Ergebnisbericht

Verteilte Systeme – Übungsblatt 3

Maxim Hotz (1338419)

Dieser Ergebnisbericht ist wie folgt strukturiert: Zunächst wird kurz die Wahl des implementierten Agreement-Protokolls diskutiert. Anschließend werden die theoretischen Kernmechanismen und die damit verbunden Problemstellungen des Protokolls beschrieben. Abschließend wird dann anhand eines durch meine eigne Implementierung erzeugten Log-Files die Funktionsweise der konkreten Implementierung aufgezeigt und bestimmte Aspekte des Implementierungsdesigns erläutert.

Wahl des Protokolls

Wie in Aufgabe 1) beschreiben, habe ich mir zunächst die vorgeschlagenen Protokolle *Paxos* und *Raft* sowie deren konkrete Unterschiede zueinander angesehen. Es stellte sich schnell heraus, dass Raft generell als deutlich leichter zu verstehen gilt als Paxos, was auch durch eine konkrete Studie bestätigt wurde[1]. Darüber hinaus habe ich mir auch weitere transaktions-basierte Protokolle zur Erzielung von Konsens in einem Netzwerk angesehen, die aber in Punkto Fehlertoleranz und konkretem Anwendungsgebiet von den komplexeren Protokollen wie Paxos oder Raft abweichen. Konkret meine ich hier 2-Phase-Commit bzw. 3-Phase-Commit, die zwar deutlich einfacher zu implementieren sind, aber eher im Bereich von verteilten Datenbanken angewendet werden und im Gegensatz zu Paxos und Raft nicht so gut mit Ausfällen von Knoten zurechtkommen beziehungsweise im Fehlerfall menschliche Intervention benötigen. Daher kamen diese für die Bearbeitung dieser Aufgabe nicht in Frage. Durch weitere Recherche wurde dann meine Aufmerksamkeit auf den Einsatz von Byzantine Fault Tolerance Protokollen gelenkt, wie sie auch oftmals in Blockchain-Technologien eingesetzt werden. Ein konkretes Beispiel hierfür ist Tendermint, was sich selbst als „*The best-in-class BFT consensus engine for your blockchain*“[2] beschreibt. Generell hat sich mir hier auch die Frage gestellt, ob die Erreichung von Konsens in Blockchains durch allgemeine Proof-of-Work oder Proof-of-Stake Protokolle dem Ziel dieser Aufgabe gerecht werden würde. Da jedoch diese Protokolle speziell auf Blockchains zugeschnitten sind, bei denen das Verhindern von Angriffen/bewusstem Fehlverhalten und Sicherheit im Vordergrund steht, habe ich mich schlussendlich doch für die Implementierung eines Protokolls zur Anwendung in traditionellen verteilten Systemen entschieden. Hier musste ich nun die Wahl zwischen Paxos und Raft treffen. Auf Grund der leichteren Verständlichkeit/ geringeren Komplexität, als auch der meiner Meinung nach besseren Dokumentation/Verfügbarkeit von Ressourcen, habe ich mich letztendlich für Raft entschieden.

[1] https://raft.github.io/raft.pdf
[2] https://tendermint.com

Funktionsweise von Raft

Grundsätzlich hat Raft die Aufgabe, in verteilten Systemen Konsens über die gemeinsame Reihenfolge von Operationen über alle Netzwerk-Knoten zu erreichen. Dabei basiert Raft auf der Idee, dass immer nur ein Knoten des Clusters (Leader-Knoten) für die Koordination verantwortlich ist, während die anderen Knoten (Follower-Knoten) die Autorität des Leaders und damit auch seine Entscheidungen akzeptieren. Jeder Knoten verfügt dabei über eine „Replicated Sate Machine", welche den Zustand der Daten/Reihenfolge der Operationen speichert. Die Kommunikationsschnittstelle zur Außenwelt (Client) stellt somit der Leader dar. In anderen Worten: Jeder Befehl von außen (z.B. Einreichen eines neuen Commands) wird über den Leader abgewickelt.

