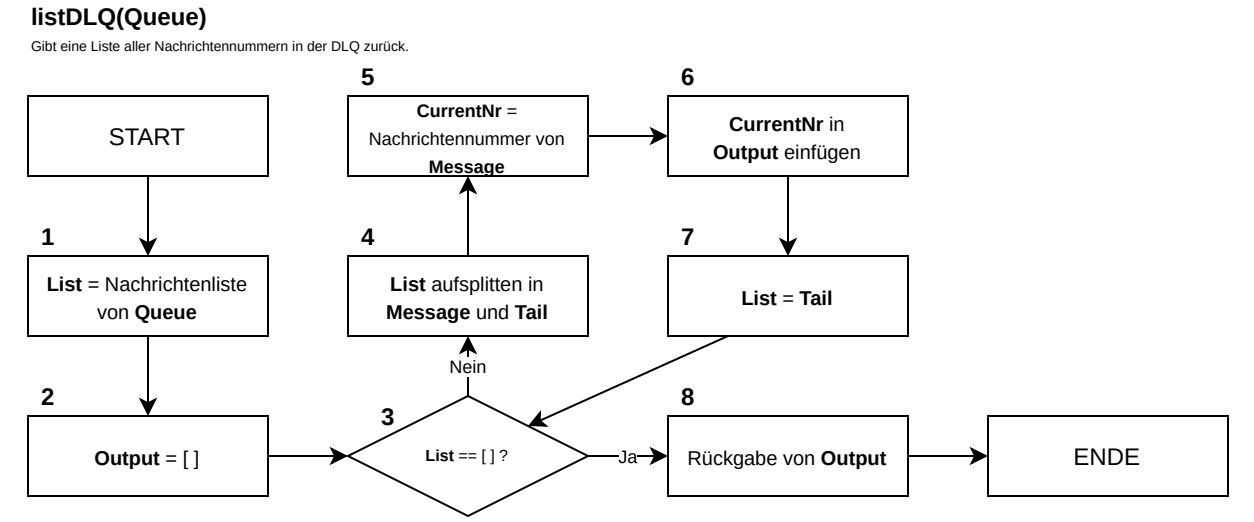
Anschließend wird die tatsächlich versendete Nachrichtennummer zurückgegeben.

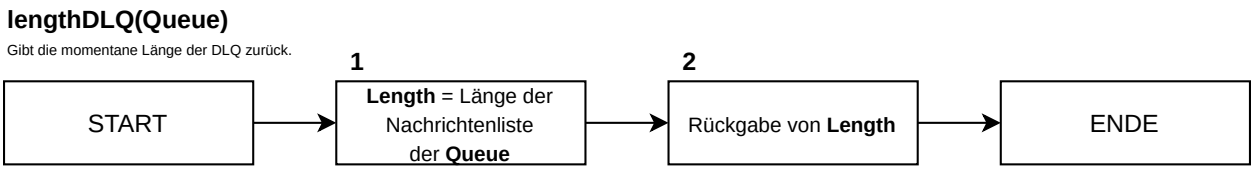


getMessageByNr(NNr, Queue, Datei)

Gibt bei Erfolg die zugehörige Nachricht zu einer angegebenen Nachrichtennummer zurück.
Sollte diese nicht gefunden worden sein, wird stattdessen "nok" zurückgegeben und der Fehler in einer Datei geloggt.







Client

In einer Datei `client.erl` zu finden. Die Konfiguration des Servers ist in der Datei `client.cfg` möglich:

Parameter	Beschreibung
`clients`	Anzahl der zu startenden Client-Prozesse.
`lifetime`	Zeit bis zum automatischen Beenden der Client-Prozesse in Sekunden.
`sendeintervall`	Zeitabstand zwischen dem Aussenden von Nachrichten.
`servername`	Name des Server-Prozesses im lokalen Namensdienst.
`servernode`	Node, über welche der Server läuft.

Der Client arbeitet in zwei verschiedenen Rollen, zwischen denen er kontinuierlich wechselt: Als Redakteur sendet er in regelmäßigen Abständen (`sendeintervall`) Nachrichten an den Server und als Leser empfängt er die von ihm und anderen Clients geposteten Nachrichten.

