Team: 03, Michael Müller & Arne Thiele

Aufgabenaufteilung:

1. Alle Aufgaben wurden in enger Zusammenarbeit bearbeitet.

Quellenangaben: <Angabe aller genutzten Quellen>

Bearbeitungszeitraum:

Was?	**Wer?**	**Wann?**	**Stunden?**
Grundzüge der Anforderungen und Einarbeitung	Beide	09.04.17	Je 5,5
Verfeinerung des Entwurfs	Beide	12.04.17	Je 3
Implementation	Beide	15.04.17	Je 6
Implementation	Beide	16.04.17	Je 4
Tests und Behebung der Fehler anhand der Logs	Beide	17.04.17	Je 8
Tests und Behebung der Fehler anhand der Logs	Beide	18.04.17	Je 6
Praktikumstermin und Debugging	Beide	19.04.17	Je 5
Behebung der Fehler anhand der Logs	Thiele	21.04.17	1
Letzter Testlauf und Abgabe fertig gemacht	Müller	22.04.17	0,5

Aktueller Stand: Die Implementation ist vollständig erfolgt.

Änderungen des Entwurfs: Nach neuem Verständnis wurden im Entwurf die Rückgabewerte im CMEM angepasst und eine Erläuterung zum Rückgabewert in der HBQ angelegt. Die Änderungen sind rot markiert.

Entwurf:

Funktionale Anforderungen an das System:

Server:

1. Der Server verwaltet ID's (beginnend bei 1), welche Textzeilen eindeutig zugeordnet werden können. Nach Außen bietet der Server eine Schnittstelle an welche die aktuelle ID liefert, welche von Redakteuren abgerufen werden kann.
2. Die dem Server zugestellten Textzeilen sollen bzgl. der Nummerierung in zusammenhängender Reihenfolge erscheinen. Da Nachrichten durchaus verloren gehen können oder in falscher Reihenfolge beim Server ankommen können, arbeitet der Server intern mit einer Holdback- und einer Deliveryqueue (Abk.: HBQ & DLQ). Eine Nachricht ist entweder in der DLQ oder HBQ!
3. In der **DLQ** stehen Nachrichten, welche an Leser ausgeliefert werden können. In der DLQ können nur eine bestimmte Anzahl an Textzeilen stehen, welche durch die Größe der DLQ festgelegt wird.

