Team: TEAM 01, Falco Winkler (FW), Daniel Schruhl (DS)

Aufgabenteilung:

- DLQ (DS)
- CMEM (DS)
- HBQ (FW)
- Server (DS)
- Client (FW)

Quellenangaben:

Bearbeitungszeitraum:

- 26.03.2017 (FW,DS)
- 03.04.2017 (DS)
- 04.04.2017 (DS)
- 05.04.2017 (FW,DS)
- 06.04.2017 (FW,DS)
- 07.04.2017 (FW,DS)

Aktueller Stand:

- DLQ fertig und getestet
- CMEM fertig und getestet
- HBQ fertig
- Server fertig und getestet
- Client fertig

Änderung des Entwurfs:

- Formatierung angepasst
- Komponentendiagramm erweitert
- Detailbeschreibungen für jedes Modul und Paket

1 Einführung und Ziele

Es soll eine Message of the Day Anwendung erstellt werden. Dabei werden von verschiedenen Clients an einen Server verschiedene Nachrichten des Tages gesendet. Die Clients rufen vom Server alle Nachrichten ab, so dass jeder Client alle Nachrichten in einer festen Reihenfolge hat.

1.1 Randbedingungen

Es soll eine Client/Server-Architektur implementiert werden. Der Server verwaltet dabei die ihm von den Clients gesendeten Nachrichten. Das beinhaltet eine feste Numerierung der Nachrichten.
Die Clients rufen dabei in bestimmten Abständen die Nachrichten ab. Falls ein Client dem Server schon bekannt ist, bekommt der nur die ihm noch unbekannten (neuen) Nachrichten.
Der Server muss sich also die Clients merken. Es soll mit einer Holdbackqueue und einer Deliveryqueue gearbeitet werden, um die korrekte Auslieferung in einer bestimmten Reihenfolge der Nachrichten zu garantieren.

1.2 Kontextbegrenzung

Das System soll in Erlang umgesetzt werden. Es muss auf Computern mit Linux Betriebssystem lauffähig sein.

2 Gesamtsystem

2.1 Bausteinsicht

Das Softwareprodukt besteht aus mehreren Modulen und Paketen (Abbildung 1). Diese Pakete setzen sich zusammen aus dem Client-Paket und dem Server-Paket.

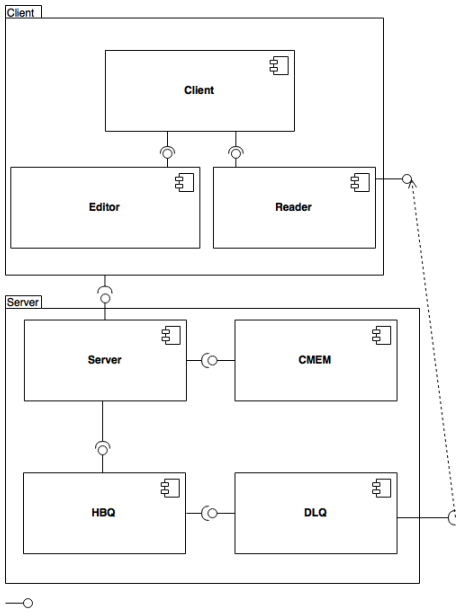

Abbildung 1: Komponentendiagramm der Message Of The Day App

Das Server Paket beinhaltet das Server-Modul und alle vom Server-Modul verwendeten Datenstrukturen.
Das Server-Modul ist für alle Funktionalitäten des Servers zuständig. Dazu gehört das Nummerieren und Verwalten der Nachrichten und die Verwaltung der Clients. Das Server-Modul benutzt daher per Schnittstelle das HBQ-Modul und das CMEM-Modul. Das Server-Modul stellt eine Schnittstelle für die Clients bereit.

Das CMEM-Modul ist für die Speicherung der Clients und ihrer aktuellen zu erwartenden Nachrichten Nummer zuständig. Das CMEM-Modul soll als lokale ADT realisiert werden. Diese wird nur vom Server angesprochen.

Das HBQ-Modul ist für die Holdbackqueue zuständig und regelt die Sortierung der einkommenden Nachrichten in die Deliveryqueue und der damit verbundenen Fehlerbehandlung. Dabei ist das HBQ-Modul als entfernte ADT realisiert. Diese wird nur von dem Server-Modul verwendet.