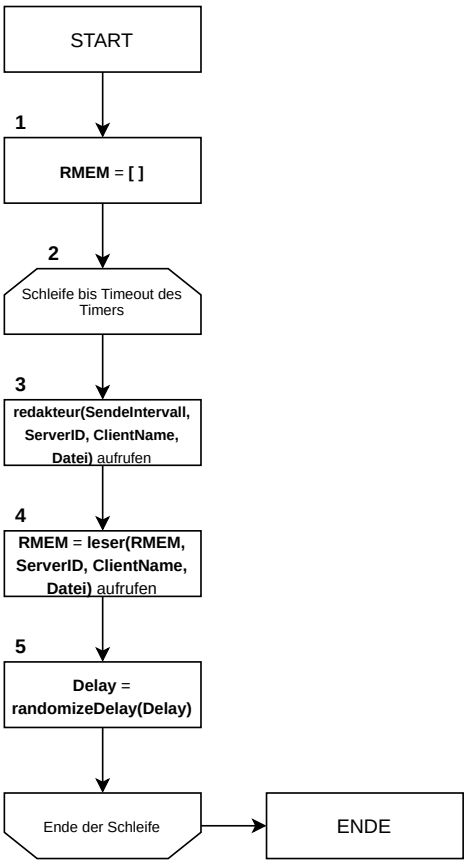
Hierbei merkt er sich, welche Nachrichten er in der Rolle des Lesers erhalten hat, um Wiederholungen feststellen zu können. Dies wird mithilfe der Datenstruktur `ReadMsgMEM`

(RMEM) realisiert.

Es werden hier außerdem mehrere Client-Prozesse gestartet (`clients`), um die Interaktion vom Server mit mehreren Clients beobachten zu können. Nach der angegebenen `lifetime` werden diese beendet.

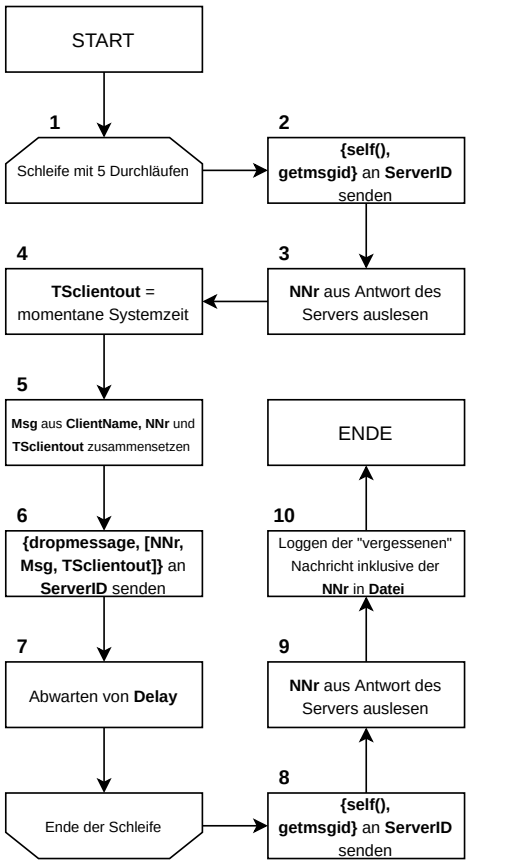
loop(LifeTime, Delay, Server, ClientName, Datei)

Wechselt kontinuierlich zwischen Redakteur und Leser, bis der Client beendet wird.
Merkt sich die bereits erhaltenen Nachrichtennummern auch zwischen den Leser-Wechseln im RMEM.



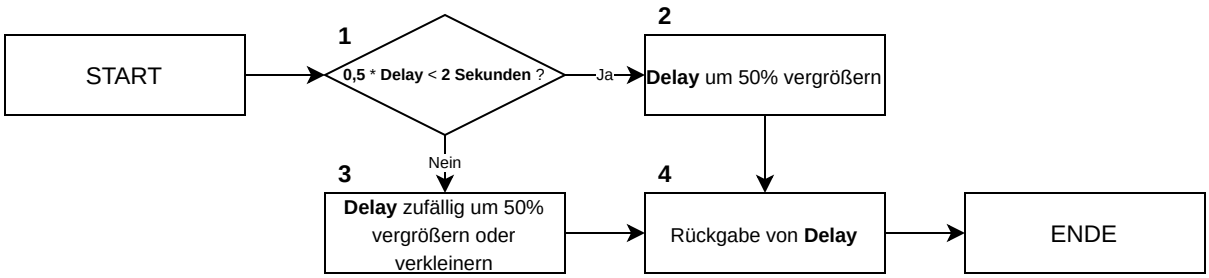
redakteur(Delay, ServerID, ClientName, Datei)