4. In der **HBQ** stehen alle Nachrichten, welche noch nicht ausgeliefert werden dürfen (noch nicht vollständig, falsche Reihenfolge, etc. ?).
5. **Bei Empfang einer neuen Nachricht** wird die Empfangszeit beim Eintrag in die HBQ und die Übertragungszeit beim Eintrag in die DLQ der Zeichenkette der neuen Nachricht hinzugefügt. Die Zeiten können anhand von **wergzeug:timeMilliSecond()** bestimmt werden. Zusätzlich werden der Nachrichten-Liste diese Zeitstempel mittels **erlang:now()** am Ende hinzugefügt.
6. Der Server bietet nach Außen eine Schnittstelle, an welcher Leser neue Nachrichten anfordern können und gemäß der Nummerierung eine Nachricht zugesandt bekommen, welche noch nicht an den Leser ausgeliefert wurde. In einem Flag wird dem Leser mitgeteilt, ob es noch weitere für ihn unbekannte Nachrichten gibt und die ID der nächsten Nachricht geliefert. Falls keine weiteren Nachrichten vorhanden sind, oder noch keine Nachrichten beim Server eingegangen sind, dann wird eine (nicht-leere) Dummy-Nachricht an den Leser versendet.
7. Der Server verwaltet Client-Verbindungen in einem ADT (**CMEM**). Wenn ein Client eine bestimmte Zeit keine Anfragen mehr gesendet hat, so wird er vergessen und bei der nächsten Anfrage wie ein unbekannter Client behandelt.
8. Wenn in der HBQ mehr als 2/3el an Nachrichten enthalten sind, als durch die vorgegebene maximale Anzahl an Nachrichten in der DLQ stehen können, dann wird, sofern eine Lücke besteht, diese Lücke zwischen HBQ und DLQ mit genau einer Fehlernachricht geschlossen. Etwa: ***Fehlernachricht fuer Nachrichtennummern 11 bis 17 um 16.05 18:01:30,580|. Hierbei handelt es sich bei „***“ um eine zu definierende Zeichenkette. Es werden keine weiteren Lücken innerhalb der Holdbackqueue behandelt! Die Fehlernachricht soll nur anhand einer entsprechenden Zeichenkette innerhalb der verschickten Nachricht zu erkennen sein und sich sonst nicht von gewöhnlichen Nachrichten unterscheiden.
9. Der Server terminiert, wenn sein Timer ausläuft, welcher durch Client-Anfragen jeweils zurückgesetzt wird.
10. Der Server verwendet drei **ADTs**: **HBQ** (Datei hbq.erl), **DLQ** (Datei dlq.erl) und **CMEM** (Datei cmem.erl) als Gedächtnis für die Leser-Clients. Diese dürfen hauptsächlich nur als Erlang-Liste ([ ]) realisiert werden! Als Hilfsstrukturen dürfen Tupel eingesetzt werden. Dazu sind die weiter unten aufgeführten Vorgaben zu beachten!
11. Die HBQ ist als entfernte ADT zu implementieren. Ihre Schnittstelle ist daher durch Nachrichtenformate beschrieben. Die DLQ oder das CMEM sind als lokale ADT zu implementieren. Daher sind deren Schnittstellen durch Funktionen beschrieben. Intern kann die DLQ bzw. das CMEM jedoch als externer Prozess realisiert werden.
12. Die steuernden Werte sind in einer Datei server.cfg anzugeben. Der Server ist unter einem **Namen** <name> im lokalen Namensdienst von Erlang zu registrieren (register(<name>,ServerPid)).

Client (Redakteur):

13. Vor dem Versenden einer Textzeile an den Server ruft ein Redakteur die neuste Nachrichtennummer ab und stellt diese seiner Textzeile voran.
14. Ein Redakteur schickt in einem festgelegten Zeitintervall Textzeilen an den Server, welche seinen Rechnernamen, die Praktikumsgruppe, die Teamnummer und die aktuelle Systemzeit enthalten. Optional können weitere Informationen in den Nachrichten stehen.
15. Nachdem der Redakteur eine eindeutige ID für seine Nachricht angefordert hat, wartet er so lange, bis das Zeitintervall abgelaufen ist und versendet anschließend seine Nachricht
16. Das Intervall wird nach dem Versenden von 5 Textzeilen jeweils um ca 50% per Zufall vergrößert oder verkleinert, darf hierbei aber nicht unter 2 Sekunden fallen.

17. Nach dem Versenden von 5 Textzeilen, vergisst der Redakteur nach Abruf einer Nachrichtennummer die zugehörige Nachricht zu verschicken und vermerkt dies in seinem Log. (Nummer, Zeit, vergessen zu senden)

Client (Leser):

18. Ein Leser-Client kann dem Server Anfragen senden, woraufhin dieser eine gemäß der Nummerierung noch nicht an den Leser ausgelieferte Nachricht versendet.
19. Ein Leser nimmt seine Rolle erst ein, nachdem er Redakteur war, wechselt aber beständig die Rollen.
20. Ein Leser fragt solange aktuelle Textzeilen ab, bis er alle erhalten hat und gibt sie dann in seiner GUI aus. Dabei erhält er immer genau eine ihm unbekannte Textzeile pro Nachricht.
21. Ein Leser merkt sich die Nachrichtennummern der erhaltenen Nachrichten und fügt den Nachrichten die durch seinen Redakteur-Client erstellt wurden die Zeichenfolge ******* an.
22. Der Leser wertet die mitgelieferten Zeitstempel aus. Wenn eine Nachricht aus der Zukunft stammt, dann wird diese mit der entsprechenden Zeitdifferenz am Ende der Zeichenkette markiert.
23. Bei der Initialisierung des Clients wird eine Lebenszeit definiert, nach Ablauf welcher er sich selbst terminiert.
24. Die steuernden Werte sind in einer Datei client.cfg anzugeben. Lediglich die aktuelle node des Servers kann als Parameter übergeben werden, d.h. beim Starten des Clients ist maximal ein Parameter erlaubt! Der Client darf (neben dem Prozess zum loggen) maximal aus einem Prozess bestehen.