Das DLQ-Modul realsiert die Deliveryqueue, die für die Auslieferung der Nachrich-

ten in Reihenfolge an die Clients zuständig ist. Die Schnittstelle des DLQ-Moduls wird nur vom HBQ-Modul konsumiert.

Das Client-Paket beinhaltet die Client-Module. Diese sind zum einen der Lese Client (Reader-Modul) und der Redakteur Client (Editor-Modul). Beide Clients sind als ein Prozess implementiert und verwenden die vom Server bereitgestellte Schnittstelle. Der Redakteur Client ist für das Schicken von Nachrichten zuständig und der Lese Client für das Lesen von Nachrichten.

2.2 Laufzeitsicht

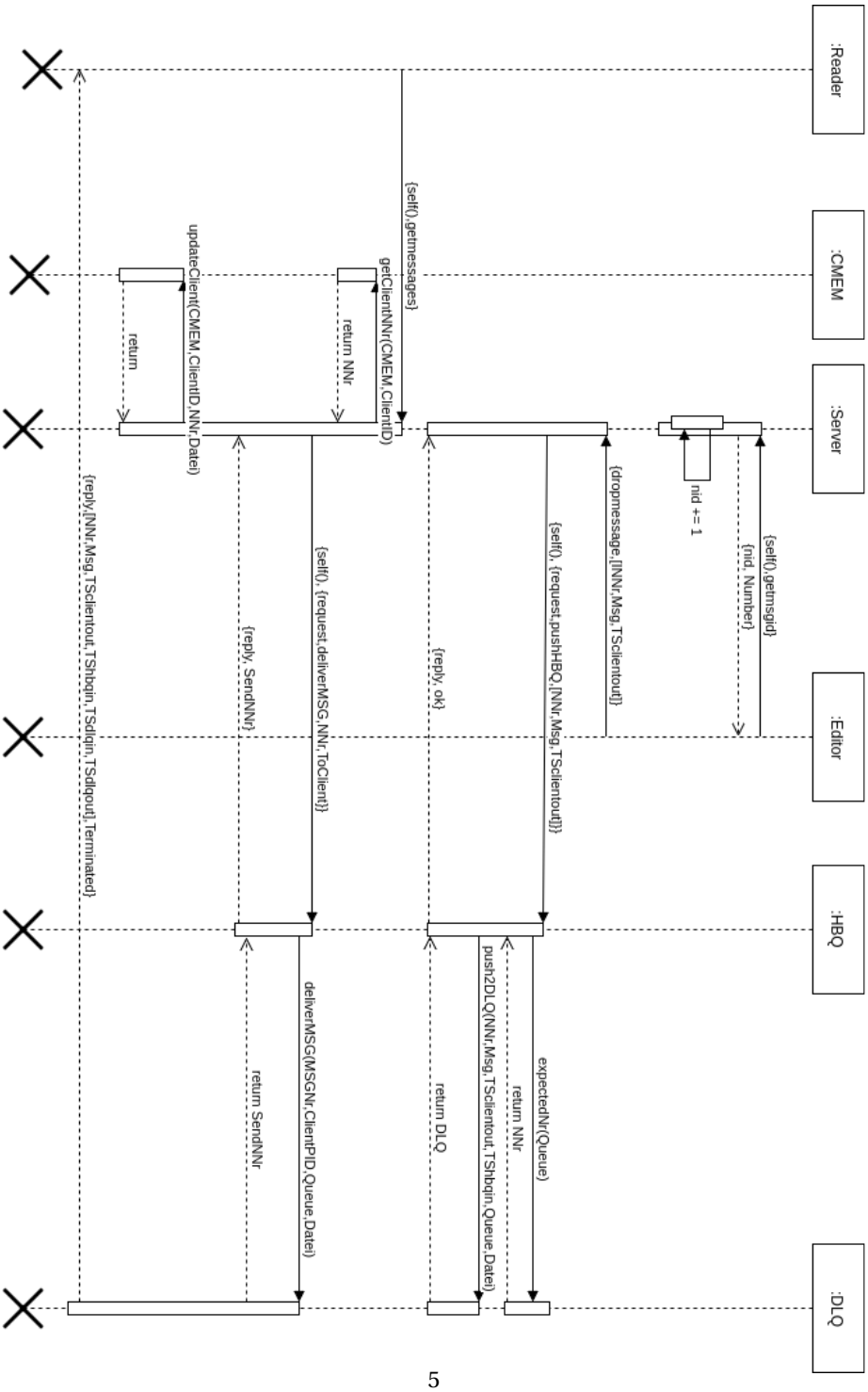

Abbildung 2: Sequenzdiagramm bei fehlerfreiem Nachrichtenaustausch

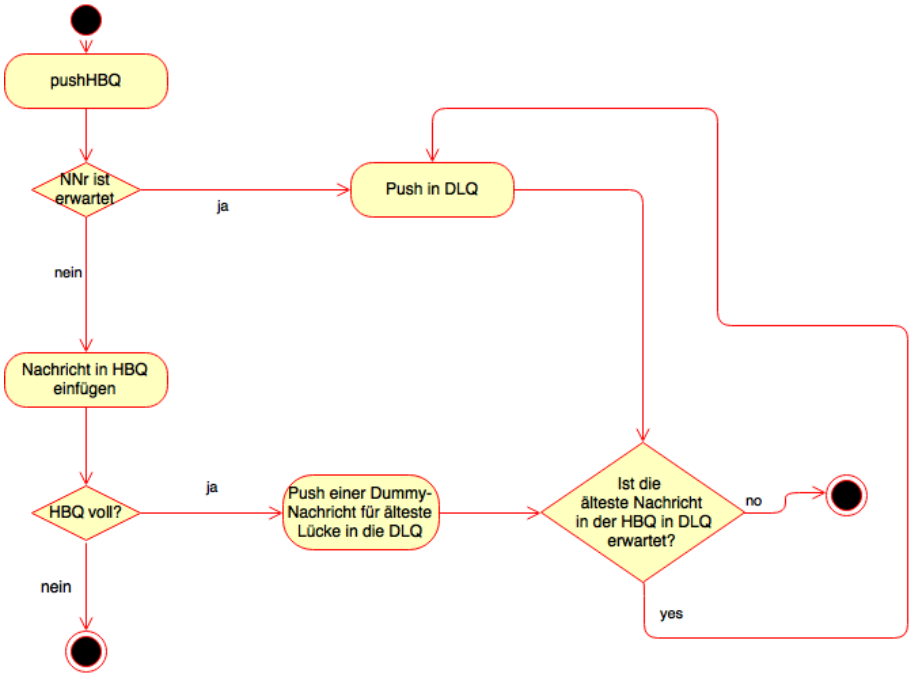

Abbildung 3: Aktivitätsdiagramm beim Hinzufügen von Nachrichten in die HBQ

3 Subsysteme und Komponenten

Das Server Modul und das Client Modul sind über Konfigurationsdateien konfigurierbar. Der Server, der Client und die HBQ stellen Prozesse dar, die nebenläufig laufen. Dabei schreiben sie Log-Dateien, in denen ihre jeweiligen Aktionen protokolliert werden.

3.1 Client-Modul

3.1.1 Aufgabe und Verantwortung

Das Client-Modul stellt einen Client in einem Prozess dar. Es kann eingestellt werden, wie viele Clients auf einmal gestartet werden und wie lange diese Clients existieren.
Clients bestehen aus einem Leser (Reader) und einem Redakteur (Editor). Diese Rollen werden sequentiell im Laufe des Lebenszyklus des Clients immer abwechselnd durchgetauscht. Das Client-Modul koordiniert also auch die Rollen des Clients.

3.1.2 Schnittstelle

```
/* Startet einen Client mit gegebener Lifetime */
startClient(Lifetime): Integer -> PID

/* Startet eine feste Anzahl an Clients (Config) */
startClients()
```

startClient(Lifetime): Startet einen Client als Prozess, der zwischen Redakteur und Leser wechselt. Terminiert nach ablauf der Lifetime.

startClients(): Startet eine feste Anzahl an Clients. Die Anzahl wird aus der Config Datei herausgelesen.