Sendet in regelmäßigen Abständen 5 Nachrichten an einen Server, fragt anschließend eine weitere Nachrichtennummer an und bricht dann ab. Die dadurch angefragte aber "vergessene" Nachricht wird in einer Datei geloggt.



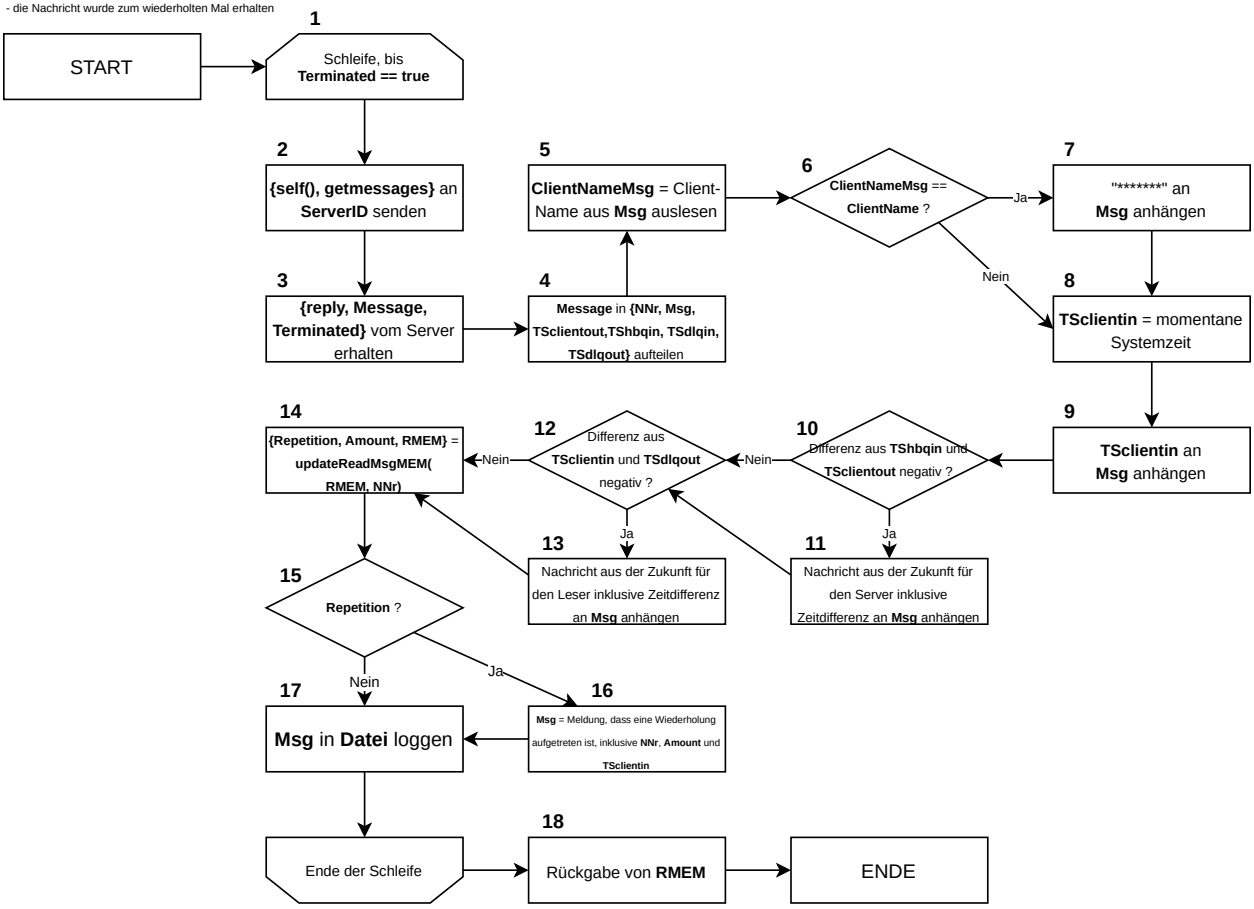
randomizeDelay(Delay)

Vergrößert oder verkleinert zufällig das übergebene Delay um 50%. Hierbei wird jedoch nie ein Minimum von 2 Sekunden unterschritten.



