Team: 01, Francis Opoku, Fabian Reiber

Aufgabenaufteilung:

1. Sequenzdiagramme und zugehörige Doku, sowie restliche Dokumentation. Server, CMEM, Lifetimetimer und HBQ implementiert. Bugs gefixt und getestet.
2. Klassen- und Komponentendiagramm, zugehörige Doku, sowie restliche Dokumentation. Client, MessagSendTimer, Lifetimetimer und DLQ implementiert. Bugs gefixt und getestet.

Quellenangaben: http://learnyousomeerlang.com/what-is-otp#the-basic-server
http://erlang.org/doc/man/erl.html

Begründung für Codeübernahme: keine Codeübernahme

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 und Testen:
01.04.15 – 08.04.15: Fabian Reiber ca. 12 Std.
01.04.15 – 08.04.15: Francis Opoku ca. 14-15 Std.

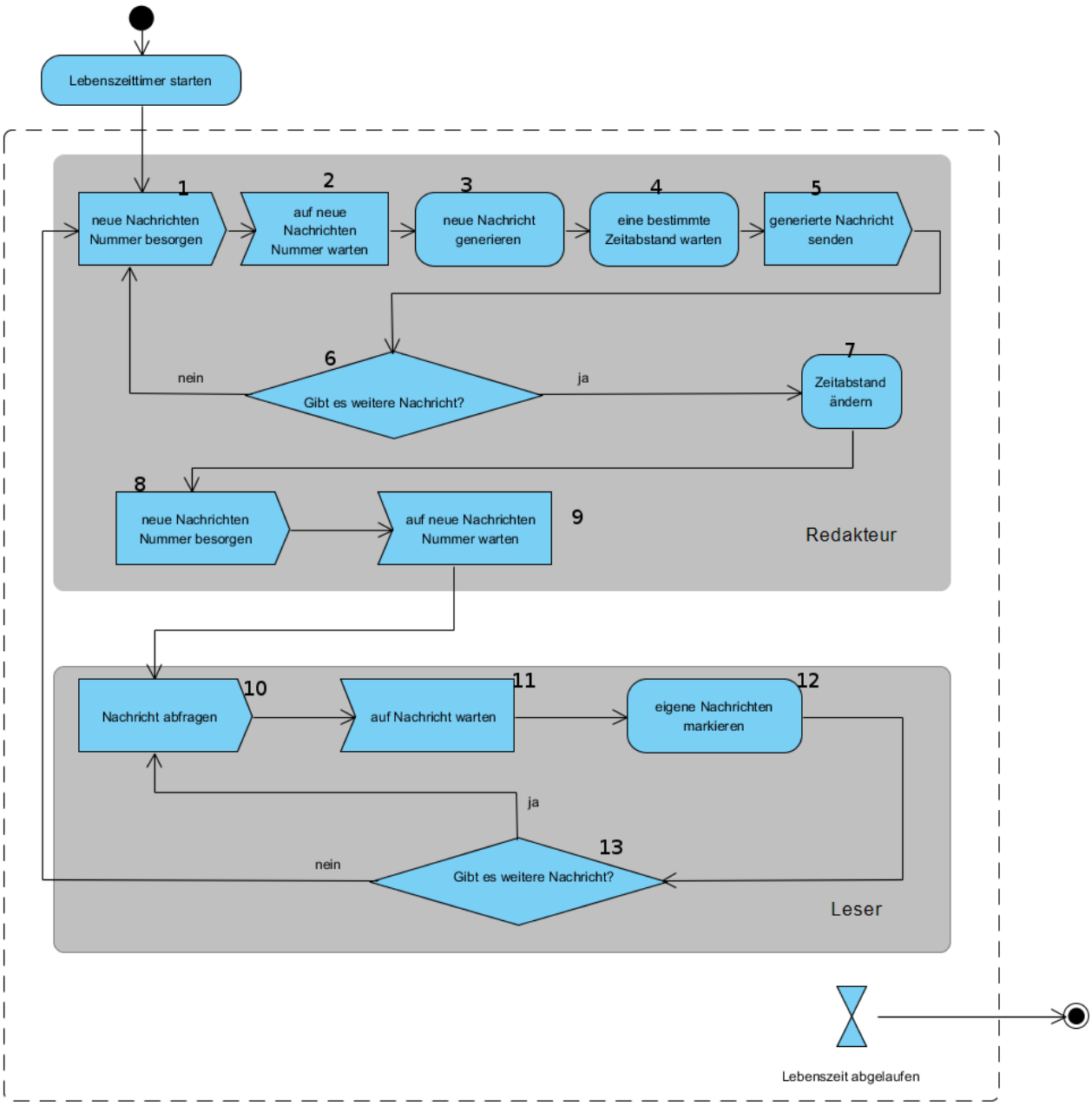
Aktueller Stand:
27.03.15: Entwurf ist fertig. Implementierung wurde noch nicht angefangen, da Fremdentwurf verwendet werden soll.
08.04.15: Implementierung ist nun fertig und getestet.