3.1.3 Entwurdsentscheidungen

Beim Starten eines Client-Prozesses wird gleichzeitig ein Timer mit der Lifetime pro Prozess gestartet, der dem Client in seiner jeweiligen Rolle eine Terminierungsnachricht schickt. Dadurch wird der Client nach dem Ablaufen der Lebenszeit heruntergefahren. Ansonsten wird die jeweilige Rolle des Clients durchgewechselt.

3.1.4 Konfigurationsparameter

- Anzahl der Clients
- Lebensdauer eines Clients
- Initiales Sendeintervall
- Name des Servers

3.2 Reader-Modul

3.2.1 Aufgabe und Verantwortung

Das Reader-Modul übernimmt das Anfordern und Empfangen von Nachrichten wenn sich der Client im lesenden Zustand befindet.

3.2.2 Schnittstelle

start_reading(MoreFlag, Logfile, ReaderNNrs, ServerPID)
Der Parameter MoreFlag wird vom Server zurückgeschickt. Er zeigt an, ob es keine weiteren Nachrichten mehr für den Reader gibt.
ReaderNNrs enthalten die vom Editor gesendeten Nachrichtennummern. Diese Nachrichtennummern sollen der Reihe nach vom Reader angefragt werden
Der Reader-Client sendet seine Anfragen an den Server mit der PID ServerPID.

3.2.3 Entwurfsentscheidungen

Innerhalb des Reader - Moduls gibt es eine Methode zum Formatieren und zum Loggen einer empfangenen Nachricht. Die Funktion zum Anfragen und Empfangen von Nachrichten arbeitet rekursiv: Sie ruft sich so lange selbst auf, bis getmessages das flag true zurückliefert. Das Reader-Modul ist eine ADT und es wird nur vom Client angesprochen.
Der Reader terminiert wenn:

- Er im MoreFlag mitgeteilt bekommt, dass keine weiteren Nachrichten vorliegen
- Er von dem Client-Modul terminiert wird

3.3 Server-Modul

3.3.1 Aufgabe und Verantwortung

Der Server hat die Aufgabe Nachrichten von Clients entgegen zu nehmen, diese zu verarbeiten und an seine Subprozesse bzw. an seine an ihn verbundenen Module weiter zu schicken. Der Server ist außerdem die zentrale Stelle für die Koordinierung der nächst höchsten freien Nachrichtennummer, die von den Clients zum Senden verwendet werden soll. Clients holen sich über den Server Nachrichten. Der Server reicht diese Anfragen an die relevanten Subprozesse weiter.
Wenn der Server für eine bestimmte Zeit von keinem Client mehr angesprochen wird, soll er sich und seine Subprozesse herunterfahren.

3.3.2 Schnittstelle

```
/* Abfragen einer Nachricht */
Server ! {self(), getmessages},
receive {reply, [NNr,Msg, TSclientout, TShbqin, TSdlqin, TSdlqout], Terminated}
```

```
/* Senden einer Nachricht */
Server ! {dropmessage, [INNr, Msg, TSclientout]},

/* Abfragen der eindeutigen Nachrichtennummer */
Server ! {self(), getmsgid}

/* Nur fuer interne Prozesse: Einleiten des Herunterfahrens */
Server ! terminate

/* Nur fuer interne Prozesse: Terminierungsprozess erfolgreich, fahre runter*/
Server ! {reply, ok}
```

getmessages: Fragt beim Server eine aktuelle Textzeile ab. self() stellt die Rückrufadresse des Leser-Clients dar. Als Rückgabewert erhält er eine für ihn aktuelle Textzeile (Zeichenkette) zugestellt (Msg) und deren eindeutige Nummer (NNr). Zudem erhält er die Zeitstempel explizit (erstellt durch erlang:now(), TSclientout, TShbqin, TSdlqin, TSdlqout). Mit der Variablen Terminated signailiert der Server, ob noch für ihn aktuelle Nachrichten vorhanden sind. Terminated == false bedeutet, es gibt noch weitere aktuelle Nachrichten, Terminated == true bedeutet, dass es keine aktuellen Nachrichten mehr gibt, d.h. weitere Aufrufe von getmessages sind nicht notwendig.

