Übung 1

Repository: `https://github.com/Raphael90K/DSA2`

1 Aufgabe 2

Zur Lösung der Aufgabe wurde eine Klasse `NetworkNode`, die von der Klasse `Node` erbt, entworfen. Diese Klasse repräsentiert die Netzwerkteilnehmer. Sie wurde so implementiert, dass sie für Aufgabe 3 ebenfalls unverändert übernommen werden kann. Die initial Wahrscheinlichkeit, ob Nachrichten versandt werden, wird für alle Knoten aus einem statischen Zufallsgenerator erzeugt. Dies macht das Experiment wiederholbar. In der `engage` Methode wird eine Schleife implementiert, die testet, ob der Knoten ein Feuerwerk empfängt und daher aktiviert bleibt oder wieder aktiviert werden muss. Für das Senden der Nachrichten werden die Methode `initSending` für die erste Iteration und `sendMessages` für die Nachfolgenden Iterationen nach einer erneuten Aktivierung verwendet. Initial wartet die Methode eine Zufallszeit zwischen null und einer Sekunde. Beide Methoden werden in einem separaten Thread ausgeführt, der durch die Klasse `SendHandler` implementiert wird. Dieses vorgehen wurde gewählt, sodass eintreffende „Feuerwerke“ unmittelbar abgerufen werden und nicht erst, nachdem der Knoten sich nach möglichen mehreren Iterationen deaktiviert. In diesem Fall wird der Knoten möglicherweise unmittelbar wieder aktiviert, durch eine Nachricht, die verspätet abgerufen wird. Sollte ein Knoten bei Erhalt eines Feuerwerks noch aktiv sein, bleibt er einfach weiter aktiv. Eine erneute Aktivierung erfolgt erst, sobald er ein „Feuerwerk“ erhält und zu diesem Zeitpunkt inaktiv ist.

Bei den Experimenten passierte es regelmäßig, dass die festgelegte Simulationszeit abgelaufen war und das Ende der Simulation auch mitgeloggt wurde, das Programm jedoch nicht terminierte. Bei der Analyse im Debugmodus wurde festgestellt, dass die Threads, die für die Knoten instanziiert wurden, beim Warten auf eine Nachricht stecken bleiben. Um dies zu beheben wurden die Threads in der Klasse `NetworkConnection` als „Daemon-Threads“[1] gestartet. Wie in der Aufgabenbeschreibung vorgegeben wird der Anfangswert von `p` nach jeder Iteration um die Hälfte verringert. Sobald

[1] `https://www.geeksforgeeks.org/daemon-thread-java/`

ein Knoten aus dem inaktiven Zustand zurückkehrt, wird der Anfangswert von `p` wieder hergestellt. Dies sorgt in Experimenten bei einer größeren Anzahl an Knoten dazu, dass die Knoten nahezu nicht verstummen. In der aktuellen Implementierung werden bei $N = 5$ in der Simulation nur eine Sekunden lang Nachrichten verschickt. Bei $N = 10$ sind es bereits 42 Sekunden.

2 Aufgabe 3

Für diese Aufgabe wurde der Observerknoten in der Klasse `ObserverNode` implementiert, der von `Node` erbt. Da es nur einen Observerknoten gibt, wird eine Instanz im Singleton-Pattern[2] erzeugt. Die Überprüfung ob das Netzwerk terminiert ist, erfolgt über das in der Vorlesung vorgestellte Doppelzählverfahren. Der Observer besitzt die folgenden Variablen:

- `sentCount`: zählt in jeder Iteration die gesendeten Nachrichten
- `reveiceCount`: zählt in jeder Iteration die erhaltenen Nachrichten
- `receiveInactiveCount`: zählt in jeder Iteration die Nachrichten mit dem Status `false`
- `roundsAllInactive`: zählt die Durchgänge in denen alle Knoten inaktiv waren.
- `roundsRequired`: Anzahl der Runden, die alle Knoten inaktiv sein müssen, bis die Terminierung erkannt wird.

In der `engage` Methode wird der Status aller Knoten per Broadcast abgefragt. Der Observer startet dann einen Thread der die eingehenden Nachrichten verarbeitet. Nach drei Sekunden wird dieser Thread beendet, dies soll das Programm auch bei verlorenen Nachrichten weiterführen. Im Thread wird die Methode `checkIncomingMessages` aufgerufen, die die Nachrichten verarbeitet und die Anzahl der empfangen Nachrichten und der inaktiven Knoten zählt. Sobald in der vorgegeben Anzahl an Runden (im Beispiel zwei) festgestellt wurde, dass so viele Knoten inaktiv sind, wie Nachrichten versendet wurden, wird die Terminierung festgestellt und die Simulation beendet.
Bei dieser Aufgabe erfolgte ebenfalls eine Änderung im Quellcode des Simulators. In der Klasse `Node` wurde die Methode `getLogger` eingeführt. Diese macht den Logger für Klassen, die von `Node`

[2] `https://javabeginners.de/Design_Patterns/Singleton_-Pattern.php`

erben, zugänglich. So können beliebige Logeinträge hinzugefügt werden und im aktuellen Fall die festgestellt Terminierung des Observers dokumentiert werden.
Im Vergleich zur ersten Aufgabe hat sich der Programmablauf nicht geändert, jedoch wird festgestellt, dass der Observerknoten die Terminierung festgestellt, nachdem die letzten Nachricht in der Konsole angezeigt wurde. Dies wird auch in der Logdatei eingetragen.