Team: 01, Francis Opoku, Fabian Reiber

Aufgabenaufteilung:

1. Sequenzdiagramme und zugehörige Doku, sowie restliche Dokumentation
2. Klassen- und Komponentendiagramm, zugehörige Doku, sowie restliche Dokumentation

Quellenangaben: http://learnyousomeerlang.com/what-is-otp#the-basic-server

Begründung für Codeübernahme: keine Übernahme

Bearbeitungszeitraum:

Entwurf: 23.03.15: Francis Opoku 3,5 Std.; Fabian Reiber 2Std.; davon 2 Std. gemeinsam

24.03.15: Francis Opoku 6Std.; Fabian Reiber 6 Std.

25.03.15: 2,25 Std. gemeinsam; Fabian Reiber 2 Std.

26.03.15: Francis Opoku 2 Std.

27.03.15: ca. 3 Std. gemeinsam

Implementierung: bisher noch nichts

Aktueller Stand: Entwurf ist fertig. Implementierung wurde noch nicht angefangen, da Fremdentwurf verwendet werden soll.

Änderungen im Entwurf: keine bisher

Entwurf: siehe folgend Seiten...

1.Einleitung

1.1.Ziel

In diesem Dokument wird eine einfache Client-Server-Anwendung, in Form eines Nachrichtendienstes, detailliert entworfen. Ziel ist es, dass Redakteure (Clients) dem Nachrichtendienst (Server) Nachrichten des Tages zukommen lassen können. Diese Nachrichten können dann von Lesern (Clients) wiederum abgerufen werden. Dabei werden die einzelnen Nachrichten und Ereignisse in einer GUI (Log-Files) dargestellt.

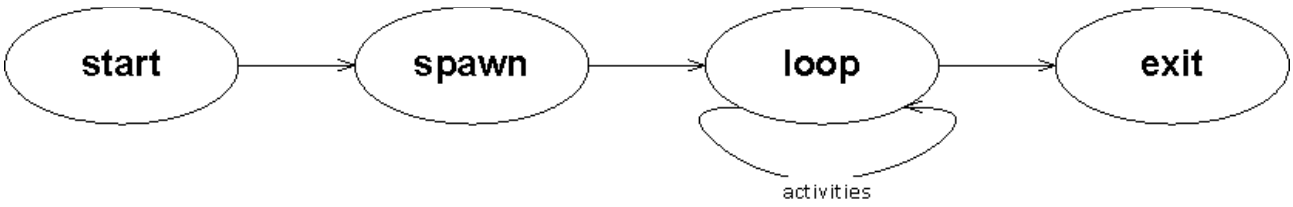
1.2.Konventionen

In der Beschreibung des Klassendiagramms werden zu den Methodensignaturen, welche fett und rekursiv gekennzeichnet sind, die pre- und post-conditions aufgeführt, sowie der Rückgabewert der Methoden. In den Definitionen die zu jeder Methode angegeben sind, werden auf Methoden im selben Modul, wenn nicht anders ausdrücklich angegeben, hingewiesen. Diese sind in Hochkommata („“) gekennzeichnet.
Der Entwurf lehnt sich an die Programmiersprache Erlang an, sodass sich Erlang-Strukturen wie z.B. Listen, Tupel, Atome, Variablen etc. in diesem Dokument wieder finden lassen.
Teilweise sind Verweise auf die Aufgabenstellung angegeben, damit die genaueren Details dort nachgelesen werden können.

2.Rahmenbedingungen

2.1.allg. Ablauf (Prozesse)

Eine Bedingung, um den Entwurf genauer verstehen zu können, ist, dass man sich den allgemeinen Ablauf der Prozess-Erstellung genauer ansehen muss. Dieser wird im folgenden Diagramm angegeben und kurz erläutert.
Dazu muss vorab erwähnt werden, dass bei dieser Anwendung drei Prozesse zu tragen kommen, die erstellt und verwaltet werden müssen. Dabei handelt es sich um den Server-, Client- und den HBQ-Prozess.
Die Server-, Client- und HBQ-Anwendungen werden jeweils mit den Methoden „start()“ gestartet. Dort wird jeweils mit der spawn-Funktionalität ein neuer Prozess erzeugt. Diese Prozesse steigen jeweils in die eigene loop-Methode ein und arbeiten ihre Aktivitäten dort ab. Treten bestimmte jeweilige Bedingungen für Server, Client oder HBQ ein, werden die jeweiligen Prozesse terminiert (Zustand exit).



2.2.Verwendete Bibliotheken/externe Module

Wir verwenden in unserem Entwurf, hauptsächlich zum Loggen der Informationen und auslesen der Konfigurationsdateien, das Modul „werkzeug.erl“ vom Professor Klauck. Das Loggen stellt die GUI für diese Anwendung dar und ist in Form von log-Files repräsentiert. Das Modul bietet noch weitere nützliche Methoden an, welche ebenfalls genutzt werden können.

2.3.Beschreibung der Startroutine

In der Startroutine werden der HBQ- und der Server-Prozess gestartet. Diese Routine ist sinnvoll, damit ein geordneter Start der Anwendung gewährleistet ist. Es ist zu beachten, dass der HBQ-Prozess, vor dem Server-Prozess zu starten ist, da der Server zu Beginn die HBQ Datenstruktur initialisiert. Der HBQ-Prozess muss genauso wie der Server-Prozess im lokalen Namensdienst von Erlang registriert werden.

3.Modularisierung

3.1.Beschreibung der Module

a)Komponentendiagramm

•**Komponente Client:**

Verantwortungen der Komponente: Die Komponente Client bündelt den Redakteur- und Leser-Client und hat die Aufgabe eine gewisse Anzahl an Nachrichten an den Server zu senden und nach 5 gesendeten Nachrichten für ihn noch unbekannte Nachrichten vom Server anzufordern.
Außensicht: Die Client-Komponente bietet nach außen hin keine Schnittstellen an, nutzt allerdings die Schnittstellen anderer Komponenten: Server (IMessageService), Tool (ILogService, IClientConfig).
Innensicht: Die Komponente Client besteht aus dem Modul „client.erl“. In ihr werden sowohl der Redakteur- als auch der Leser-Client realisiert.
Entwurfsentscheidungen: Die Vorgaben der Aufgabenstellung wurden hier berücksichtigt und werden als Entwurfsentscheidung betrachtet.
Schnittstellen zu Nachbarsystemen: keine