Im Gegensatz zu Paxos lässt sich Raft in relativ unabhängige Subprozesse unterteilen, was die Verständlichkeit des Protokolls erheblich erleichtert. Im Folgenden werden diese Subprozesse getrennt voneinander erläutert.

1. Wahl des Leader-Knotens
2. Replikation des Logs
3. Sicherstellung der Konsistenz/ Sicherheit

Wahl des Leader-Knotens Das Raft-Protokoll beschreibt drei verschiedene Rollen, welche ein Knoten des Clusters einnehmen kann: Leader, Follower, und Candidate. Dabei kann ein Cluster immer nur genau einen Leader haben. Wie oben bereits beschrieben, ist der Leader für die Koordination der Log-Replikation und die Kommunikation mit dem Client zuständig. Zusätzlich hat er die Aufgabe, in periodischen Abständen Heartbeats an seine Follower zu versenden, damit diese über die Präsenz eines Leaders im Clusters informiert werden. Das Raft-Protokoll legt fest, dass alle Follower-Knoten eine randomisierte Wartezeit (in einem bestimmten Intervall) besitzen, die zwischen zwei Heartbeats vergehen darf, ohne dass ein Follower den Ausfall des Leaders vermutet. Wird bei einem Follower dieser Heartbeat-Timeout überschritten, übernimmt dieser Knoten die Rolle eines Candidate und initiiert die Wahl eines neuen Leaders. Dabei stimmt er für sich selbst und sendet eine Wahlaufforderung an alle anderen Follower. Zur besseren Organisation des Clusters ist die Regentschaft jeden Leaders ein einen Term geknüpft. Das bedeutet, dass ein Term endet und ein neuer Term beginnt, sobald ein neuer Leader gewählt wird. Um Leader zu werden, muss ein Candidate die Mehrheit aller Stimmen der Follower in seinem Cluster erhalten. In einem Cluster bestehend aus 5 Knoten müsste ein Candidate also beispielsweise Stimmen von zwei weiteren Knoten erhalten, um 3/5 Stimmen zu erreichen und somit zum neuen Leader zu werden. Hier wird nun auch klar, wieso die randomisierten Timeouts wichtig sind. Hätten alle Follower das gleiche Timeout, würden alle zur gleichen Zeit den Ausfall des Leaders bemerken und sich selbst zur Wahl stellen. Somit würden alle Knoten für sich selbst stimmen und es könnte nie zur Erzielung einer Mehrheit kommen.

Ohne zu viel vorwegzugreifen habe ich bei der Implementierung feststellen müssen, dass trotz randomisierter Timeouts in manchen Fällen durch „unglückliche" Verteilung der Timeouts Szenarien entstehen können, in denen mehrere Knoten nahezu zeitgleich den Ausfall des Leaders feststellen und sich als Candidate aufstellen. In diesen Fällen (mehrere Candidates) definiert Raft klare Regeln, nach denen Follower sich verhalten, die Wahlanfragen von mehreren Candidates erhalten:

1. Follower wählen nur Candidates, deren Term mindestens so hoch ist wie der eigene Term
2. Gilt die erste Regel und es gibt mehrere Candidates, auf die das erste Kriterium zutrifft, wird geprüft, ob der Log des Candidates up-to-date mit dem eigenen Log ist (mehr dazu im Log-Abschnitt)
3. Gelten beide Regeln für mehrere Candidates, stimmt der Knoten für den Candidate, dessen Wahlaufforderung zuerst eingetroffen ist.

Trotz dieser Regeln kann es manchmal zu einem sogenannten „Split-Vote“ Szenario kommen, in dem für den jeweiligen Term keine Mehrheit erzielt werden kann. Je nach betrachteter Raft-Implementierung gibt es hier dem Anschein nach mehreren Möglichkeiten, wie eine solche Situation gehandhabt werden kann. Allgemein jedoch endet der betroffene Term im Falle eines Split-Votes vorzeitig, und eine neue Wahl wird angestoßen. Wie dieses Problem in meiner Implementierung gelöst wurde, wird an späterer Stelle dieses Berichts behandelt.