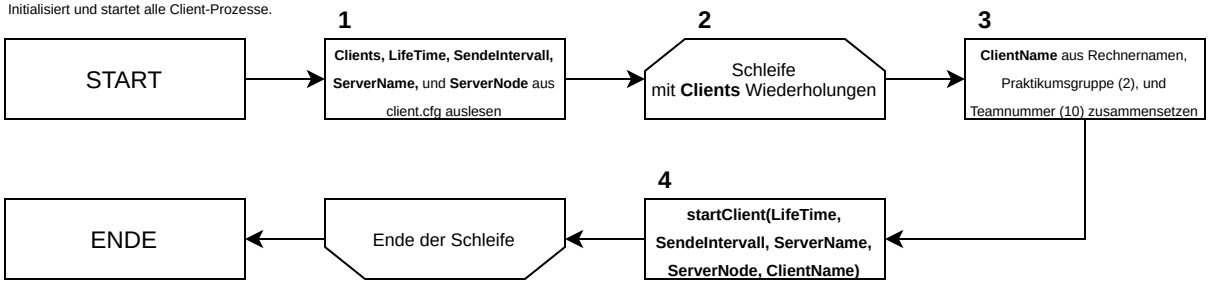
leser(RMEM, Server, ClientName, Datei)

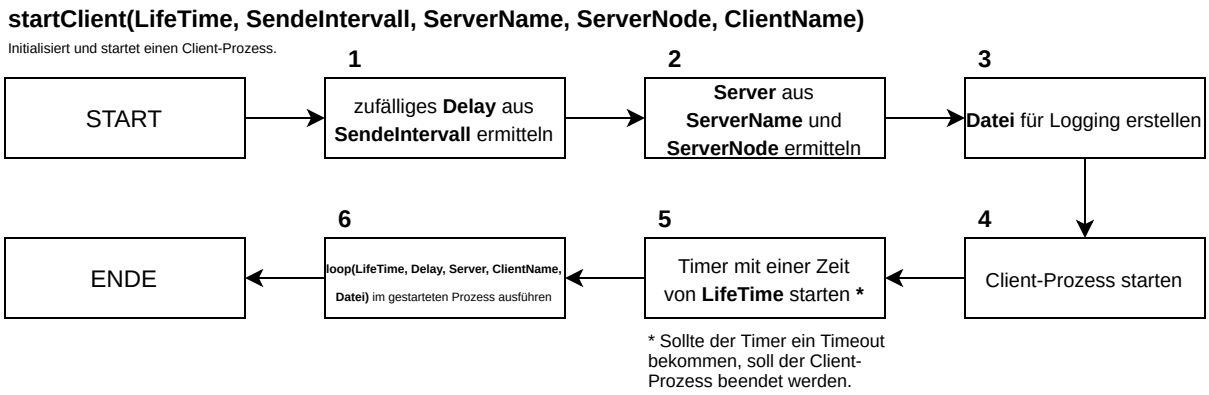
Liest Nachrichten vom Server und loggt diese in einer Datei. Außerdem werden die erhaltenen Nachrichten um Informationen zu folgenden Ereignissen ergänzt:
- die Nachricht stammt vom eigenen Redakteur ("********")
- die Nachricht kommt "aus der Zukunft"
- die Nachricht wurde zum wiederholten Mal erhalten



start()

Initialisiert und startet alle Client-Prozesse.