•**Komponente Server:**

Verantwortungen der Komponente: Der Server fungiert als Nachrichtendienst. Er nimmt Nachrichten von Redakteur-Clients entgegen und verteilt die Nachrichten an interessierte Leser-Clients. Er merkt sich intern welche Leser-Clients sich in der Vergangenheit bei ihm gemeldet und Nachrichten angefordert haben.
Außensicht: Die Server-Komponente bietet seine Dienste für die Clients an (IMessageService) und nutzt drei weitere Dienste anderer Komponenten: Tool (ILogService, IServerConfig) zum Loggen der Ausgaben, sowie das Einlesen der Konfigurationsdatei, Queue (IHBQService) für das Verwalten der Nachrichten in den Schlangen, sowie dem Anweisen, dass Nachrichten an die Clients gesendet werde sollen.
Innensicht: Die Komponente besteht aus den zwei Modulen „server.erl“ und „cmem.erl“. Das Modul „cmem.erl“ definiert die Datenstruktur zur Verwaltung der Leser-Clients.
Entwurfsentscheidungen: Die Vorgaben der Aufgabenstellung wurden hier berücksichtigt und werden als Entwurfsentscheidung betrachtet.
Schnittstellen zu Nachbarsystemen: keine

•**Komponente Queue:**

Verantwortungen der Komponente: In dieser Komponente werden die eingetroffenen Nachrichten verwaltet. Nachrichten die noch nicht versendet werden können, werden in einer HBQ gesichert. Nachrichten die an einen Client gesendet werden können, werden in der DLQ gespeichert. Die DLQ ist auch dafür verantwortlich die Nachrichten an einen Leser-Client zu senden, wenn es die Aufforderung der HBQ erhält.
Außensicht: Die Komponente bietet ihren Service für die Server-Komponente an (IHBQService). Allerdings nur die Dienste der HBQ, da die DLQ intern nur von der HBQ angesprochen wird.
Innensicht: Die Komponente Queue besteht aus den zwei Modulen „hbq.erl“, zur Speicherung der Nachrichten, welche noch nicht sende-bereit sind sowie dem Modul „dlq.erl“, welches die Nachrichten verwaltet, welche auf Anfrage eines Leser-Clients gesendet werden.
Entwurfsentscheidungen: Die Vorgaben der Aufgabenstellung wurden hier berücksichtigt und werden als Entwurfsentscheidung betrachtet.
Schnittstellen zu Nachbarsystemen: keine

•**Komponente Tool:**

Verantwortungen der Komponente: Diese Komponente stellt nützliche Funktionen bereit, welche bei dieser Anwendung genutzt werden sollen. Hauptsächlich geht es dabei um das Loggen und Auslesen der Konfigurationsdatei.
Außensicht: Diese Komponente bietet 2 gleiche Dienste (Loggen und Auslesen der Konfigurationsdatei) an, welche jeweils von 2 unterschiedlichen Komponenten (Client, Server) genutzt werden.
Innensicht: Die Tool-Komponente besteht lediglich aus dem Modul „werkzeug.erl“. Die wichtigsten Funktionalitäten die hier verwendet werden sollen, sind diese zum Loggen der Nachrichten und zum Konfigurieren des Clients und des Servers.
Entwurfsentscheidungen: keine, da diese so übernommen wurde.
Schnittstellen zu Nachbarsystemen: keine

b)Klassendiagramm
Hier ist eine Übersicht über die anzufertigenden Zeitstempeln, die auf dem Weg einer Nachricht vom Redakteur-Client zum Leser-Client erzeugt werden:

TSclientout: wird erzeugt, wenn der Redakteur-Client die Nachricht an den Server übermittelt. Dies geschieht in der Methode „sendMSG(Servername, Servernode“ des Clients.
TShbqin: wird erzeugt, wenn die vom Server weitergeleitete Nachricht in die HBQ eingefügt wird. Dies geschieht in der Methode „pushHBQ(ServerPID, OldHBQ, [NNr, Msg, TSclientout])“ im Modul „hbq.erl“.
TSdlqin: wird erzeugt, wenn der HBQ-Prozess die Nachricht in die DLQ einreiht. Dies geschieht in der Methode „push2DLQ([NNr, Msg, TSclientout, TShbqin], Queue, Datei)“ im Modul „dlq.erl“.
TSdlqout: wird erzeugt, wenn die DLQ die angeforderte Nachricht an den Client sendet. Dies geschieht in der Methode „deliverMSG(MSGNr, ClientPID, Queue, Datei)“ im Modul „dlq.erl“.

Wann genau geloggt wird kann der Aufgabenstellung genauer entnommen werden, sowie das mögliche Aussehen eines Log-Eintrages.
Im folgenden werden die Methoden der einzelnen Module (in dieser Reihenfolge: Client, Server, HBQ, DLQ, CMEM) kurz definiert und erläutert.

Client:

start()
Definition: Startet einen neuen Client-Prozess der Nachrichten an den (laufenden) Server senden kann und Nachrichten abrufen kann.
pre: keine
post: Ein neuer Prozess wurde gestartet, der mit dem Server kommunizieren kann
return: client started als Atom sonst eine sinnvolle Error-Meldung

readConfig()
Definition: Vor dem Start des Client-Prozesses muss die Konfigurationsdatei (siehe Vorlage) des Clients ausgelesen werden. Das Modul „werkzeug.erl“ vom Professor wird hierfür verwendet.
pre: Die Datei „client.cfg“ ist vorhanden
post: Die Datei wurde erfolgreich ausgelesen und die erforderlichen Werte zurückgeliefert
return: {Clients, Lifetime, Servername, Servernode, Sendinterval}

loop(Lifetime, Servername, Servernode, Sendinterval)
Definition: In der Hauptschleife werden die Nachrichten in bestimmten Zeitabständen an den Server versendet. Dabei wird die Methode „sendMSG(Servername, Servernode)“ verwendet.

Weiterhin wird nach dem Senden von 5 Nachrichten die Methode „changeSendintervall(Sendinterval)“ aufgerufen, um das Sendeintervall neu zu berechnen. Ebenfalls nach 5 Nachrichten wird die Methode „askForMSGID(Servername, Servernode)“ aufgerufen. Hiermit wird der Server aufgefordert, ihm (dem Client) eine eindeutige Nachrichtennummer zu übermitteln.
Sobald die Rolle des Leser-Clients übernommen wird (nach 5 gesendeten Nachrichten, siehe Aufgabenstellung), wird die Methode „getMSG(Servername, Servernode)“ aufgerufen.
Weiterhin ist es notwendig in dieser Methode zu prüfen, wann die Lebenszeit des Clients vorüber ist.
pre: die nötigen Übergabeparameter sind korrekt eingelesen und der Server ist noch nicht terminiert
post: der Prozess wurde erfolgreich, nach angegebenen Kriterien, terminiert
return: client terminated als Atom

sendMSG(Servername, Servernode)
Definition: Hier wird dem Server die Nachricht {dropmessage, [INNr, Msg, TSclientout]} gesendet. Auf eine Antwort des Server wird nicht gewartet.
pre: Server ist unter angegebenen Servername und Servernode weiterhin erreichbar
post: Server hat erfolgreich eine Nachricht erhalten
return: Atom ok wird zurückgegeben

