Verteilte Systeme
Übungsblatt 2

Kilian Bartz (Mknr.: 1538561)

Hinweis. Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/vertsys_ueb02`.

Aufgabe 2

Zur Lösung dieser Aufgabe habe ich eine Klasse *Actor* angelegt, die von *Node* erbt. Jede Instanz von *Actor* verwaltet ihren aktuellen Zustand (`boolean active`), ihren Namen, die Namen der anderen Aktoren (`other_participants`) und ihre aktuelle Wahrscheinlichkeit zum Propagieren von Nachrichten (p). Zusätzlich haben sie die Möglichkeit, einer zufälligen Teilmenge der anderen Aktoren eine Nachricht zu senden. Dazu wird `other_participants` gemischt und allen Einträgen bis zu einem zufälligen Index eine Nachricht gesendet. Weil der eigene Name nicht in diesem Array vorhanden ist, wird auch gewährleistet, dass ein Aktor sich selbst keine Nachricht sendet.
Nachdem die Simulation gestartet wurde, versendet jeder Aktor nach einer zufälligen Wartezeit (zwischen 0 und 4sek) wie in der Aufgabe beschrieben eine Nachricht an eine zufällige Menge anderer Aktoren. Nach dem Versenden wechseln die Aktoren in den passiven Zustand, bis sie eine Nachricht erhalten. Eine erhaltene Nachricht wird mit einer stets schrumpfenden Wahrscheinlichkeit ($p = \frac{p}{2}$ bei jeder empfangenen Nachricht) propagiert. Der *Random Number Generator* eines Aktors wird mit dem Hashcode des Namen als Seed initialisiert, sodass ein Aktor mit einem spezifischen Namen sich immer gleich verhält und das Experiment somit reproduzierbar wird.

Experiment

Die Simulation wurde mit $n = 1000$ Aktoren für $4s, 5s, 20s$ Simulationsdauer durchgeführt. Anschließend wurden 2min abgewartet, bis keine neue `InterruptedException` mehr ausgegeben wurde. Ab einem Zeitpunkt t wird ein stabiler Zustand erreicht, das verteilte System terminiert also, was jedoch vom Simulator nicht festgestellt werden kann, da alle Aktoren noch immer in einer Endlosschleife auf neue Nachrichten warten. Somit terminiert auch die gesamte Java-Applikation (diese wartet via `.join()` auf alle Aktoren) nicht nach `sim.shutdown()` und muss stattdessen manuell beendet werden. Durch das Auswerten der Logdateien mithilfe von `CountOcc.java` lassen sich die Zahl der insgesamt gesendeten und empfangenen Nachrichten rekonstruieren (Tabelle 1).

Simulationsdauer	# Nachrichten gesendet	# Nachrichten empfangen
4s	1,336,256	1,189,151
5s	1,498,634	1,498,634
20s	1,498,634	1,498,634

Tabelle 1: Summe der insgesamt gesendeten und empfangenen Nachrichten nach 4s, 5s und 6s Simulationszeit

Man sieht, dass nach 4s Simulationszeit noch nicht alle gesendeten Nachrichten empfangen wurden. Teilweise werden diese ihrerseits noch weiterpropagiert (dadurch kommt es bei längerer Simulationszeit zu einer größeren Anzahl), sodass die Simulation an dieser Stelle auf jeden Fall noch nicht terminiert ist.
Nach 5s entspricht zwar die Zahl der gesendeten Nachrichten den empfangenen Nachrichten, jedoch kann durch diese einfache Zählung bekanntermaßen die verteilte Terminierung nicht garantiert werden, da potenziell Nachrichten aus der Zukunft (deren Empfang, aber nicht das Senden gezählt wurde) und Nachrichten in die Zukunft (deren Senden, aber nicht ihr Empfang gezählt wurde) miteinander verrechnet werden können. So kann es dazu kommen, dass das verteilte System zum Ende der Simulationszeit noch nicht terminiert ist, obwohl die Zahlen der gesendeten und empfangenen Nachrichten gleich ist.
Durch die Reproduzierbarkeit des Experiments durch die festen Seeds der Zufallsgeneratoren lässt sich nun das Doppelzählverfahren simulieren, indem die Anzahl der gesendeten und empfangenen Nachrichten bei einem Experiment

mit längerer Simulationsdauer erneut bestimmt wird. Die Zeitpunkte, an denen die `NetworkConnections` der Aktoren interrupted werden, stellt dabei den Zeitpunkt dar, an dem ein Observer die Zahl der gesendeten und empfangenen Nachrichten beim Aktor erfragen würde.
Dazu habe ich die Simulation hinreichend (auf 20s) verlängert und nach dieser Zeit erneut die Simulation gestoppt und anschließend die Zahl der Nachrichten in der entsprechenden Logdatei gezählt. Da dies in den gleichen Zahlen resultiert, lässt sich darauf schließen, dass der Terminierungszeitpunkt t dieser Simulation mit $n = 1000, p \mapsto \frac{p}{2}$ und Wartezeit $\in [0, \ldots, 4s]$ auf dem Testsystem zwischen 4s und 5s liegt.