Technische Anforderungen an das System:

Im Folgenden sind die jeweiligen Schnittstellen der einzelnen Komponenten aufgeführt:

Server: Alle im Server aufgeführten Schnittstellen sind externe Aufrufe. Daher müssen bei einigen Befehlen die eigene PID mitgegeben werden, damit das Ergebnis zurückgeliefert werden kann.

getmessages:
Befehl für Leser (Clients) um die nächste neue Nachricht abzurufen.
Server ! {self(),getmessages} →
receive {reply,[NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout],Terminated}
reply: Indikator dass eine Nachricht zurückgeliefert wird (Atom)
NNr: Nachrichtennummer (Integer)
Msg: Die tatsächliche Nachricht (Zeichenkette)
TSclientout: Timestamp (TS) wann die Nachricht vom Redakteur verschickt wurde
TShbqin: Timestamp (TS) wann die Nachricht bei der Holdbackqueue (HBQ) eingegangen ist
TSdlqin: Timestamp (TS) wann die Nachricht in die Deliveryqueue (DLQ) eingegangen ist
TSdlqout: Timestamp (TS) wann die Nachricht aus der DLQ ausgelesen (nicht entfernt) wurde
Terminated: Indikator (Boolean) ob es noch weitere für den aktuellen Client unbekannte Nachrichten gibt. Wenn Terminated == false ist, sind noch weitere Nachrichten vorhanden. Terminated == false analog → weitere Aufrufe von **getmessages** sind nicht nötig.

dropmessage:
Befehl für Redakteure (Clients) um eine neue Nachricht beim Server zu hinterlegen.
Server ! {dropmessage,[NNr,Msg,TSclientout]} → void
NNR: Eindeutige Nachrichtennummer, welche vom Server vergeben wird (also vorher abgefragt

werden muss).
Msg: Die tatsächsliche Nachricht (Zeichenkette)
TSclientout: Timestamp (TS) wann die Nachricht vom Redakteur versendet wurde.

getmsgid:
Befehl für Redakteure um die nächste Nachrichtennummer in Erfahrung zu bringen.
Server ! {self(),getmsgid} → receive {nid, Number}
nid: Indikator dass das nächste Element eine ID ist
Number: Die angeforderte ID der nächsten Nachricht

Hold-Back-Queue (HBQ): Alle im folgenden aufgeführten Befehle sind nach den angegebenen Aufrufsmustern zu verwenden. Die Form der Antworten sind durch die angegebenen Antwortmuster spezifiziert. *Die Schnittstelle gilt für den Server!*

Struktur der HBQ:
HBQ und DLQ sind eine Komponente. Die Aufrufe von der HBQ zur DLQ sind lokal. Vorerst sind HBQ und DLQ in einem gemeinsamen Prozess, wir behalten uns jedoch (wenn die Zeit ist) die DLQ in einen extra Prozess zu verlagern.

Ablauf 1 – Verarbeitung neuer Nachrichten:

1. Ein Redakteur schickt dem Server eine neue Nachricht, dieser wendet sich mit dem Aufruf „pushHBQ“ an die HBQ.
2. Die HBQ überprüft anhand der Funktion „expectedNr“, ob die erhaltene Nachricht bereits ausgeliefert werden kann. Hierzu bedient sich die HBQ intern einer Methode „isInOrder“.
    a. Ist die Nachricht nicht direkt auslieferbar (in Reihenfolge) so wird die Funktion „checkLimit“ aufgerufen. Diese prüft ob sich in der HBQ bereits mehr als 2/3 echte Nachrichten, gemessen an der DLQ Größe, befinden. Trifft dies zu wird erste gefundene Lücke mit einer Fehlernachricht quittiert, geschlossen und die so lückenlose Nachrichtenfolge der DLQ übermittelt.
    b. Ist die Nachricht auslieferbar (in Reihenfolge) dann wird sie der DLQ mittels „push2DLQ“ übergeben. Danach wird noch überprüft ob weitere Nachrichten aus der HBQ auslieferbar sind, sind sie es werden sie hinterhergeschickt.