changeSendinterval(Sendinterval)
Definition: Auf Grundlage des alten Sendeintervalls (Parameter Sendintervall) wird dieses um ca. 50% zufällig vergrößert oder verkleinert. Jedoch darf das Sendeintervall niemals kleiner als 2 Sekunden sein.
pre: keine
post: keine
return: Das neu berechnete Intervall NewSendInterval wird zurückgeliefert

askForMSGID(Servername, Servernode)
Definition: Dem Server wird folgende Nachricht übermittelt: {self(), getmsgid}. Im Anschluss wartet er auf folgende Antwort des Server: {nid, Number}.
pre: Server ist unter angegebenen Servername und Servernode weiterhin erreichbar
post: erhaltene Nachrichten-ID ist korrekt
return: die Nachrichten-ID (Number) als Integer-Wert wird zurückgegeben

getMSG(Servername, Servernode)
Definition: Dem Server wird folgende Nachricht übermittelt: {self(), getmessages}. Er wartet auf die Antwort des Servers (DLQ) mit folgendem Format: {reply,[NNr, Msg, TSclientout, TShbqin, TSdlqin, TSdlqout], Terminated}. Der Server wird somit aufgefordert alle Nachrichten einzeln an den Leser-Client zu übermitteln. Diesen Vorgang wiederholt der Client solange, bis er alle Nachrichten bekommen hat und somit auf dem neuesten Stand ist. Dies erkennt der Client an dem Flag Terminated(true). Mithilfe des Moduls „werkzeug.erl“ können die Nachrichten in der GUI (Logfile) ausgegeben werden. Nach jedem Erhalt einer Nachricht, fügt der Leser-Client eine eigene Nachricht, der jeweiligen erhaltenen Nachricht, hinzu, sowie ein selbst erstellten Timestamp (siehe Anforderung 12).
pre: Server ist unter angegebenen Servername und Servernode weiterhin erreichbar
post: Nachricht erfolgreich erhalten, damit sie geloggt werden kann
return: Atom ok wird zurückgegeben

Server:

start()

Definition: Startet einen neuen Server-Prozess, damit dieser im Verlauf Nachrichten erhalten kann und den HBQ-Prozess auffordern kann Nachrichten an einen Leser-Clients zu senden. Der Server initialisiert eine HBQ, indem er dem HBQ-Prozess folgende Nachricht schickt: {self(), {request,initHBQ}}. Während der HBQ-Prozess die Datenstrukturen HBQ und DLQ erzeugt, erzeugt der Server die CMEM-Datenstrutktur, indem er „initCMEM(RemTime, Datei)“ aufruft. Im Anschluss wartet er auf eine Antwort, seitens des HBQ-Prozesses: {reply, ok}. Des Weiteren muss der Server im lokalen Namensdienst registriert werden (siehe Anforderung 8).
pre: keine
post: Server ist gestartet und im lokalen Namensdienst von Erlang registriert
return: server started als Atom sonst eine sinnvolle Error-Meldung

readConfig()

Definition: Vor dem Start des Server-Prozesses muss die Konfigurationsdatei (siehe Vorlage) des Servers ausgelesen werden. Das Modul „werkzeug.erl“ vom Professor wird hierfür verwendet.
pre: Die Datei „server.cfg“ ist vorhanden
post: Die Datei wurde erfolgreich ausgelesen und die erforderlichen Werte zurückgeliefert
return: {Latency, Clientlifetime, Servername, HBQname, HBQnode, DLQlimit}

loop(Latency, Clientlifetime, Servername, HBQname, HBQnode, DLQlimit)

Definition: Zu Beginn dieser Methode prüft ruft der Server die Methode „delExpiredCl(CMEM, Clientlifetime)“ aus dem Modul „cmem.erl“ auf, um zu prüfen, ob Clients gelöscht werden können. In dieser Hauptschleife wartet der Server auf Anfragen seitens der Clients. Ebenso müssen Nachrichten seitens des HBQ-Prozesses abgearbeitet werden. Erhält der Server die Nachricht eines Clients mit folgendem Format: {ClientPID, getmessages}, wird die Methode „sendMessages(ClientPID)“ aufgerufen.
Wird die Nachricht {dropmessage,[INNr, Msg, TSclientout]} von einem Client empfangen, ruft der Server die Methode „dropmessage(HBQname, HBQnode)“ auf.
Eine weitere Nachricht die seitens eines Clients beim Server eintreffen kann ist folgende: {ClientPID, getmsgid}. Es muss nun die Methode sendMSGID(ClientPID) aufgerufen werden.
Wird der Server terminiert, da sich nach einer gewissen Zeit kein Client mehr bei ihm gemeldet hat, muss auch der HBQ-Prozess terminiert werden. Dies erfolgt mit der Nachricht: {self(), {request,dellHBQ}} an den HBQ-Prozess. Der Server wartet auf folgende Antwort: {reply, ok}.
pre: die nötigen Übergabeparameter sind korrekt eingelesen
post: der Prozess wurde erfolgreich, nach angegebenen Kriterien, terminiert
return: server terminated als Atom

sendMessages(ToClient, CMEM)

Definition: In dieser Methode muss der Server zuerst die korrekte Nachrichtennummer aus dem CMEM holen. Er ruft die Methode „getClientNNr(CMEM, ClientID)“ aus dem Modul „cmem.erl“ auf um diese zu bekommen. Die Parameter „ToClient“ und „ClientID“ sind identisch. Im Anschluss sendet der Server eine Nachricht an den HBQ-Prozess: {self(), {request,deliverMSG,NNr,ToClient}} Hiermit wird der HBQ-Prozess aufgefordert Nachrichten an den Client („ToClient“) zu senden. Der Server wartet auf die Antwort der HBQ: {reply, SendNNr}. Sobald er diese SendNNr hat, muss der Server ebenfalls den CMEM-Eintrag des jeweiligen Clients anpassen und ruft die Methode „updateClient(CMEM, ClientID, NNR, Datei)“ aus dem Modul „cmem.erl“ auf.
pre: korrekt übergebene Client PID

post: die Nachrichten wurden erfolgreich aus der DLQ geholt und an Client gesendet
return: Atom ok wird zurückgegeben

dropmessage(HBQname, HBQnode)
Definition: Der Server sendet folgende Nachricht an den HBQ-Prozess: {self(), {request,pushHBQ, [NNr,Msg,TSclientout]}}. Somit wird der HBQ-Prozess aufgefordert eine Nachricht zu sichern. Der Server wartet auf Antwort der HBQ und erwartet folgende Nachricht: {reply, ok}.
pre: HBQ-Prozess vorhanden und mit übergebenen Parametern ansprechbar
post: erreichte Nachricht vom Client wurde erfolgreich in die HBQ eingetragen
return: Atom ok wird zurückgegeben

sendMSGID(ClientPID, CMEM)
Definition: Der Server muss nun im Modul „cmem.erl“ die Methode „getClientNNr(CMEM,ClientID)“ aufrufen, damit dieser die aktuelle Nachrichtennummer erhalten kann. Der Server schickt dann die Nachricht {nid, Number} an die angegeben ClientPID. Number ist die empfangene Nachrichtennummer.
pre: korrekt übergebene Client PID
post: erfolgreiche Ermittlung einer neuen MSGID
return: Atom ok wird zurückgegeben