Generell wird jegliche Kommunikation der Knoten untereinander über sogenannte „Remote-Procedure-Calls“ (RPCs) abgewickelt. Im Falle des Wahlprozesses wird beispielsweise ein RequestVote-RPC im Cluster gebroadcastet. Im Falle eines Heartbeats wird ein „leerer“ AppendEntries-RPC verwendet (leer = keine Log-Einträge, die übertragen werden).

Replikation des Logs Ist ein Leader gewählt, ist dieser für die Verwaltung und Replikation des Logs zuständig. Ein Log ist eine simple Datenstruktur (welche jeder Knoten verfügt), die aus einer Liste von Log-Einträgen besteht. Dabei speichert ein Log-Eintrag einen Index, den Term zu dem der Eintrag erstellt wurde, und die eigentliche Instruktion/Operation, die durchgeführt werden soll. Zusätzlich speichert der Log einen Commit-Index, welcher den höchsten Index des Logs angibt, bis zu dem sich die Mehrheit der Knoten über den Zustand des Logs einig ist. Es wird also klar, dass die Log-Replikation auf einem „Append and Commit“ Prinzip aufbaut, was sich dadurch auszeichnet, dass jegliche Log-Einträge erst finalisiert werden und somit auf die State-Machine angewandt werden, sobald diese von der Mehrheit bestätigt wurden.

Der konkrete Ablauf der Log-Replikation kann wie folgt beschreiben werden: Der Client übergibt dem Leader eine Instruktion, wie etwa „$SET\ a\ =\ 14$“. Der Leader erzeugt daraus lokal einen Log-Eintrag, und „appended“ diesen zu seinem eigenen Log. Daraufhin versendet der Leader AppendEntries-RPCs an seine Follower, die den jeweiligen Log-Eintrag enthalten. Bei Erhalt des RPCs durch einen Follower, fügt dieser den neuen Eintrag seinem eigenen Log hinzu und sendet eine Bestätigung (AppendEntriesResponse RPC) zurück an den Leader. Durch diese Antwort kann der Leader nachverfolgen, wie viele Follower den neuen Eintrag erfolgreich zu ihrem Log hinzugefügt haben. Stellt der Leader fest, dass die Mehrheit der Follower den Log-Eintrag repliziert hat, wird dieser Eintrag als committed betrachtet. Der Leader inkrementiert seinen Commit-Index, und propagiert dies an seine Follower, die daraufhin auch ihren Commit-Index updaten. Ein Commit sorgt zusätzlich dafür, dass die Operation nun auf die State-Machine des jeweiligen Knotens angewandt wird. An dieser Stelle existieren einige Szenarien, in denen Log-Inkonsistenzen auftreten können, welche dazu führen, dass ein Append/Commit durch einen Follower abgelehnt wird. Diese werden im nächsten Abschnitt genauer beschrieben. Gilt ein Log-Eintrag als committed kann garantiert werden, dass bei gleichem Index und Term auch die gleiche Operation in allen Logs gespeichert ist. Zusätzlich sind auch alle Log-Einträge mit Index < Commit/Index identisch zueinander.

Sicherstellung der Konsistenz/ Sicherheit Um Konsistenz und Fehlertoleranz im System zu gewährleisten, bietet Raft einige Mechanismen, die teils auch in die Wahl der Leader verstrickt sind. Generell verfolgt Raft das Prinzip, dass der Zustand des Logs des Leaders immer dem Zustand des Logs aller anderen Follower vorzuziehen ist. Dies bedeutet, dass bei Inkonsistenzen typischerweise der Log des Leaders verwendet wird, um den inkonsistenten Log eines Followers zu ergänzen beziehungsweise zu überschreiben. Ein Kernmechansimus zum Erhalt der Konsistenz ist beispielsweise die Speicherung des Terms in den Log-Einträgen. So kann sichergestellt werden, das Knoten sich ihres potenziell veralteten Zustands bewusst sind, da die Hearbeats des Leaders immer den aktuellen Term beinhalten. Auch wird dies klar bei der Wahl des Leaders, bei der Follower immer nur für Candidates stimmen, deren Term mindestens so hoch wie ihr eigener ist.