Ablauf 2 – Auslieferung einer Nachricht

1. Ein Klient fragt beim Server an. Mittels CMEM wird diejenige Nachrichtennummer ermittelt, welche der Leser als nächstes bekommen sollte. Zusammen mit der Prozess ID wird sie beim Aufruf von „deliverMSG“ vom Server an die HBQ übertragen.
2. Die HBQ reicht die Informationen an die DLQ weiter in dem sie selbst „deliverMSG“ mit zusätzlich Queue und LogDatei aufruft.
3. DLQ sucht anhand der Nachrichtennummer die entsprechende Nachricht heraus.
    a. Die Nachricht ist vorhanden, sie wird an den Leser rausgeschickt und an die CMEM wir die gesendete Nachrichtennummer geschickt.

b. Die Nachricht ist in einer Lücke, so wird die nächste Nachricht gesendet die sich in der DLQ befindet. Und dann genauso wie bei a.
c. Die Nachrichtennummer ist größer als die aktuell größte Nachrichtennummer, es wird eine nicht leere Dummy Nachricht erstellt wobei das Terminierungsflag auf True gesetzt wurde.

initHBQ:

Befehl für den Server um seine Holdbackqueue zu initialisieren.
HBQ ! {self(), {request,initHBQ}} → receive {reply, ok}
request: Indikator, dass eine Anfrage gestellt wird (Atom)
reply: Indikator für eine atomare Antwort (Atom)
ok: Atomare Antwort zur Bestätigung des Vorgangs

pushHBQ

Befehl um eine Nachricht an die Holdbackqueue (HBQ) zu übergeben.
HBQ ! {self(), {request,pushHBQ,[NNr,Msg,TSclientout]}} → receive {reply, ok}
request: Indikator, dass eine Anfrage gestellt wird (Atom)
NNR: Aktuelle Nachrichtennummer (Integer)
Msg: Die tatsächliche Nachricht (Zeichenkette)
TSclientout: Timestamp (TS) wann die Nachricht von dem zugehörigen Redakteur verschickt wurde
reply: Indikator, dass eine Antwort gegeben wird (Atom)
ok: Atomare Antwort zur Bestätigung des Vorgangs

deliverMSG:

Befehl um eine Nachricht auszuliefern und an die DLQ zu übergeben.
HBQ ! {self(), {request,deliverMSG,NNr,ToClient}} → receive {reply, SentNNr}
request: Indikator, dass eine Anfrage gestellt wird (Atom)
NNR: Nachrichtennummer (Integer)
ToClient: Angabe welcher Client die Nachricht geliefert bekommen soll (PID)
reply: Indikator, dass eine Antwort gegeben wird (Atom)
SentNNR: Die Nummer der versendeten Nachricht (Integer), schickt diese Nummer mit Server ! {reply,SentNNR} an den Server (für CMEM).

dellHBQ:

Befehl um die HBQ zu terminieren.
HBQ ! {self(), {request,dellHBQ}} → receive {reply, ok}
request: Indikator, dass eine Anfrage gestellt wird (Atom)
reply: Indikator, dass eine Antwort gegeben wird (Atom)
ok: Atomare Antwort zur Bestätigung des Vorgangs