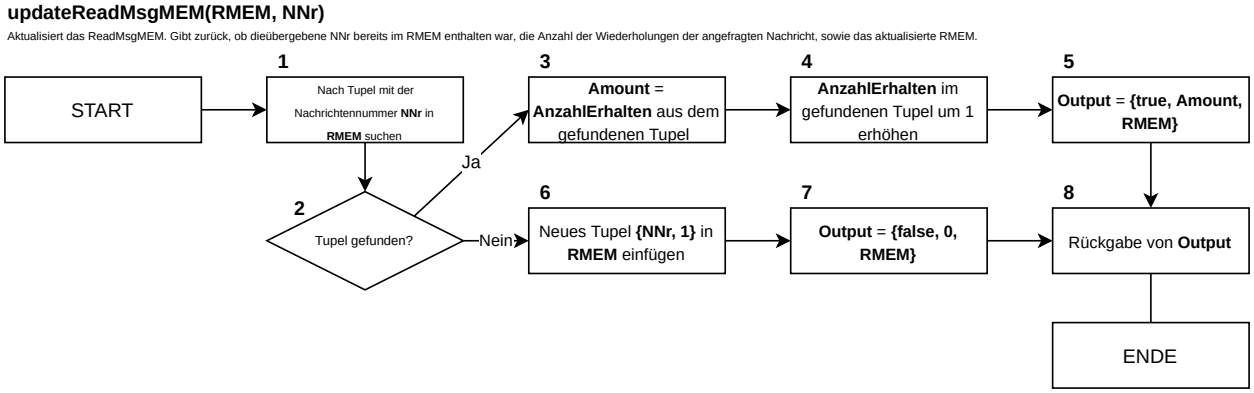
ReadMsgMEM

Ebenfalls in der Datei `client.erl` zu finden.

Enthält, wie oft der Leser eine bestimmte Nachricht vom Server erhalten hat.

ReadMsgMEM: `[{<Nachrichtennummer>, <AnzahlErhalten>}, …]`

⇒ Nachrichten aufsteigend sortiert nach Nachrichtennummer



Funktionalität Beschreibung

Konstanten

`?Xdlq` - Größe DLQ

`?Xleser` - Zeit bis zum Vergessen eines Lesers (in s)

`?Xredakteur` - Zeitabstand zwischen Nachrichtenversendungen des Redakteurs

Server

1. eindeutige Nachrichten-IDs (müssen vom Server angefragt werden) → `NNr`
2. Zwei Queues:
   - Deliveryqueue (auslieferbare Nachrichten)
   - Holdbackqueue (nicht auslieferbare Nachrichten)
3. fügt hinten/rechts folgende Infos zu allen empf. Nachrichten (`Msg`) ein:
   - Empfangszeit beim Eintrag in die HBQ (`TShbqin`)
   - Übertragungszeit beim Eintrag in die DLQ (`TSdlqin`)
     -> `vsutil:now2string(<Zeitstempel>)`
   - Zeitstempel als viertes Argument mit einfügen (`erlang:timestamp()`)
4. Anfrage vom Leser: Server verschickt noch nicht ausgelieferte Nachricht (gemäß `NNr`)
   - Flag: gibt es weitere Nachrichten für Leser in DLQ?
   - Anfrage, obwohl keine Nachrichten verfügbar sind → Antwort (dummy-Nachricht): `[-1, nokA, 0, 0, 0, 0]`
5. Leser wird nach `Xleser` Sekunden vergessen, wenn er keine neue Anfrage macht
   - anschließend behandeln wie unbekannten Leser
6. Fehlernachricht wenn HBQ zu `(2/3)*Xdlq` gefüllt:
   - "`***Fehlernachricht fuer Nachrichtennummern <X> bis <Y> um <Zeitstempel>`"
   - wird in DLQ eingefügt und ersetzt die fehlenden Nachrichten
7. terminiert, wenn die Wartezeit (`latency` in der `server.cfg`) überschritten wird, ohne dass neuer Kontakt zu ihm aufgenommen wurde
8. verwendet 3 ADTs: HBQ, DLQ, CMEM (Gedächtnis für Leser)
9. HBQ: entfernte ADT (Schnittstelle durch Nachrichtenformate beschrieben)DLQ & CMEM: lokale ADTs (Schnittstellen durch Funktionen beschrieben)
10. Konfiguration in `server.cfg`

- Server muss im lokalen Namensdienst von Erlang registriert werden (`register(<name>, ServerPid)`)