Sollte ein Follower temporär ausfallen, und somit einige Append- und Commit-Zyklen verpassen, stellt der Follower beim nächsten AppendEntries-RPC fest, dass sein Log-Index (und eventuell auch Term) nicht mit dem des Leaders übereinstimmt. Durch eine negative Antwort des Followers auf den RPC des Leaders erkennt der Leader, dass Handlungsbedarf besteht. Typischerweise wird nun durch eine „matchIndex“ Variable ermittelt, ab welchem Punkt die Inkonsistenzen auftreten. Ist dieser Punkt ermittelt, sendet der Leader von diesem Index and mittels weiterer AppendEntries-RPCs die korrekten Log-Einträge an den Follower, sodass dieser wieder auf dem neusten Stand ist.

Diese Beschreibung der Funktionsweise von Raft sollte grob die grundlegenden Mechanismen des Protokolls beschreiben. Auf Grund von Platzgründen (Bericht max. 5-7 Seiten) habe ich mich an manchen Stellen dafür entschieden, nicht zu sehr ins Detail zu gehen. Zusätzlich zu den oben beschriebenen Features bieten weiterführende Versionen/Implementierungen von Raft auch Features, wie „Log-Compaction“ zur Bekämpfung von immer weiter anwachsenden Logs oder „Snapshotting“ zur Persistenz des Cluster-Zustands. Diese Aspekte wurden jedoch sowohl hier als auch bei meiner Implementierung nicht berücksichtigt.

Implementierung und Simulation

Ich habe mich dazu entschieden, bei der Implementierung einen simulativen Ansatz zu verfolgen, welcher die grundlegenden Funktionsweisen anhand eines Beispiel-Clusters aufzeigt. Das Projekt wurde „von Scratch“ in Java 21 und dem Build-Tool Maven entwickelt (für JUnit Tests). Die Simulation der Kommunikation der Knoten untereinander erfolgt in-memory. Dies bedeutet, dass die Knoten-Objekte untereinander ihre eigenen Methoden aufrufen, und somit keine Netzwerklatenz o.Ä. auftritt. Im Folgenden wird zunächst kurz das Implementierungsdesign vorgestellt. Anschließend werden Ausschnitte eines Log-Outputs präsentiert, der durch Ausführung der Simulation eines Clusters mit 5 Knoten zustande gekommen ist und die Funktionsweise und bestehende Probleme verdeutlichen soll. Genauere Paramater zur Durchführung der Simulation sind der untenstehenden Tabelle zu entnehmen.

Anzahl der Knoten	5
minElectionTimeout	1s
maxElectionTimeout	10s
heartbeatInterval	50ms

Für die Simulation selbst wurden 4 Test-Cases in der Klasse *RaftClusterTest* entwickelt, welche häufig auftretende Szenarien bei der Anwendung des Protokolls abdecken sollen:

1. ***testLeaderElection***
   Da ein Cluster zu beginn als eine Menge von Follower Knoten (ohne Leader) initialisiert wird, überprüft dieser Testfall, ob der Mechanismus der Leader-Wahl ordnungsgemäß funktioniert.

2. ***testLogReplicationAndCommit***
   Zusätzlich zu der Wahl eines Leaders wird hier ein Command an das Cluster übergeben und überprüft, ob der „Append and Commit“ Mechanismus des Protokolls funktioniert.

3. ***testMutlipleCommandsReplicationAndCommit***
   Ähnlich wie oben, nur dieses Mal mit 3 aufeinanderfolgenden Submits.

4. ***testLeaderFailover***
   Simuliert den Ausfall des Leaders, wodurch ein neuer Leader gewählt werden sollte. Zudem wird getestet, ob auch nach dem Ausfall die Log-Replikation noch ordnungsgemäß funktioniert.