Delivery-Queue (DLQ): Alle im folgenden aufgeführten Befehle sind nach den angegebenen Aufrufsmustern zu verwenden. Die Form der Antworten sind durch die angegebenen Antwortmuster spezifiziert. *Die Schnittstelle gilt für die Hold-Back-Queue!*

initDLQ(Size,Datei) → void

Befehl um die DLQ zu initialisieren.
Size: Maximale Anzahl an Plätzen der DLQ (Integer)
Datei: Die von der DLQ zu verwendende Log-Datei

delDLQ(Queue) → ok

Befehl um die DLQ zu löschen.
Queue: Referenz auf die DLQ
ok: Atomare Antwort zur Bestätigung des Vorgangs

expectedNr(Queue) → Number

Befehl um die nächste Nummer abzufragen welche in der DLQ gespeichert werden kann.
Queue: Referenz auf die DLQ
Number: Die erwartete nächste Nummer. Wenn die DLQ leer ist, wird 1 zurückgegeben (Integer).

push2DLQ([NNr,Msg,TSclientout,TShbqin],Queue,Datei) → ModifiedQueue

Befehl um eine Nachricht in der DLQ abzuspeichern.
NNR: Nachrichtennummer (Integer)
Msg: Die tatsächliche Nachricht (Zeichenkette)
TSclientout: Timestamp (TS) wann die Nachricht von dem zugehörigen Redakteur verschickt wurde
TShbqin: Timestamp (TS) wann die Nachricht bei der Holdbackqueue (HBQ) eingegangen ist
Queue: Referenz auf die aktuelle DLQ
Datei: Zum Logging zu verwendende Datei
TSdlqin: Timestamp (TS) wann die Nachricht bei der DLQ eingegangen ist. Dieser Parameter wird der Liste **[NNr,Msg,TSclientout,TShbqin]** angefügt.
ModifiedQueue: Die modifizierte Queue

deliverMSG(MSGNr,ClientPID,Queue,Datei) → SentMsgNum

Befehl um eine Nachricht an einen Client auszuliefern.
MSGNr: Nummer der auszuliefernden Nachricht (Integer). Sollte die Nachrichtennummer nicht mehr vorhanden sein, wird die Nachricht mit der nächsthöheren Nummer aus der DLQ versendet
ClientPID: Die Prozess-ID des Clients, welcher die Nachricht erhalten soll
Queue: Referenz auf die DLQ
Datei: Zum Logging zu verwendende Datei
TSdlqout: Timestamp (TS) wann die Nachricht bei der DLQ ausgegangen ist. Dieser Parameter wird am Ende der Messagelist der zu versendenden Nachricht angefügt.
SentMsgNum: Die Nummer der tatsächlich versendeten Nachricht.

CMEM: Verwaltungs-ADT um die Verbindungen zu den Clients zu verwalten. Alle im folgenden aufgeführten Befehle sind nach den angegebenen Aufrufsmustern zu verwenden. Die Form der Antworten sind durch die angegebenen Antwortmuster spezifiziert. *Die Schnittstelle gilt für den Server!*

Struktur der CMEM:
Die CMEM gehört in die Server Komponente. Und ist vorerst auch in diesem Prozess mit dem Server vereint. Wir behalten uns jedoch vor (wenn die Zeit ist) die CMEM in einen eigenen Prozess auszulagern.

Ablauf (Manche Parameter sind gekürzt um den Ablauf hier besser beschreiben zu können.):

1. Leser fordert vom Server eine neue Nachricht an.
2. Der Server ruft nun per „getclientNNR“ die nächste Nachrichtennummer für den Client ab. Und schickt diese zusammen mit der Client Prozess ID per „deliverMSG“ an die HBQ Komponente.
3. Nach dem Vorgang innerhalb der HBQ Komponente erreicht den Server ein „reply“ mit der gesendeten Nachrichtennummer. Mit dieser wird der entsprechende Clienteintrag (mit Timer und zuletzt gesendeter Nummer) mit „updateClient“ aktualisiert bzw. erstellt.