Änderungen im Entwurf: die Änderungen sind in einem extra Abschnitt „Dokumentation zum Entwurf“ (siehe unten) festgehalten.

Entwurf: ist in der PDF-Datei “12_VSP1E.pdf” enthalten.

Dokumentation zum Entwurf

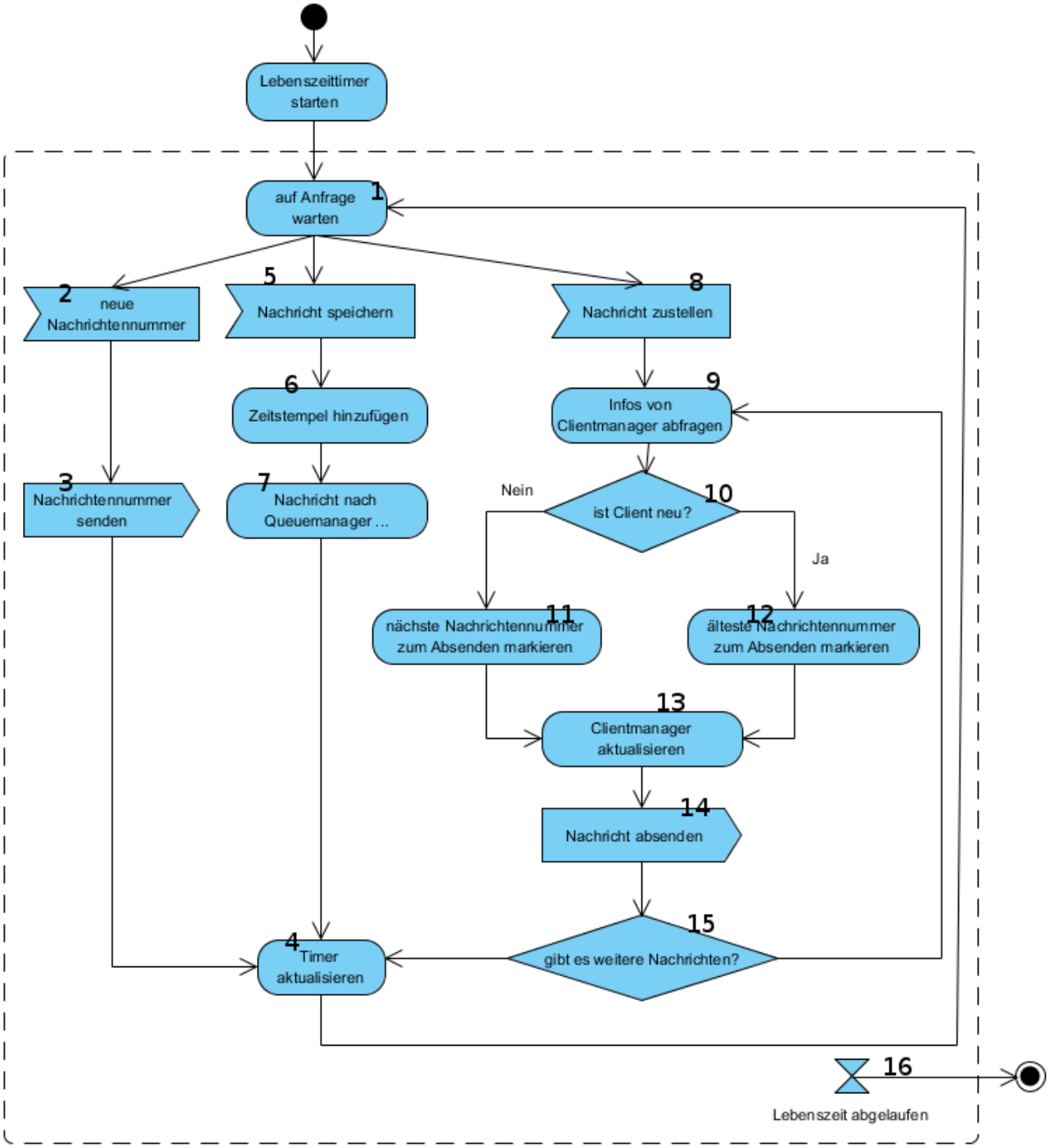
Client:



Die Schritte wurden im Diagramm nummeriert und sind im Code in den Kommentaren wieder gespiegelt. Die Lebenszeit wird durch das Modul “lifetimeTimer” gesetzt. Sobald diese abgelaufen ist, erhält der Client vom internen Erlang Timer eine vorher definierte Nachricht ({interrupt, timeout}) und beendet sich.