Aufgabe 3

Wie bereits in Aufgabe 2 simuliert, entschied ich mich dazu, dass Doppelzählverfahren zur Feststellung der verteilten Terminierung zu implementieren. Dazu wurde ein `Observer` angelegt, der periodisch die Zahl der versendeten und empfangenen Nachrichten bei allen Aktoren erfragt. Falls diese beiden Zahlen nicht gleich sind, wird nach einer Pause von einer Sekunde die nächste Anfrage gestartet. Andernfalls wartet der Observer, bis sich diese Zahlen bei zwei aufeinander folgende Anfragen x_1, x_2 mit $x_i = (\text{ges. versendet zu Zeitpunkt } i, \text{ges. empfangen zu Zeitpunkt } i)$ entsprechen. Gilt nun $x_1 = x_2$, so ist das System nach dem Doppelzählverfahren terminiert. Nun kann eine Terminierungsnachricht an alle Aktoren versendet werden, sodass die Aktoren nach dem Empfangen dieser Nachricht aus der `receive()`-Schleife ausbrechen können.
Zur Unterscheidung zwischen Basis- und Observer-Nachrichten wurde im Nachrichten-Header ein Typ angelegt, der entweder den Wert `normal_message` oder `observer_message` annehmen kann. Somit kann garantiert werden, dass Aktoren nicht durch das Empfangen einer Observer-Nachricht in einen aktiven Zustand übergehen können. Observer-Nachrichten enthalten außerdem im Nachrichten-Body ein Argument `command`. Entspricht dieses `query`, übermittelt der Aktor die nun erfassten Zahlen gesendeter Nachrichten und empfangener Nachrichten. Handelt es sich um den `stop`-Befehl bricht der Aktor stattdessen aus der `receive()`-Schleife aus. Somit kann der gesamte Simulator sauber terminieren, nachdem alle Aktoren diese Nachricht empfangen haben.

Experiment

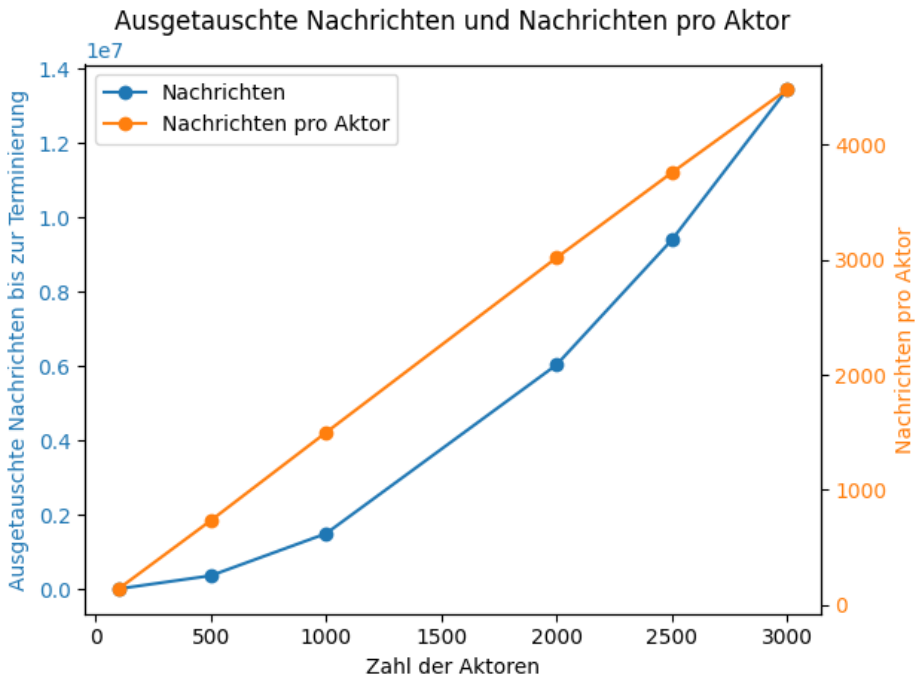

Abbildung 1: Zahl der ausgetauschten Nachrichten und durchschnittliche Nachrichten pro Aktor

Zur Verifizierung der Ergebnisse von Aufgabe 2 wurde das Experiment zunächst analog mit $n = 1000$ und einer unbegrenzten Simulationszeit (`simulator.simulate()`) durchgeführt. Der Simulator terminiert, nachdem der Observer die Terminierung des verteilten Systems festgestellt hat und infolgedessen alle Aktoren nach Empfang einer `stop`-Nachricht aus ihrer `receive()`-Schleife ausgebrochen sind. Dies dauert etwa 1:14min und passiert erneut nachdem **1,498,634** Nachrichten gesendet und empfangen wurden. Somit ist ersichtlich, dass das Doppelzählverfahren zwar durchaus korrekt durch Analyse der Logdatei "von außen" wie in Aufgabe 2 durchgeführt werden kann, jedoch ein Observer in der Simulation benötigt wird, um die Terminierung innerhalb der Simulation festzustellen und basierend darauf dann weitere Aktionen (z. B. Beenden der Aktoren) durchführen kann.
Zusätzlich wurde das Experiment mit $n = 100, n = 500, n = 2000, n = 2500, n = 3000$ durchgeführt. Wie in Abbildung 1 zu sehen, steigt die Zahl der ausgetauschten Nachrichten exponentiell und die Zahl der Nachrichten pro Aktor linear mit wachsender Teilnehmerzahl an. Dies zeigt die Problematik von Nachrichtenstürmen eines verteilten Systems mit vielen Teilnehmern auf. Dadurch, dass die Zahl der Nachrichten, die jeder Aktor senden und empfangen muss, so schnell steigt, würde in einem realen System ab einem bestimmten Schwellwert bzgl. der Teilnehmeranzahl der Aufwand der Nachrichtenabfertigung den Rechenaufwand der eigentlichen Aufgabe, welche die Aktoren bearbeiten sollen, übersteigen. Zusätzlich käme in einem realen System der beträchtliche Aufwand der Verwaltung (v. a. Initialisierung) eines solchen verteilten Systems dazu.