initCMEM(RemTime,Datei) → EmptyCMEM
Befehl um den CMEM zu initialisieren.
RemTime: Zeit in [s] nach Ablauf derer ein Client vergessen wird, wenn er keine neuen Anfragen an die Server-Komponente stellt
Datei: Angabe einer Logging-Datei
EmptyCMEM: Einen leeren CMEM

delCMEM(CMEM) → ok
Befehl um den CMEM zu löschen.
CMEM: Referenz auf den aktuell verwendeten CMEM
ok: Atomare Antwort zur Bestätigung des Vorgangs

updateClient(CMEM,ClientID,NNr,Datei) → CMEM vorher: void
Aktualisiert einen verwalteten Client und speichert die an ihn gesendete Nachrichtennummer. Ist der Client noch nicht vorhanden, so wird ein Eintrag für ihn angelegt.
CMEM: Referenz auf den verwendeten CMEM
ClientID: ID um den zu aktualisierenden Client zu identifizieren
NNR: Nachrichtennummer (Integer)
Datei: Zu verwendende Logging-Datei
CMEM: Es wird die neue CMEM zurückgegeben.

getClientNNr(CMEM,ClientID) → ExpectedNNR
Liefert die nächste vom Client erwartete Nachrichtennummer zurück. Ist der Client unbekannt wird 1 zurückgegeben.
CMEM: Referenz auf den aktuell verwendeten CMEM
ClientID: ID um den Client zu referenzieren
ExpectedNNR: Die vom Client als nächstes erwartete Nachrichtennummer (Integer)

Beispielhafter Ablauf des Systems:

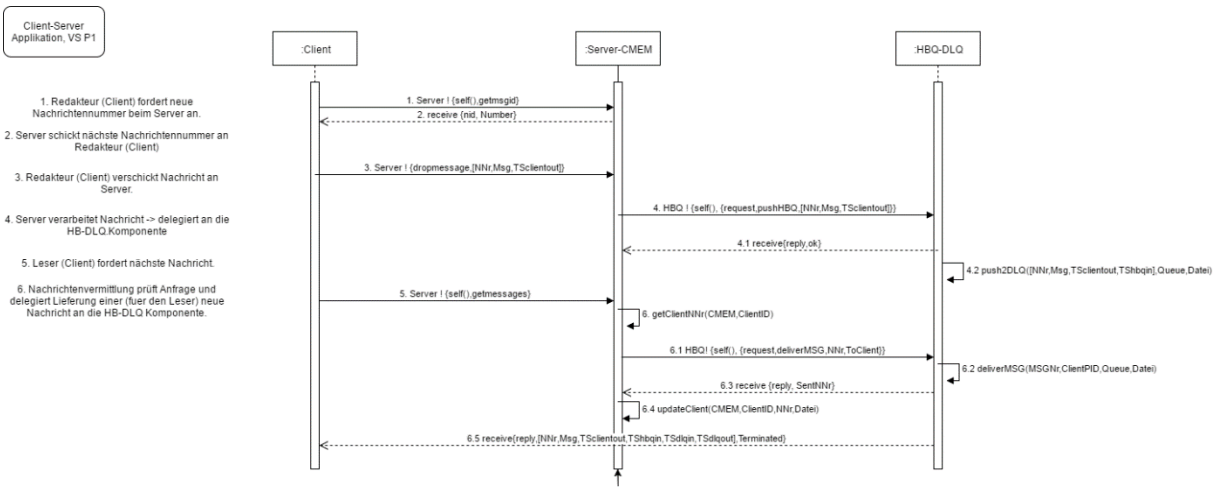

Abbildung 1: Sequenzdiagramm eines Beispielhaften Ablaufs

In Abbildung 1 ist ein kurzer Ablauf als Sequenzdiagramm dargestellt. In diesem wurde der Einfachheit halber darauf verzichtet die einzelnen Komponenten zu initialisieren. Es wird davon ausgegangen, dass dies bereits korrekt getan wurde. Beginn der Sequenz ist das verschicken einer Nachricht des Clients (als Redakteur) an den Server. Anschließend werden die etwaigen Bearbeitungsschritte durchlaufen. Diese sind mit den jeweiligen Aufrufen gekennzeichnet.

Da Abbildung 1 in diesem Dokument sehr klein dargestellt ist, wird zusätzlich zum Entwurf eine Bild-Datei ausgeliefert, welche das dargestellte Sequenzdiagramm beinhaltet.