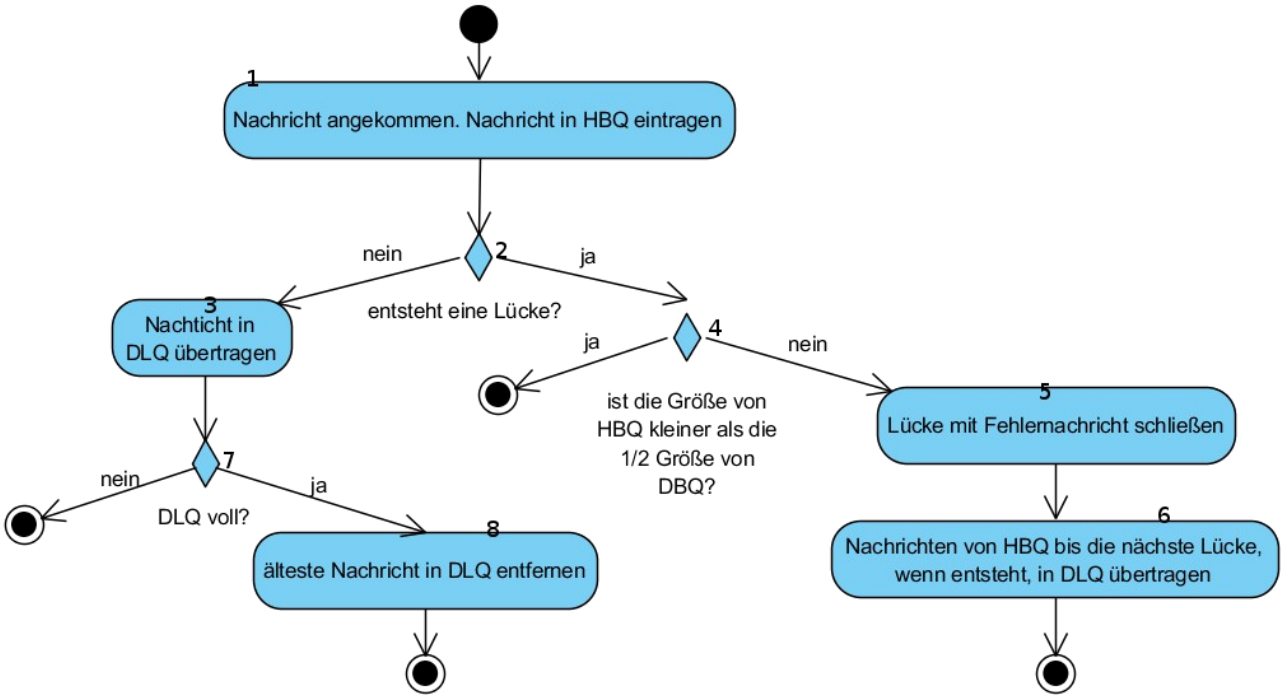
Server:

Im Entwurf steht, dass der Server aus 2 Prozessen besteht: HBQ, DLQ und Server als ein Prozess und CMEM als zweiten Prozess. Da dies jedoch den Anforderungen widerspricht, haben wir in Rücksprache mit dem Team 12 den CMEM nicht als Prozess realisiert. Der Server besteht also immernoch aus 2 Prozessen – dem Server und der HBQ.