HBQ:

start()
Definition: Startet einen neuen HBQ-Prozess, der die HBQ verwaltet und alleinigen Zugriff auf die DLQ besitzt.
pre: Server-Prozess ist gestartet
post: Es wurde ein neuer HBQ-Prozess gestartet der nun vom Server verwendet werden kann
return: hbq-process started als Atom sonst eine sinnvolle Error-Meldung

loop()
Definition: Die Hauptschleife der HBQ wartet auf Anfragen bzgl. des Servers. Direkt nach der Erstellung des HBQ-Prozesses, bekommt dieser die Anfrage: {ServerPID, {request,initHBQ}} vom Server. Es wird also die Methode „initHBQandDLQ(ServerPID)“ aufgerufen.
Erhält der HBQ-Prozess die Nachricht {ServerPID, {request,pushHBQ,[NNr,Msg,TSclientout]}} wird die Methode „pushHBQ(ServerPID, OldHBQ, [NNr,Msg,TSclientout])“ aufgerufen.
Eine weitee Nachricht die dieser Prozess erhalten kann ist: {ServerPID, {request,deliverMSG,NNr,ToClient}}. Nun wird die Methode „deliverMSG(ServerPID, DLQ, NNr, ToClient)“ aufgerufen.
Wird nun der Server terminiert, so muss auch der HBQ-Prozess geschlossen werden. Dieser erhält folgende Nachricht: {ServerPID, {request,dellHBQ}}. Es wird die Methode „dellHBQ(ServerPID)“ für diese Aufgabe aufgerufen.
pre: keine
post: der Prozess wurde erfolgreich terminiert
return: hbq-process terminated als Atom

initHBQandDLQ(*ServerPID*)
Definition: Initialisiert die HBQ und DLQ. Ruft die Methode „initDLQ(Size, Datei)“ in dem Modul „dlq.erl“ auf. Diese erhält nun die DLQ-ADT in der Form: {Size, []} (siehe DLQ-Definition). Dem Server wird die Meldung {reply,ok} geschickt, als Zeichen der korrekten Initialisierung. Der loop-Methode wird das 2-Tupel {HBQ, DLQ} zurückgegeben (siehe *return*).

pre: korrekte ServerPID unter der der Server erreichbar ist
post: ein 2-Tupel wurde erstellt. Das 1. Element ist die HBQ und das 2. Element die DLQ.
return: 2-Tupel: {[], DLQ}

pushHBQ(ServerPID, OldHBQ, [NNr, Msg, TSclientout])
Definition: Fügt die Msg (Textzeile) mit Nummer (NNr) und dem Sende-Zeitstempel (TSclientout) in die alte HBQ ein. Dem Server wird über die ServerPID ebenfalls ein {reply, ok} zugeschickt.
pre: ServerPID mit der der Server erreicht werden kann, sowie eine korrekte OldHBQ.
post: Der alten HBQ wurde ein neues Element beigefügt und der Server hat eine Nachricht erhalten.
return: NewHBQ

deliverMSG(ServerPID, DLQ, NNr, ToClient)
Definition: Beauftragt die DLQ die Nachricht mit geforderter NNr an den Client (ToClient) zu senden. Sie ruft intern die Methode „deliverMSG(MSGNr, ClientPID, Queue, Datei)“ aus dem Modul „dlq.erl“ auf. Dem Server ist im Anschluss an dieser Methode die Nachricht {reply, SendNNr} zurück zu senden. SendNNr ist die vom DLQ-Modul zurückgegebene tatsächlich verschickte Nachrichtennummer.
pre: korrekte Server-und ClientPID unter die beide Prozesse zu erreichen sind
post: Der Client hat eine neue Nachricht erhalten, die DLQ ist um eine Nachricht kleiner geworden und der Server hat ein die tatsächlich gesendete Nachrichtennummer erhalten.
return: Atom ok wird zurückgegeben

dellHBQ(ServerPID)
Definition: Terminiert den HBQ-Prozess und schickt dem Server die Nachricht {reply, ok}
pre: korrekte ServerPID an die ein {reply, ok} gesendet werden kann
post: der Prozess wurde erfolgreich beendet
return: Atom ok wird zurückgegeben

pushSeries(HBQ, DLQ)
Definition: Prüft auf Nachrichten / Nachrichtenfolgen, die ohne eine Lücke zu bilden in die DLQ eingefügt werden können. Prüft außerdem, ob die Anzahl der Nachrichten, die in der HBQ sind, 2/3 der Anzahl beträgt die in die DLQ passen. Ist dies der Fall, wird einmalig die Lücke der DLQ mit einer Fehlernachricht geschlossen (siehe Anforderung 6).
pre: korrekt initialisierte HBQ- und DLQ-Datenstruktur
post: veränderte HBQ- und DLQ-Datenstruktur
return: {HBQ, DLQ} als 2-Tupel

DLQ:

initDLQ(Size, Datei)
Definition: Vom HBQ-Prozess aufgerufene Methode die eine neue leere DLQ-ADT zurückliefert.
pre: keine
post: 2-Tupel mit Größenangabe, sowie einer leeren Liste
return: {Size, []} - 2-Tupel mit Size als 1. Element und einer leeren Liste als 2. Element

expectedNr(Queue)
Definition: Als nächstes zu speichernde Nachrichtennummer wird an HBQ-Prozess zurückgegeben.
pre: eine Queue in Form der DLQ-ADT (siehe initDLQ)
post: Queue ist unverändert und eine korrekte Nachrichtennummer wurde zurückgegeben

return: nächste Nachrichtennummer die verwendet werden kann, sonst 1 bei leerer Liste

push2DLQ([NNr, Msg, TSclientout, TShbqin], Queue, Datei)
Definition: Die von dem HBQ-Prozess kommende Nachricht im Format: [NNr, Msg, TSclientout, TShbqin] wird in die DLQ eingefügt. Ebenso wird an die Msg ein aktueller Zeitstempel angehängt, sowie nochmals am Ende des Nachrichtenformates (TSdlqin).
pre: Übergabeparameter Queue ist korrekte DLQ-ADT (siehe initDLQ)
post: die DLQ ist nun um eine Nachricht gewachsen, sofern die Größe es zugelassen hat
return: die neue DLQ: NewDLQ ; wurde die maximale Größe der DLQ erreicht wird eine erkennbare Error-Meldung zurückgegeben

deliverMSG(MSGNr, ClientPID, Queue, Datei)
Definition: Der HBQ-Prozess ruft diese Methode der DLQ auf, um sie aufzufordern die Nachricht mit angegebener MSGNr an die ClientPID zu senden. An die Nachricht wird zusätzlich ein Ausgangszeitstempel (TSdlqout) angehängt. Ist die MSGNr allerdings nicht vorhanden, wird die nächst größere Nummer verwendet. Das Format der Nachricht die dem Client gesendet wird hat folgende Gestalt: {reply,[NNr, Msg, TSclientout, TShbqin, TSdlqin, TSdlqout], Terminated}. Terminated signalisiert, ob es noch weitere Nachrichten zu senden gibt (false). Sonst true.
pre: die Queue ist in Form der DLQ-ADT (siehe initDLQ), sowie die ClientPID ist korrekt
post: eine MSGNr wurde zurückgegeben und die Queue um diese Nachricht verkleinert
return: die tatsächlich verschickte MSGNr als Integer-Wert an den HBQ-Prozess