Um die Funktionsweise der konkreten Implementierung zu veranschaulichen, wird hier nun der erzeugte Log des zweiten Test-Cases beschrieben. Zu Beginn werden 5 Knoten und die dazugehörigen zufälligen Timeouts (zwischen 1 und 10 Sekunden) erzeugt:

```
2024-09-01 13:41:34 [INFO] RaftNode - node_1 initialized as FOLLOWER with election timeout 1.901 ms
2024-09-01 13:41:34 [INFO] RaftNode - node_2 initialized as FOLLOWER with election timeout 8.793 ms
2024-09-01 13:41:34 [INFO] RaftNode - node_3 initialized as FOLLOWER with election timeout 6.485 ms
2024-09-01 13:41:34 [INFO] RaftNode - node_4 initialized as FOLLOWER with election timeout 4.054 ms
2024-09-01 13:41:34 [INFO] RaftNode - node_5 initialized as FOLLOWER with election timeout 7.468 ms
```

Da zu Beginn noch kein Leader existiert, sollte der Knoten mit dem geringsten timeout (*node_1*) das Ausbleiben des Heartbeats feststellen und die Wahl initiieren. Dementsprechend sollten RPCs an alle anderen Knoten des Clusters gesendet werden und sie zur Wahl auffordern:

```
2024-09-01 13:41:36 [INFO] RaftNode - node_1 did not receive heartbeat in time and is starting election
2024-09-01 13:41:36 [INFO] RaftNode - node_1 became CANDIDATE for term 1 and needs 3 votes to win
2024-09-01 13:41:36 [INFO] RaftNode - node_1 voted for himself. Total votes: 1/3
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending RequestVoteRequest to node_2 for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending RequestVoteRequest to node_3 for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending RequestVoteRequest to node_4 for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending RequestVoteRequest to node_5 for term 1
```

Da initial noch keine Log-Einträge existieren, und dementsprechend auch alle Knoten denselben Term haben (0), folgen die Knoten der Wahlaufforderung und stimmen für *node_1*.

```
2024-09-01 13:41:36 [INFO] RaftNode - node_2 updated term to 1 due to higher term in RequestVoteRequest
2024-09-01 13:41:36 [INFO] RaftNode - node_2 granted vote to candidate node_1 for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_1 received vote response from node_2: true
2024-09-01 13:41:36 [INFO] RaftNode - node_1 received vote from node_2. Total votes: 2/3
2024-09-01 13:41:36 [INFO] RaftNode - node_3 updated term to 1 due to higher term in RequestVoteRequest
2024-09-01 13:41:36 [INFO] RaftNode - node_3 granted vote to candidate node_1 for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_1 received vote response from node_3: true
2024-09-01 13:41:36 [INFO] RaftNode - node_1 received vote from node_3. Total votes: 3/3
```

Sobald die Mehrheit erreicht wurde (hier: 3 Stimmen), erkennen alle Knoten den neuen Leader an und der Leader beginnt Heartbeats zu senden. Das Cluster hat nun einen stabilen Zustand eingenommen, in dem es ohne externe Einflüsse (wie etwa Commands durch den Client oder Ausfall eines Knotens) für immer verweilt. Anmerkung: In dieser Implementierung reichen alle Knoten ihre Stimme ein, auch wenn die Mehrheit bereits erreicht wurde. Dies hat aber keinen Einfluss auf die Funktionalität.

```
2024-09-01 13:41:36 [INFO] RaftNode - node_1 became LEADER for term 1
2024-09-01 13:41:36 [INFO] RaftNode - node_2 recognized new leader: node_1
2024-09-01 13:41:36 [INFO] RaftNode - node_3 recognized new leader: node_1
2024-09-01 13:41:36 [INFO] RaftNode - node_4 recognized new leader: node_1
2024-09-01 13:41:36 [INFO] RaftNode - node_5 recognized new leader: node_1
2024-09-01 13:41:36 [INFO] RaftNode - node_4 updated term to 1 due to higher term in RequestVoteRequest
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending heartbeat for term 1
```