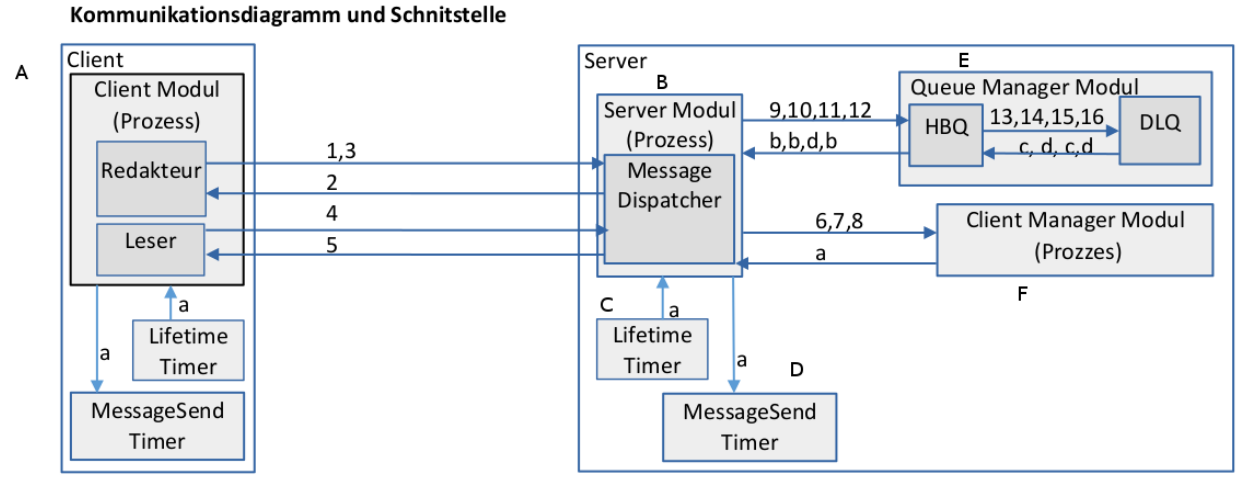
Die Schritte wurden im Diagramm nummeriert und sind im Code in den Kommentaren wieder gespiegelt. Die Lebenszeit wird durch das Modul “lifetimeTimer” gesetzt. Sobald diese abgelaufen ist, erhält der Server vom internen Erlang Timer eine vorher definierte Nachricht ({srvtimeout}) und beendet sich.Der Schritt 15. ist falsch.Würde dieser Schritt so umgesetzt werden, würde der Server in einer Schleife bleiben, bis alle Nachrichten abgeholt wurden, ohne dass der Client, der zu Beginn dieses Durchlaufs die Anfrage einer neuen Nachricht gestellt hat, diese Anfrage erneut stellen müsste. Dies widerspricht den Anforderungen, dass der Client jede Nachricht einzelnd abfragt. Theoretisch wäre es sogar möglich, dass der Server terminiert, obwohl er permanent damit beschäftigt ist, Nachrichten zuzustellen, da der Timer nur aktualisiert werden würde, wenn es keine weiteren Nachrichten gibt.

HBQ-DLQ-Algorithmus



Die Schritte wurden im Diagramm nummeriert und sind im Code in den Kommentaren wieder gespiegelt.

Kommunikationsdiagramm:



a. <<create>>
b. {reply, ok}
c. {queue, Queue}
d. {reply, SendNNr}

A: Die Komponente Client ist als eigenständiger Prozess realisiert und entspricht dem Kommunikationsdiagramm. Dieser Client-Prozess wird durch das Modul “lifetimeTimer” gestartet. Die Lebenszeit wird ebenfalls in diesem Modul gesetzt, indem ein Timer gestartet wird, welcher den Client unterbricht sobald die Zeit abgelaufen ist. Der “MessagSendTimer” ist in einem eigenen Modul implementiert und berechnet das Sendeintervall neu,nachdem 5

Nachrichten vom Redakteu gesendet wurden. Redakteur und Leser werden sequentiell, wie dargestellt, ausgeführt. Die Komponente entspricht also dem Kommunikationsdiagramm.

B: Der Server ist ein eigener Prozess und wird durch den lifetimeTimer gestartet. Der MessageDispatcher ist durch einen receive-Block in der loop-Funktion des Server-Moduls realisiert.

C: Der LifetimeTimer ist ein eigenes Modul: lifetimeTimer.erl.

D: Der “MessageSendTimer” in der Serverkomponente macht hier keinen Sinn, da der Redakteur-Client von sich aus die Nachrichten an den Server sendet und selber das Intervall neu berechnet. Auf der anderen Seite hat der Server mit einem Sendeintervall nichts zu tun, da der Leser-Client beim Server nach Nachrichten anfragt und der Server bzw. die DLQ die Nachrichten einzeln an den Leser-Client ausliefert.

E: Das Queue-Manager-Modul besteht aus dem Modulen HBQ und DLQ. Die HBQ ist ein eigener Prozess und gleichzeitig ein ADT, die DLQ ist nur ein ADT. Dies ist im Kommunikationsdiagramm nicht dargestellt, widerspricht deshalb den Anforderungen. Mit Team 12 wurde diese Änderung in der Implementierung besprochen.

F: Der Clientmanager ist kein Prozess (mit Team 12 abgesprochen) da dies den Anforderungen widersprach. Der Clientmanager ist laut Team 12 äquivalent zum CMEM und ist deshalb als CMEM bezeichnet.

Im Folgenden wird auf die Anforderungen, die Team 12 der Aufgabenstellung entnommen hat und im Kommunikationsdiagramm in den Nummern widergespiegelt sind, bezug genommen.

Alle Nummern entsprechen unserer Implementierung, abgesehen von den folgenden Nummern:
5.) Der Client erhält eine Nachricht nicht vom Server sondern von der DLQ. Diese Änderung ist mit Team 12 abgesprochen.
<<create>> (MessageSendTimer) in Clientkomponente: An dieser Stelle wird kein MessageSendTimer erstellt, sondern die Methode in diesem Modul aufgerufen, die das Sendeintervall des Redakteur-Clients neu berechnet.
<<create>> (Message Dispatcher) in Serverkomponente: Dieses Erstellen fällt an dieser Stelle weg, da das Client Manager Modul kein eigenständiger Prozess ist.