CMEM:

initCMEM(RemTime, Datei)
Definition: Initialisiert die CMEM für den Server.
pre: keine
post: neues 2-Tupel erstellt
return: {RemTime, []} - 2-Tupel mit RemTime als erstes Element und eine leere CMEM-Liste als 2. Element

updateClient(CMEM, ClientID, NNr, Datei)
Definition: Speichert/Aktualisiert im CMEM die ClientID mit der NNr.
pre: nötige Übergabeparameter sind korrekt
post: neuer Client gespeichert, oder einen bereits vorhandenen Client aktualisiert
return: aktualisiertes CMEM

getClientNNr(CMEM, ClientID)
Definition: Liefert dem Server die nächste Nachrichtennummer die an die ClientID geschickt werden soll.
pre: keine
post: nicht veränderte CMEM, da nur lesend
return: ClientID als Integer-Wert, wenn nicht vorhanden wird 1 zurückgegeben

delExpiredCl(CMEM, Clientlifetime)
Definition: In dieser Methode werden die Clients gelöscht, welche die Clientlifetime überschritten haben.
pre: keine
post: veränderte CMEM
return: Das Atom ok als Rückgabewert

3.2.Kommunikation zwischen den Modulen

a)**Sequenzdiagramme**

Allgemeines

Da die Aufgabenstellung in 15 Unterpunkte unterteilt ist, wird im Folgenden auf diese Bezug genommen um zu erläutern, in wie fern diese Anforderungen im Sequenzdiagramm repräsentiert sind. Das loggen ist nicht an jeder Stelle an der ein Loggen sinnvoll / nötig ist, aufgeführt, um die im Sequenzdiagramm wichtigsten Abläufe zu verdeutlichen.

•Anforderung 13

◦ Die Lebenszeit wird im cfg-File gesetzt, welches der Client in der Funktion readConfig() auswertet. Das er nach Ablauf der Lebenszeit terminiert, ist durch die äußerste Schleife realisiert.

•Anforderung 14

◦ Die Steuernden Werte des Clients werden in der Funktion readConfig() des Clients gelesen. Dass der Client aus nur einem Prozess besteht und seine Rollen als Redakteur und Leser sequentiell ausführt ist in den Sequenzdiagrammen A und B dargestellt.

Redakteur-Client

Der Ablauf des Redakteur-Clients ist im Sequenzdiagramm A dargestellt.
Der Client ist Redakteur-Client, sofern seine Rolle „redakteur“ ist. Ist dies der Fall, läuft er in seiner innersten Schleife.

•Anforderung 1

◦ Das der Redakteur-Client die eindeutige ID vor versenden einer Nachricht anfordert ist durch das INNRflag sichergestellt, welches in der ersten Alternative der innersten Schleife abgefragt wird und auf false gesetzt wird, sobald der Redakteur-Client eine Nachricht gesendet hat. Das er diese Nr. seiner Nachricht voranstellt ist am Nachrichtenpfeil zu sehen, der von der ersten Alternative des Clients zum Server führt.

◦ Siehe auch Überschrift „Server“ Anforderung 1.

•Anforderung 9

◦ Die Zeitabstände des Sendens von Nachrichten ist realisiert durch die 2 Bedingungen der ersten Alternative: [TimeNow – TimeLastSending >= Sendeintervall && INNRflag == true]. Das Ergebnis der ersten Auswertung (von links) stellt sicher, dass er keine Nachricht vor Ablauf der festgelegten Zeit sendet, das Ergebnis der zweiten Auswertung stellt sicher, dass er eine globale Nachrichtennummer hat (INNR).

•Anforderung 10

◦ Die Anpassung des Abstandes wird in der zweiten Alternative der innersten Schleife realisiert. Da in der ersten Alternative die Variable „TransmittedMessages” um 1 erhöht wird, wenn eine Nachricht gesendet wurde, wird in der zweiten Alternative durch die Bedingung [TransmittedMessages == 5] die Variable auf 0 gesetzt und die Funktion „changeSendInterval(Sendinterval)“ aufgerufen um den Zeitabstand anzupassen. Innerhalb der Funktion muss dafür gesorgt werden, dass die Zeit nicht unter 2 Sekunden rutscht. Nachdem der Zeitabstand angepasst wurde, wechselt der Client in seine Rolle als Leser und springt sogleich aus der innersten Schleife um seine Rolle als Leser-Client auszuführen. Der Ablauf des Leser-Clients ist in Sequenzdiagramm B dargestellt, auf welches mit einer reference aus dem Sequenzdiagramm A heraus verwiesen wird.

•Anforderung 11

◦ Die eindeutige Nachrichtennummer wird in der zweiten Alternative der innersten Schleife durch das übersenden der message „{self(), getmsgid} angefordert. Daraufhin erhält er diese eindeutige Nachrichtennummer vom Server.

Leser-Client

Der Ablauf des Leser-Clients ist im Sequenzdiagramm B dargestellt. Auf dieses Sequenzdiagramm wird aus dem Sequenzdiagramm A heraus verwiesen.

•Anforderung 12

◦ Durch die zweite Bedingung der ersten Alternative der innersten Schleife ist sichergestellt, dass der Client seine Rolle als Leser einnimmt. Dass er Textzeilen abfragt bis er alle erhalten hat, ist sichergestellt durch das „Terminated"-Flag der Bedingung der ersten Alternative. Diesen Wahrheitswert erhält er durch die ihm zuletzt übermittelte Nachricht. Die GUI ist realisiert durch das Log-File des Leser-Clients. Die Ausgabe der Nachrichten erfolgt in der Funktion logging(Datei, Inhalt) (aus dem Modul „werkzeug.erl") der Alternative.

Server

Der Ablauf des Servers in Verbindung mit der Rolle des Redakteurs des Clients ist in Sequenzdiagramm A dargestellt. In Sequenzdiagramm B ist sein Ablauf in Verbindung mit der Rolle des Lesers dargestellt.

•Anforderung 1

◦ Durch die Variable INNR := 1 ist sichergestellt, dass die eindeutige ID für jede Textzeile vorhanden ist. Das der Redakteur-Client diese Nummer erhält sobald er diese anfordert ist in der zweiten Bedingung der Alternative zu sehen. Durch die Inkrementierung der Variablen INNR um 1 ist sichergestellt, dass nach seiner Anforderung kein anderer Redakteur-Client dieselbe Nr. bekommt. Siehe auch Überschrift „Redakteur-Client" Anforderung 1.