dropmessage: Sendet dem Server eine Textzeile (Msg), die den Namen des aufrufenden Clients und seine aktuelle Systemzeit sowie ggf. irgendeinen Text beinhaltet, zudem die zugeordnete (globale) Nummer der Textzeile (INNr) und seine Sendezeit (erstellt mit erlang:now(), TSclientout).

getmsgid: Fragt beim Server die aktuelle Nachrichtenummer ab. self() stellt die Rückrufadresse des Redakteur-Clients dar. Als Rückgabewert erhält er die aktuelle und eindeutige Nachrichtennummer (Number).

terminate: Nur von internen Prozessen zu verwenden. Startet den Terminierungsprozess.

reply, ok: Nur von internen Prozessen zu verwenden. Signalisiert, dass alle Submodule und Prozesse des Servers heruntergefahren wurden und der Server nun selber herunterfahren kann.

3.3.3 Entwurfsentscheidungen

Der Server wird mit einer Start Funktion hochgefahren. Diese Funktion startet den Server-Prozess und konfiguriert ihn mit den Werten, die in der Config stehen. Außerdem initialisiert der Server dabei eine CMEM (interne ADT) und startet den Prozess der HBQ (externe ADT als Prozess) und initalisiert diese.
Der Server-Prozess hat einen State. In diesem State wird die CMEM, die Config, eine Verbindung zur HBQ (ProzessID), die nächste Nachrichtennummer für einen Client

zum Senden, ein Timer und die Latenz gespeichert.
Der Timer ist so eingestellt, dass er nach dem Ablaufen der Latenzzeit dem Server benachrichtigt sich herunterzufahren (terminate). Dabei werden die Subprozesse heruntergefahren bevor der Server herunterfährt.
Jedes mal, wenn ein Client den Server über seine Schnittstelle anspricht, wird der Timer abgebrochen und zurück auf die initiale Latenzzeit gestellt.
Bevor Clients dem Server eine Nachricht schicken können, müssen sie vom Server eine aktuelle Nachritennummer abfragen, die sie für die neue Nachricht verwenden können. Diese Nummer ist im State des Server gespeichert und wird dann an den anfragenden Client übermittelt. Danach wird diese Nummer im State inkrementiert.
Eingehende Nachrichtenzeilen von Clients werden vom Server an die HBQ weitergeleitet. Dabei wird dem Nachrichtentext der aktuelle Zeitstempel hinzugefügt.
Wenn ein Client nach Nachrichten am Server anfragt, wird im CMEM nachgeschaut welche Nachrichtennummer für den Client als nächstes vorgesehen wird und die HBQ beauftragt, diese dem Client zu schicken. Die dabei zurück gegeben verschickte Nachrichtennummer wird dann im CMEM für den Client geupdated. Falls jedoch eine dummy Nachricht zurück kommt (Nachrichtennummer: -1) findet dieses Update nicht statt.

3.3.4 Konfigurationsparameter

- Latenzzeit bestimmt die maximale Zeit die der Server ungenutzt läuft
- Lebenszeit des Clients bestimmt die Zeit, für die sich der Server einen Client merkt
- Namen für den Server Prozess
- Größe der Deliveryqueue

3.4 CMEM-Modul

3.4.1 Aufgabe und Verantwortung

Dieses Modul hat die Aufgabe sich anfragende Clients, ihre letzte bekommene Nachrichtennummer und den letzten Zeitpunkt ihrer Anfrage zu merken. Clients, die sich seit einiger Zeit (Konfigurationsparameter) nicht gemeldet haben, werden vom CMEM als neue Clients behandelt (vergessen).
Die gespeicherten Daten sind über die Schnittstelle des Moduls vom Server aus abrufbar.