Client

Redakteur

9. sendet alle `?Xredakteur` Sekunden eine Nachricht an den Server:
   - Rechnernamen (z.B. lab18), Praktikumsgruppe (2), Teamnummer (10) → lab18210
   - Nachrichten-ID (`NNr`)
   - aktuelle Systemzeit (`TSclientout`) → `erlang:timestamp()`
   - z.B. “`lab18210: 22te_Nachricht. C Out: 02.03. 10:54:36,381`”
10. `?Xredakteur` wird nach dem Senden von 5 Nachrichten zufällig um 50% vergrößert oder verkleinert (mindestens 2 Sekunden)
11. nach 5ter gesendeter Nachricht wird erneut eine NNr beim Server angefragt, anschließend allerdings keine Nachricht mehr versendet → Lücke in HBQ entsteht
    - zu vermerken in log:
      “66te_Nachricht um <Zeitstempel>| vergessen zu senden ******”
    - anschließend wechseln in Rolle des Lesers

⇒ 5x NNr anfragen & Nachricht (siehe 9.) versenden

⇒ anschließend erneut NNr anfragen, aber keine Nachricht versenden → Lücke in HBQ

⇒ “vergessene” Nachricht loggen (siehe 11.)

⇒ `?Xredakteur` für den nächsten Durchlauf zufällig anpassen (siehe 10.)

⇒ Wechseln in Rolle des Lesers

Leser

12. fragt so lange Nachrichten beim Server ab, bis alle erhalten wurden, und loggt diese
    - eine Nachricht pro Anfrage → Anfragen werden gelooped, bis dummy-Nachricht erhalten wird

13. fügt **Nachrichten vom eigenen Redakteur** die Zeichenfolge “*******” an und **allen Nachrichten** die Eingangszeit am Ende an
    - `lab18210: 8te_Nachricht. C Out: 02.03 10:54:30,365| HBQ In: 02.03 10:54:30,383| DLQ Out: 02.03 10:54:42,431| *******; C In: 02.03 10:54:42,427|` oder
    - `lab18103: 8te_Nachricht. C Out: 02.03 10:54:30,365| HBQ In: 02.03 10:54:30,383| DLQ Out: 02.03 10:54:42,431| C In: 02.03 10:54:42,427|`
14. merkt sich permanent die vom Server erhaltenen Nachrichtennummern und gibt bei wiederholten Nachrichten die Anzahl der Wiederholungen mit an (im Log)
    - `>>>Wiederholung<<<: Nummer30 zum 1-ten mal erhalten. ; C In: 16.03 08:48:55,823|`
15. wertet Nachrichten mit `validTS/1`, `lessTS/2`, `diffTS/2`, `now2stringD/1` aus `vsutil.erl` aus:
    - bei “Nachricht aus der Zukunft” → Bemerkung und Zeitdifferenz (`diffTS/2` & `now2stringD/1`) bei Ausgabe mit anhängen
        - Server: `TSclientout` > `TShbqin`: `>**Nachricht aus der Zukunft fuer Server:00.00 00:00:00,0029999|<`
        - Leser: `TSdlqout` > `TSclientin`: `>**Nachricht aus der Zukunft fuer Leser:00.00 00:00:00,0003260|<`
16. `lifetime` aus `client.cfg` auslesen
    a. wenn Zeitlimit erreicht selbst terminieren
17. Konfiguration in `client.cfg`
18. besitzt Startfunktion (z.B. `startClient`) zum parallelen Starten

GUI

13. Server-GUI: Ausgaben in `Server<Node>.log` und `HB-DLQ<Node>.log`
14. Client-GUI: Ausgaben jeweils in `Client_<Nummer><Node>.log`

Analyse

client.cfg

- **clients:** bei mehr Clients kommen mehr Nachrichten
- **lifetime:** bei einer höheren Lifetime werden deutlich mehr Nachrichten an den Server versendet
- **sendeintervall:** wenn das SendeIntervall sehr klein ist, werden deutlich mehr Nachrichten versendet

server.cfg

- **latency:** bei zu kleiner Latency terminiert der Server vor den Clients
- **clientlifetime:** bei zu kleinen Werten für die clientlifetime treten viele Wiederholungen auf
- **dlqlimit:** bei einem kleinen DLQ-Limit werden häufiger Fehlernachrichten in die DLQ eingefügt und dann auch versendet, da die HBQ öfter Lücken füllen muss