•Anforderung 2

◦ Deliveryqueue und Holdbackqueue sind im Sequenzdiagramm A dargestellt. Der Server initialisiert die HBQ welche wiederum die DLQ initialisiert. Hat dies geklappt, sendet die HBQ dem Server die Nachricht {reply, ok}. Die Größe der DLQ wird in der cfg-Datei festgelegt, welche wiederum in der Funktion readConfig() gelesen wird. Das in der HBQ Nachrichten stehen, die nicht ausgeliefert werden dürfen, ist sichergestellt, indem die HBQ in ihrer ersten Alternative die ihm vom Server übermittelte INNR mit der als nächstes erwarteten Nr. der DLQ vergleicht. Stimmen diese überein, fügt die HBQ die erhaltene Nachricht mit der Funktion push2DLQ([NNr, Msg, TSclientout, TShbqin], Queue, Datei) in die DLQ ein und erhält eine modifizierte DLQ. Stimmen diese nicht überein, speichert die HBQ die Nachricht zwischen. Ob die in der DLQ vorhandenen Nachrichtenfolgen durch die in der HBQ vorhandenen Nachrichten fortgeführt werden können, prüft die HBQ in der Funktion pushSeries(HBQ, DLQ).

•Anforderung 3

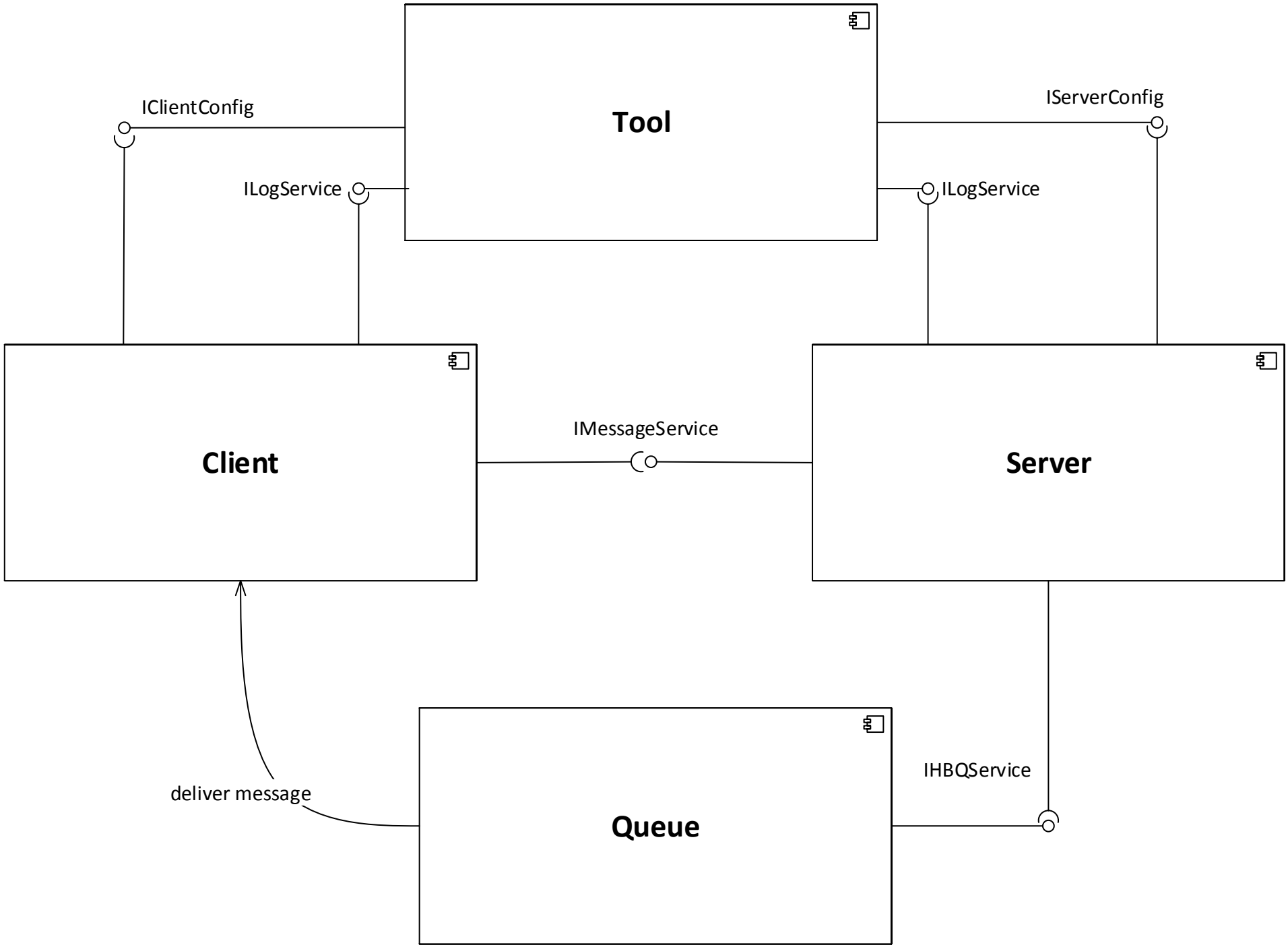
◦ Die Empfangszeit wird vom Server beim Senden der Nachricht an die HBQ eingetragen. Die HBQ trägt in der Funktion push2DLQ([NNr, Msg, TSclientout, TShbqin], Queue, Datei) die Übertragungszeit beim Einfügen in die DLQ ein oder in der Funktion pushSeries(HBQ, DLQ) wenn die Nachricht erst einmal in der HBQ zwischengespeichert werden musste.