3.4.2 Schnittstelle

```
/* Initialisieren des CMEM */
initCMEM(RemTime, Datei): Integer X Atom -> CMem

/* Speichern/Aktualisieren eines Clients in dem CMEM */
updateClient(CMEM, ClientID, NNr, Datei): Cmem X PID X Integer X Atom -> CMem

/* Abfrage welche Nachrichtennummer der Client als naechstes erhalten darf */
getClientNNr(CMEM, ClientID) : Cmem X PID -> Integer
```

initCMEM(RemTime,Datei): initialisiert den CMEM. RemTime gibt dabei die Zeit an, nach der die Clients vergessen werden Bei Erfolg wird ein leeres CMEM zurück geliefert. Datei kann für ein logging genutzt werden.

updateClient(CMEM,ClientID,NNr,Datei): speichert bzw. aktualisiert im CMEM den Client ClientID und die an ihn gesendete Nachrichtenummer NNr. Datei kann für ein logging genutzt werden.

getClientNNr(CMEM,ClientID): gibt die als nächstes vom Client erwartete Nachrichtennummer des Clients ClientID aus CMEM zurück. Ist der Client unbekannt wird 1 zurück gegeben.

3.4.3 Entwurfsentscheidungen

Die CMEM wird mit Hilfe einer Liste realisiert. An erster Stelle der CMEM Liste steht die Liste der Clients. Die Elemente der Client Liste sind Tupel, die aus der Client Prozess ID, der zuletzt erhaltenen NNr und einem Timestamp der letzten Aktion in Millisekunden bestehen.
An zweiter Stelle der CMEM Liste steht die maximale Zeit, für die ein Client gemerkt wird in Millisekunden.

```
/* Client Tupel Format */
Client := {ClientPID, NNr, ClientTS}:
    {PID X Integer X Integer}

/* CMEM Format */
```

```
/* ClientList ist Liste bestehend aus Clients */
/* RemTime ist Zeit in Millisekunden fuer die Clients gemerkt werden */
CMEM := [ClientList, RemTime]: [List X Integer]
```

Falls beim Aktualisieren eines Clients der Client noch nicht im CMEM steht, wird er hinzugefügt. Ansonsten wird er einfach mit den angegebenen Parametern in der CMEM Client Liste aktualisiert.

Wenn die nächste Nachrichtennummer für einen Client abgerufen wird (`getClientNNr`), wird ein Check gemacht, ob der Client bekannt ist. Ein Client ist bekannt, wenn er in der CMEM Liste steht und wenn die Summe seines Timestamps mit der RemTime größer gleich der aktuellen Zeit (als Timestamp) ist.

Für bekannte Clients wird die resultierende nächste Nachrichtennummer inkrementiert. Für unbekannte Clients wird 1 als nächste Nachrichtennummer zurück gegeben.

3.4.4 Konfigurationsparameter

- Zeit nach der die Clients vergessen werden in Millisekunden (RemTime)

3.5 HBQ-Modul

3.5.1 Aufgabe und Verantwortung

Von Redakteur-Clients gesendete Nachrichten werden hier zwischengespeichert falls sie noch nicht erwartet werden (nicht in Reihenfolge). Erwartete Nachrichten werden weitergeleitet, und es wird eine Fehlerbehandlung bei Überfüllung vorgenommen. Dadurch sollen Lücken zwischen Nachrichten durch eine Nachricht gefüllt werden. Das kann passieren, wenn z.B. Nachrichten nicht ankommen oder verloren gehen.

3.5.2 Schnittstelle

```
/* Initialisieren der HBQ */
HBQ ! {self(), {request,initHBQ}}
receive {reply,ok}

/* Speichern einer Nachricht in der HBQ */
HBQ ! {self(), {request,pushHBQ,[NNr,Msg,TSclientout]}}
receive {reply,ok}

/* Abfrage einer Nachricht */
HBQ ! {self(), {request,deliverMSG,NNr,ToClient}}
receive {reply,SendNNr}

/* Terminierung der HBQ */
HBQ ! {self(), {request,dellHBQ}}
receive {reply, ok}
```

request,initHBQ: initialisiert die HBQ und die DLQ. Bei Erfolg wird ein ok gesendet.

request,pushHBQ,[NNr,Msg,TSclientout]: fügt eine Nachricht Msg (Textzeile) mit Nummer NNr und dem Sende-Zeitstempel TSclientout (mit erlang:now() erstellt) in die HBQ ein. Bei Erfolg wird ein ok gesendet.

request,deliverMSG,NNr,ToClient: beauftragt die HBQ über die DLQ die Nachricht mit der Nummer NNr (falls nicht verfügbar die nächst höhere Nachrichtennummer) an den Client ToClient (als PID) auszuliefern. Bei Erfolg wird die tatsächlich gesendete Nachrichtennummer SendNNr gesendet.

request,dellHBQ: terminiert den Prozess der HBQ. Bei Erfolg wird ein ok gesendet.