Trifft nun ein Command des Clients an, wird dieser an den Leader geleitet. Der Leader „appended“ seinen Log und kommuniziert den neuen Eintrag zu seinen Followern. Hat die Mehrheit den neuen Eintrag übernommen (2 weitere Knoten, da Leader selbst mitzählt), wird der Eintrag committed. Anmerkung: Auf Grund der Nebenläufigkeit sieht es hier so aus, als würde die Mehrheit erst nach 3 externen Knoten erreicht, dem ist aber nicht so.

```
2024-09-01 13:41:36 [INFO] RaftNode - node_1 appended command to log at index 0: LogEntry{term=1, command='key1=value
2024-09-01 13:41:36 [INFO] RaftNode - node_1 replicating command to followers
2024-09-01 13:41:36 [INFO] RaftNode - node_2 appended new entry at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_3 appended new entry at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_4 appended new entry at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_1 received append responses from the majority of nodes (self included) and
2024-09-01 13:41:36 [INFO] RaftNode - node_1 committed log entry at index 0: LogEntry{term=1, command='key1=value1'}
```

Anschließend wendet der Leader das Command auf seine State-Machine an und propagiert den neuen Commit-Index an seine Follower. Bei Erhalt des neuen Commit-Index wenden auch die Follower die Instruktion auf ihre State-Machine an.

```
2024-09-01 13:41:36 [INFO] RaftNode - node_1 applying log entry to state machine at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_1 propagating commit index 0 to follower node_2
2024-09-01 13:41:36 [INFO] RaftNode - node_2 updated commitIndex to 0
2024-09-01 13:41:36 [INFO] RaftNode - node_2 applying log entry to state machine at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_1 propagating commit index 0 to follower node_3
2024-09-01 13:41:36 [INFO] RaftNode - node_3 updated commitIndex to 0
2024-09-01 13:41:36 [INFO] RaftNode - node_3 applying log entry to state machine at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_1 propagating commit index 0 to follower node_4
2024-09-01 13:41:36 [INFO] RaftNode - node_4 updated commitIndex to 0
2024-09-01 13:41:36 [INFO] RaftNode - node_4 applying log entry to state machine at index 0: LogEntry{term=1, command='key1=value1'}
```

An dieser Stelle tritt nun eine Anomalie auf, die bei meiner konkreten Implementierung zu einer leichten Abweichung des Protokolls zum erwarteten Verhalten führt. Es scheint so, als würde der neue Commit-Index propagiert werden, bevor alle Knoten des Clusters den initialen AppendEntries RPC erhalten haben. Dieses Problem kann dadurch auftreten, dass mit der Propagation des Commit-Index begonnen wird, sobald die Mehrheit der Knoten den Erhalt des RPCs bestätigt hat. In diesem konkreten Fall scheint es so, als ob es bei *node_5* also zu einem Konflikt durch Inkonsistenzen des Logs kommt. In diesem Fall greift die „Catch-Up“ Mechanik, welche im Fall von hinterherhängenden Knoten die fehlenden Einträge liefert. Zwar ist dies nicht das gewünschte Verhalten, aber es führt dennoch zu einem letztendlich konsistenten Zustand des Systems.

```
2024-09-01 13:41:36 [INFO] RaftNode - node_1 not propagating commit index to node_5 as its missing entries
2024-09-01 13:41:36 [INFO] RaftNode - node_1 sending missing entries to node_5
2024-09-01 13:41:36 [INFO] RaftNode - node_5 appended new entry at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_5 applying log entry to state machine at index 0: LogEntry{term=1, command='key1=value1'}
2024-09-01 13:41:36 [INFO] RaftNode - node_5 updated commitIndex to 0
```

Die Versuchsparameter der Simulation wurden so gewählt, sodass ein Split-Vote Szenario möglichst unwahrscheinlich wird. Leider haben meine Versuche, ein Split-Vote Szenario programmatisch zu lösen, nicht zu dem gewünschten Ergebnis geführt. Sowohl Versuche eine maximale Begrenzung für die Wahldauer festzulegen als auch andere Mechanismen (Pre-Voting, Election-Backoff) haben zu ungewünschten Seiteneffekten/Bugs bedingt durch die Nebenläufigkeit geführt. Falls also bei Ausführung der Testfälle einer der Tests fehschlagen sollte (durch Split-Vote), bitte einfach den Test erneut anstoßen. Das Problem sollte sich dann durch das erneute Erzeugen von randomisierten Timeouts erledigen.