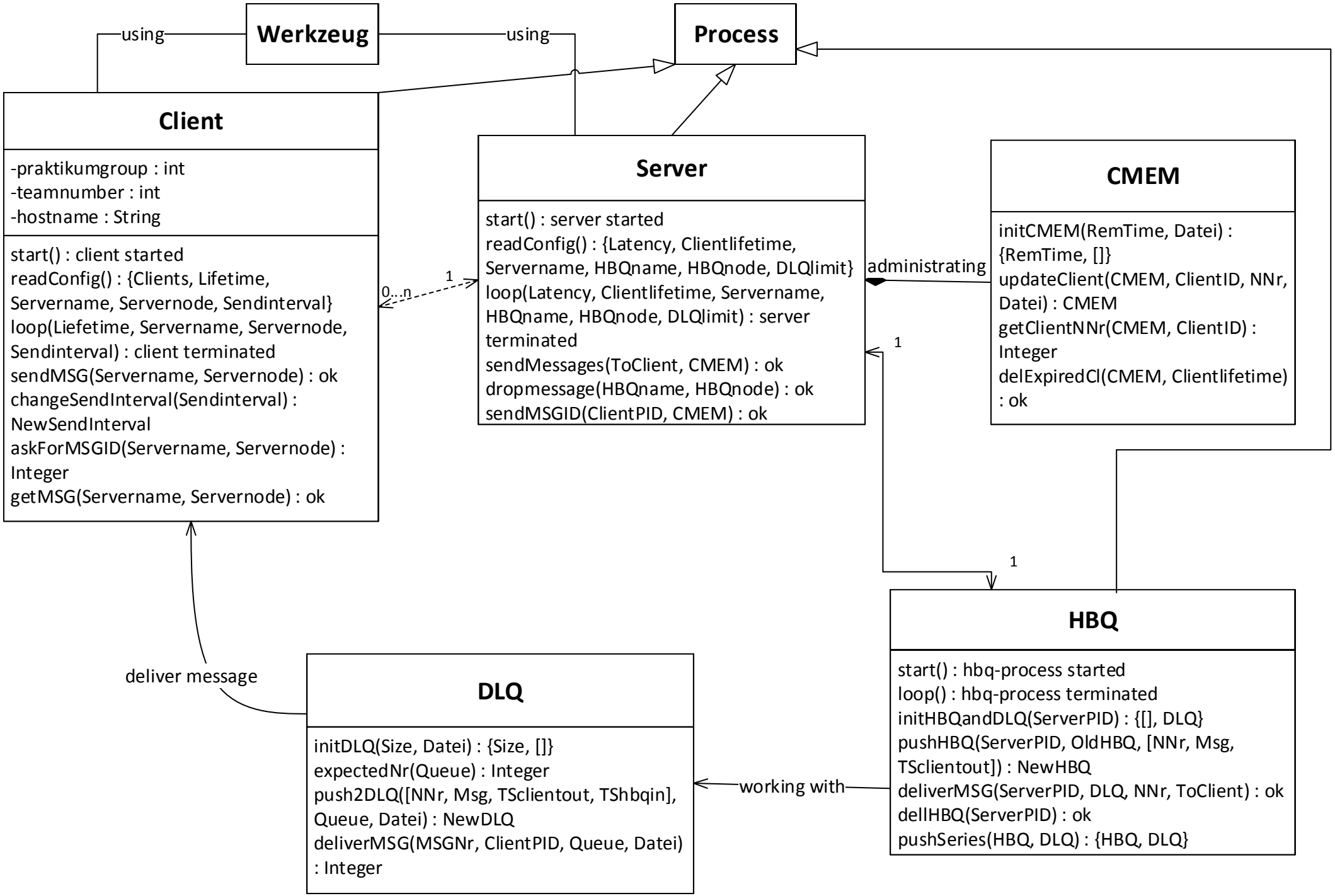
•Anforderung 4 / 5

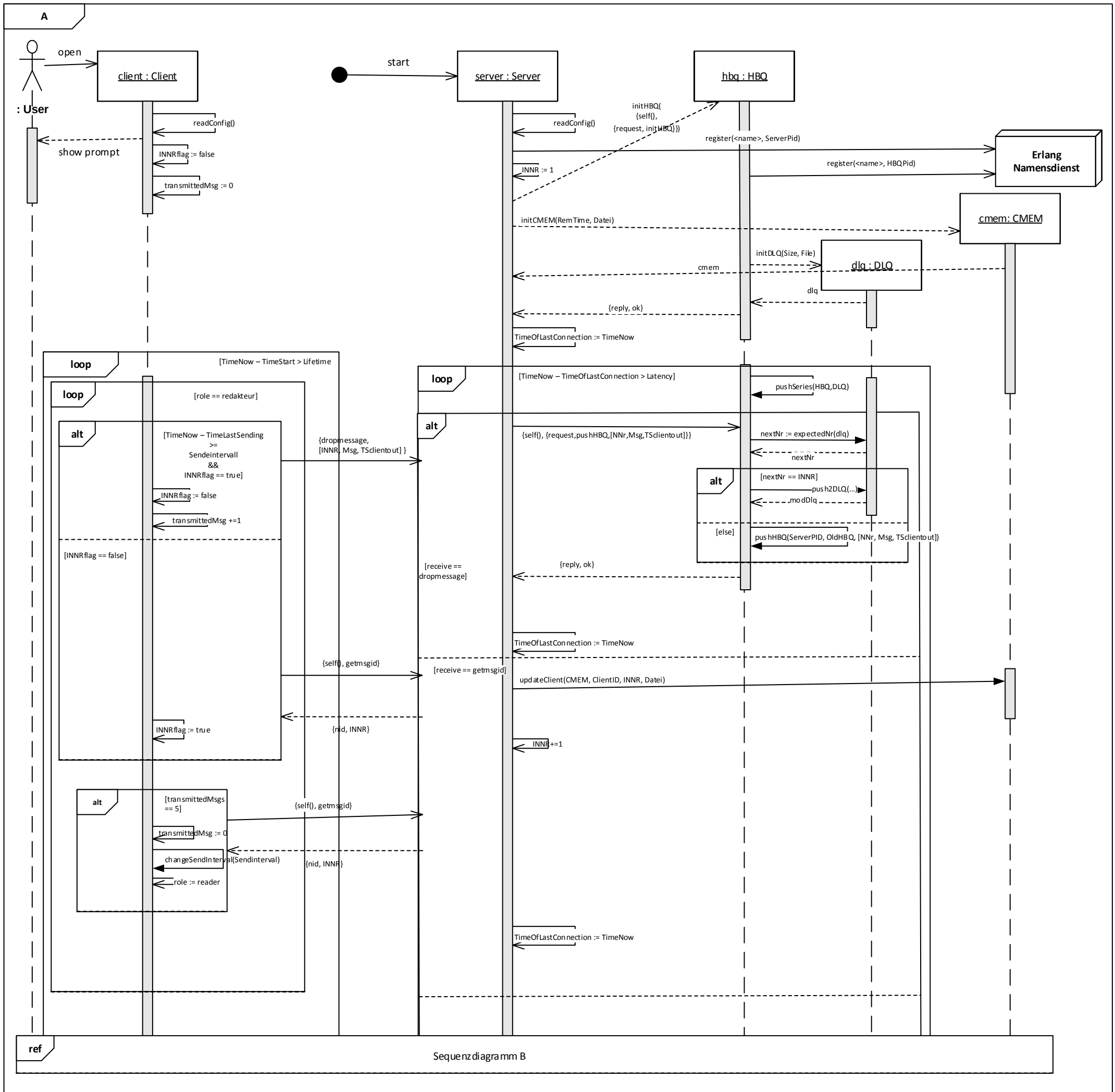
◦ Im Sequenzdiagramm B ist dargestellt, dass der Leser-Client auf Anfrage eine noch nicht an ihn ausgelieferte und beim Server bekannte Textzeile bekommt. In der Bedingung [receive == getmessages] der Alternative des Servers wird über die Funktion getClientNNR(CMEM, ClientID) auf dem CMEM diese Nummer angefordert. Da der CMEM nach einer gewissen Zeit Leser-Clients

vergessen soll, überprüft der Server mit der Funktion delExpiredCl(CMEM, Clientlifetime) auf dem CMEM die abgelaufenen Clients, welche dann entfernt werden. Der Server ruft im Falle NNR == 1 die Funktion updateClient(CMEM, ClientID, NNr, Datei) auf dem CMEM auf, was dazu führt, dass der Client im CMEM gemerkt wird. Nachdem der Server die NNR erhalten hat, sendet der Server die Nachrichtenanforderung an die HBQ welche wieder die Funktion deliverMSG(MSGNr, ClientPID, Queue, Datei) auf der DLQ aufruft. Die Nachricht erhält der Leser-Client anschließend von der DLQ.