3.5.3 Entwurfsentscheidungen

Die HBQ interne Datenstrukur zur Speicherung der Nachrichten ist eine Liste, die von vorne nach hinten aufsteigend sortiert ist. Sie stellt eine Queue dar. Die HBQ hat als State die tatsächliche HBQ, die DLQ und die Config Einstellungen. Die HBQ

wird als externe ADT mit einem eigenen Prozess abgebildet.

Beim push in die HBQ wird folgende Fallunterscheidung vorgenommen.
Im Fall dass die an die HBQ gepushte Nachricht der erwarteten NNr der DLQ entspricht wird:

1. Die Nachricht an die DLQ direkt weitergeleitet, und nicht gespeichert.
2. Die HBQ wird auf Nachrichten geprüft die nun gesendet werden können.

Im anderen Fall wird die Nachricht nur in der HBQ gespeichert, und nichts weiter getan. So werden alle von der DLQ erwarteten Nachrichten direkt zugestellt.
Wenn Nachrichten der HBQ zugefügt werden, wird danach eine Fehlerbehandlung eingeleitet. Der Fehlerfall wird durch den HBQ State: `HBQGröße >= (2/3) * Maximale DLQ Größe` beschrieben.

Nach jedem Einfügen wird die Größe der HBQ geprüft. Im Falle einer Überfüllung wird eine Dummynachricht zum Schließen einer Lücke in den Sequenznummern direkt in die DLQ gepusht. Danach wird die HBQ auf weitere Nachrichten, die nun gesendet werden können geprüft.
Die Dummynachrichten schließen Lücken bestehend aus einer oder mehrerer Sequenznummern. Eine Dummynachricht wird durch den String Fehlernachrichtïm Inhalt identifiziert. Ihre NNr wird auf das Ende des Intervalls, welches Sie abdeckt festgelegt.
Die HBQ-Liste wird nach jedem Einfügen anhand der NNr Sortiert, um die Reihenfolge zu erhalten. Dies ist nötig, um die erste Lücke in Sequenznummern zu finden. Außerdem ist so das Senden der ältesten Nachricht einfacher.

3.5.4 Konfigurationsparameter

- Die Maximalgröße der DLQ wird verwendet um die HBQ bei kritischer Überfüllung zu leeren

3.6 DLQ-Modul

3.6.1 Aufgabe und Verantwortung

Die Deliveryqueue hat die Aufgabe Nachrichten an Clients zuzustellen und stellt eine Datenstruktur dar, die eine maximale Menge (Kapazität) an Nachrichten hält. Sie verhält sich dabei wie von einer Warteschlange zu erwarten beim Einfügen und Herausholen von Nachrichten (FIFO). Nur das HBQ-Modul darf auf das DLQ-Modul zugreifen. Das DLQ-Modul sendet über eine Schnittstelle der Clients die Nachrichten an die Clients.
In der DLQ sind die Nachrichten absteigend von vorne nach hinten sortiert.

3.6.2 Schnittstelle

```
/* Initialisieren der DLQ */
initDLQ(Size,Datei): Integer X Atom -> DQueue

/* Abfrage welche Nachrichtennummer in der DLQ gespeichert werden kann */
expectedNr(Queue) : DQueue -> Integer

/* Speichern einer Nachricht in der DLQ */
push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei) :
    MSG_list X DQueue X Atom -> DQueue

/* Ausliefern einer Nachricht an einen Leser-Client */
deliverMSG(MSGNr,ClientPID,Queue,Datei):
    Integer X PID X DQueue X Atom -> Integer
```

initDLQ(Size,Datei): initialisiert die DLQ mit Kapazität Size. Bei Erfolg wird eine leere DLQ zurück geliefert. Datei kann für ein logging genutzt werden.

expectedNr(Queue): liefert die Nachrichtennummer, die als nächstes in der DLQ gespeichert werden kann. Bei leerer DLQ ist dies 1.