- Anforderung 6
  - In der Funktion pushSeries(HBQ, DLQ) der Schleife der HBQ wird der Fall behandelt, der in Anforderung 6 beschrieben ist.
- Anforderung 7
  - In der Funktion readConfig() des Servers die gleich nach dem Start des Servers aufgerufen wird, wird die Wartezeit des Servers gelesen und die Variable Latency mit diesem Wert gesetzt. Die Variable TimeOfLastConnection beinhaltet die Zeit der Letzten Client-Verbindung mit dem Server und wird immer neu gesetzt, wenn ein Client eine Anforderung an den Server stellt. Sie wird also im Sequenzdiagramm A initialisiert und neu gesetzt sowie auch im Sequenzdiagramm B neu gesetzt (also sowohl bei Leser- als auch bei Redakteur-Verbindungen / Nachrichten).
- Anforderung 8
  - Die Datenstrukturen HBQ, DLQ und CMEM sind in den Sequenzdiagrammen A und B dargestellt.
- Anforderung 9
  - Der Server registriert sich nach dem Aufruf von readConfig() beim Namensdienst von Erlang, wie im Sequenzdiagramm A und B dargestellt.


Tool
IClientConfig
IServerConfig
ILogService
ILogService
Client
IMessageService
Server
deliver message
Queue
IHBQService


Werkzeug
using
using
Process
Client
-praktikumgroup : int
-teamnumber : int
-hostname : String
start() : client started
readConfig() : {Clients, Lifetime, Servername, Servernode, Sendinterval}
loop(Liefetime, Servername, Servernode, Sendinterval) : client terminated
sendMSG(Servername, Servernode) : ok
changeSendInterval(Sendinterval) : NewSendInterval
askForMSGID(Servername, Servernode) : Integer
getMSG(Servername, Servernode) : ok
0...n
1
Server
start() : server started
readConfig() : {Latency, Clientlifetime, Servername, HBQname, HBQnode, DLQlimit}
loop(Latency, Clientlifetime, Servername, HBQname, HBQnode, DLQlimit) : server terminated
sendMessages(ToClient, CMEM) : ok
dropmessage(HBQname, HBQnode) : ok
sendMSGID(ClientPID, CMEM) : ok
administrating
CMEM
initCMEM(RemTime, Datei) : {RemTime, []}
updateClient(CMEM, ClientID, NNr, Datei) : CMEM
getClientNNr(CMEM, ClientID) : Integer
delExpiredCl(CMEM, Clientlifetime) : ok
1
1
HBQ
start() : hbq-process started
loop() : hbq-process terminated
initHBQandDLQ(ServerPID) : {[], DLQ}
pushHBQ(ServerPID, OldHBQ, [NNr, Msg, TSclientout]) : NewHBQ
deliverMSG(ServerPID, DLQ, NNr, ToClient) : ok
dellHBQ(ServerPID) : ok
pushSeries(HBQ, DLQ) : {HBQ, DLQ}
working with
DLQ
initDLQ(Size, Datei) : {Size, []}
expectedNr(Queue) : Integer
push2DLQ([NNr, Msg, TSclientout, TShbqin], Queue, Datei) : NewDLQ
deliverMSG(MSGNr, ClientPID, Queue, Datei) : Integer
deliver message


A
: User
open
client : Client
readConfig()
show prompt
INNRflag := false
transmittedMsg := 0
start
server : Server
readConfig()
initHBQ(
{self(),
{request, initHBQ}})
hbq : HBQ
register(<name>, ServerPid)
register(<name>, HBQ.Pid)
Erlang
Namensdienst
INNR := 1
initCMEM(RemTime, Datei)
cmem: CMEM
initDLQ(Size, File)
dlq : DLQ
cmem
dlq
{reply, ok}
TimeOfLastConnection := TimeNow
loop
[TimeNow – TimeStart > Lifetime
loop
[role == redakteur]
alt
[TimeNow – TimeLastSending
>=
Sendeintervall
&&
INNRflag == true]
INNRflag := false
transmittedMsg +=1
[INNRflag == false]
{dropmessage,
[INNR, Msg, TSclientout] }
loop
[TimeNow – TimeOfLastConnection > Latency]
pushSeries(HBQ,DLQ)
alt
{self(), {request,pushHBQ,[NNr,Msg,TSclientout]}}
nextNr := expectedNr(dlq)
nextNr
alt
[nextNr == INNR]
push2DLQ(...)
modDlq
[else]
pushHBQ(ServerPID, OldHBQ, [NNr, Msg, TSclientout])
{reply, ok}
[receive ==
dropmessage]
TimeOfLastConnection := TimeNow
{self(), getmsgid}
[receive == getmsgid]
updateClient(CMEM, ClientID, INNR, Datei)
INNRflag := true
{nid, INNR}
INNR+=1
alt
[transmittedMsgs
== 5]
{self(), getmsgid}
transmittedMsg := 0
changeSendInterval(Sendinterval)
role := reader
{nid, INNR}
TimeOfLastConnection := TimeNow
ref
Sequenzdiagramm B

B

: User

open

client : Client

readConfig()

show prompt

INNRflag := false

transmittedMsg := 0

server : Server

ref

Sequenzdiagramm A

hbq : HBQ

dlq : DLQ

cmem: CMEM

loop

[TimeNow – TimeStart > Lifetime

loop

[TimeNow – TimeOfLastConnection > Latency]

delExpiredCl(CMEM, Clientlifetime)

ok

alt

ref

Sequenzdiagramm A

[receive == getmessages]

loop

[role == reader]

alt

[Terminated]

{self(), getmessages}

role := redakteur

logging(Datei, Inhalt)

getClientNNR(CMEM, ClientID)

alt

[NNR == 1]

NNR

updateClient(CMEM, ClientID, NNR, File)

{self(), {request,deliverMSG,NNr,PID}}

deliverMSG(MSGNr, ClientPID,Queue,Datei)

{reply,[NNr,Msg,TSclientout,TShbqin,TSdlqin,TSdlqout],Terminated}

TimeOfLastConnection := TimeNow

{self(), {request,delIHBQ}}

ok