push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei): speichert die Nachricht [NNr,Msg,TSclientout,TShbqin] in der DLQ Queue und fügt ihr einen Eingangszeitstempel an (einmal an die Nachricht Msg und als expliziten Zeitstempel TSdlqin mit erlang:now() an die Liste an. Bei Erfolg wird die modifizierte DLQ zurück geliefert. Datei kann für ein logging genutzt werden.

deliverMSG(MSGNr,ClientPID,Queue,Datei): sendet die Nachricht MSGNr an den Leser-Client ClientPID. Dabei wird ein Ausgangszeitstempel TSdlqout mit erlang:now() an das Ende der Nachrichtenliste angefügt. Sollte die Nachrichtennummer nicht mehr vorhanden sein, wird die nächst größere in der DLQ vorhandene Nachricht gesendet. Bei Erfolg wird die tatsächlich gesendete Nachrichtennummer zurück geliefert. Datei kann für ein logging genutzt werden.

3.6.3 Entwurfsentscheidungen

Die Deliveryqueue wird als Liste realisiert. Das erste Element der Liste ist dabei eine Liste der Nachrichten und das zweite Element der Liste gibt die Kapazität der DLQ an.

```
/* Nachrichten Format */
/* minimal 3 Elemente, pro Station kommt eins hinzu; maximal 6 Elemente */
MSG_List := [NNr,Msg,TSclientout,TShbqin,TSdlqin]:
    [Integer X String X 3-Tupel X 3-Tupel X 3-Tupel]

/* DLQ Format */
/* Nachrichtenliste ist eine Liste aus MSG_List und hat */
/* maximal Kapazitaet Anzahl an Elementen */
DLQ := [Nachrichtenliste, Kapazitaet]: [List X Integer]
```

Neue Nachrichten werden am Anfang der Liste angefügt. Das vereinfacht die Nachfrage nach der nächsten zu speichernden Nachrichtennummer in der DLQ. Das ist die Nachrichtennummer der ersten Nachricht in der DLQ um eins inkrementiert.
Demnach werden Nachrichten am Ende der Liste entnommen. Wenn eine Nachricht in die DLQ eingetragen wird, wird ans Ende der Nachricht der aktuelle Zeitstempel zum Zeitpunkt des Einfügens hinzugefügt.
Beim Eintragen wird außerdem überprüft, ob die DLQ schon voll ist. Wenn die DLQ voll ist, wird die Nachricht am Ende der DLQ gelöscht und die neue DLQ danach eingefügt (FIFO).

Beim Senden einer Nachricht durch Angabe der Nachrichtennummer wird in der DLQ nach der Nachricht mit der Nachrichtennummer oder der nächst höchsten Nachrichtennummer gesucht.

Das finden der richtigen Nachricht übernimmt eine Hilfsfunktion. Diese bekommt die Nachrichtenliste der DLQ in umgekehrter Reihenfolge. Dadurch hat diese Hilfsfunktion die Nachrichtenliste der DLQ in aufsteigender Reihenfolge.
Nun sucht die Hilfsfunktion innerhalb dieser Liste das erste vorkommen einer Nachricht mit der Nachrichtennummer größer gleich der gewünschten Nachrichtennummer (von links nach rechts). Dadurch wird entweder die Nachricht mit der angefragten Nachrichtennummer gefunden oder die Nachricht mit der nächst höchsten Nachrichtennummer.
Falls überhaupt keine passende Nachricht gefunden wird (DLQ leer / Nachrichtennummer zu groß), wird eine dummy Nachricht zurückgegeben, die eine -1 als Nachrichtennummer hat. Dadurch kann sie als dummy Nachricht identifiert werden.

Die aus der Hilfsfunktion resultierende Nachricht wird mit einem Zeitstempel versehen. Beim Senden dieser Nachricht wird die Nachricht zusammen mit einem boolean Flag an den Client gesendet. Dieses boolean Flag signalisiert, ob keine weiteren Nachrichten folgen.

Dieses Flag ist wahr, wenn entweder eine Fehlernachricht vorliegt oder die Nachricht mit der Nachrichtennumer des ersten Elementes der DLQ.

3.6.4 Konfigurationsparameter

- Kapazität der DLQ als ganze Zahl