Übung 3

Repository: `https://github.com/Raphael90K/DSA3`

1 Aufgabe 1

Paxos und Raft sind Konsensalgorithmen, die in einem verteilten System dazu dienen, eine Einigung über einen Datenwert zu erzielen. Raft nutzt hierzu einen Leader-Based-Ansatz. Der gewählte Anführer ist hierbei für die Verteilung der Daten zuständig. Bei Paxos hingegen sind die Knoten gleichberechtigt und wirken zu gleichen Teilen an der Einigung über das Datum mit.[Holt23]
Aufgrund des liberalen Ansatzes von Paxos wird dieser Algorithmus als Grundlage für die Übung verwendet. Der Paxos Algorthmus wurde 1998 von Leslie Lamport vorgestellt [Lamp98]. Der Paxos Algorithmus stellt einen Konsens über einen Wert sicher, auch wenn das Netzwerk Probleme hat oder einzelne Knoten ausfallen, solange der Großteil des Netzwerks noch funktionsfähig ist.[ompr24]
In Paxos werden für die Funktionsweise folgende Annahmen getroffen[ompr24]:

1. Knoten können **unzuverlässig** sein und daher abstürzen oder ausfallen.
2. Die Knoten haben Zugang zu einem **nichtflüchtigen Speicher**, aus dem der Systemzustand nach einem Absturz wieder hergestellt werden kann.
3. Es gibt **keine Byzantinischen Fehler**. Mit diesen kommt der Paxos Algorithmus nicht zurecht.

Um den Konsens zu erreichen, werden im Paxos Algorthmus von den beteiligten Knoten mehrere Rollen angenommen. Hierbei kann ein Knoten auch mehrere, oder alle Rollen einnehmen [Torr18]:

1. **Proposer**: Der Proposer möchte Einigkeit über einen bestimmten Wert erreichen.
2. **Acceptors**: Die Acceptors stimmen über den vorgeschlagenen Wert ab. Wenn die Mehrheit für den Wert abstimmt, wird der Wert angenommen,
3. **Learners**: Die Learner lernen den angenommenen Wert.

Damit die beteiligten Knoten nun einen Konsens im Netzwerk erreichen können sind mehrere Schritte notwendig, die nacheinander abgearbeitet werden [ompr24, Torr18]:

1. **Prepare Phase**: Der Proposer initiiert den Prozess. Er sendet eine Prepare Nachricht mit einer einzigartigen ID an alle Acceptors.

2. **Promise Phase**: Die Acceptors vergleichen die ID mit eventuell bereits verarbeiteten IDs. Wenn es die erste oder größte ID ist, versendet der Acceptor ein Promise and den Proposer mit dem Versprechen, kleinere IDs zu ignorieren.

3. **Propose Phase**: Wenn der Proposer von der Mehrzahl an Acceptors ein Promise erhält, sendet er den vorgeschlagenen Wert mit der ID in einer Accept-Request bzw. Propose Nachricht an alle Acceptors.

4. **Accept Phase:** Die Acceptors prüfen den Wert und antworten mit einer Accept Nachricht, mit der sie signalisieren, dass sie den vorgeschlagenen Wert akzeptieren. Wenn wieder eine Mehrzahl an Knoten mit einer Accept Nachricht geantwortet haben, wird der vorgeschlagene Wert angenommen.

5. **Learn Phase**: Der Proposer sendet eine Learn Nachricht an die Learner mit dem akzeptierten Wert.

2 Aufgabe 2

2.1 Konzept und Implementierung

Nach Festlegung des Konsensalgorithmus wird als Beispielanwendung eine einfache, verteilte Bank-Anwendung als Simulation in Java implementiert und im Simulator `sim4da`[1] ausgeführt. Ziel ist es, in einer einfachen Anwendung Ein- und Auszahlungen vorzunehmen, wobei der Kontostand in mehreren Replikaten dezentral verwaltet wird. Hierbei soll die Anwendung eine hohe Konsistenz und Verfügbarkeit gewährleisten. Ein Kunde soll ein und Auszahlungen vornehmen. Hierfür

[1] `https://github.com/oxoo2a/sim4da-v2`

stehen ihm mehrere Bank-Knoten zur Verfügung. Die Bank-Knoten einigen sich auf den neuen Kontostand und melden dem Kunden den Erfolg oder Misserfolg des Geldtransfers. Hierdurch soll ein Bankennetzwerk mit verschiedenen Standorten simuliert werden, wobei der Kunde an jedem Standort Zugriff auf sein Konto hat, die Knoten ohne zentrale Instanz einen Konsens über den Kontostand erreichen und dann einen Geldtransfer vornehmen.
Hierzu werden im Kern folgende Komponenten erstellt:

1. Client

Die `Client` Klasse repräsentiert den Kunden. In der `engage` Methode wählt der Kunde eine Kontoänderung im Bereich von -100 und 100 und erstellt eine Nachricht mit dem typ `CHANGE`. Diese Nachricht sendet er an einen zufälligen Bankknoten des Netzwerks und wartet, ob die Ein- oder Auszahlung erfolgreich war oder nicht.

2. BankNode

Die `BankNode`-Klasse befindet sich zusammen mit den zugehörigen weiteren Klassen im Package `BankNode`. Objekte dieser Klasse repräsentieren die dezentralen Knoten des Banking Systems und besitzen für die Funktion hauptsächlich folgende Eigenschaften:

1. **bank**: Ein Objekt der Klasse `Bank`. Die Klasse befindet sich im Package `Bank`. In der Klasse werden Kontoinhaber, Kontostand sowie die Transaktionen als Liste von Objekten der Klasse `Transaction` gespeichert. Dies stellt den Log der Ein- und Auszahlungen dar, auf die sich die Knoten geeinigt haben und wird in der Folge auch als Bank-Log bezeichnet. Zur Serialisierung als JSON String und zur Sicherung auf der Festplatte wird das Framework `Jackson`[2] verwendet.

2. **paxos**: Ein Objekt der Klasse `Paxos`. Über diese Klasse werden die Funktionen des Paxos Algorithmus implementiert.

In der Methode `engage` werden Nachrichten entgegengenommen, solange die Simulation läuft. Diese Nachrichten werden in der `handle` Methode nach typen getrennt verarbeitet. Um die Typen zu

[2] `https://github.com/FasterXML/jackson`

unterscheiden wird die enumeration `Command` verwendet.
Sobald der Client eine Transaktion bei einem Knoten initiiert werden folgende Schritte durchgeführt:

1. Der Knoten fungiert als Proposer und beginnt mit der Initialisierung, loggt die angefragte Ein- oder Auszahlung und berechnet eine neue ID für die Konsensfindung. Die Berechnung der ID wird mit der Formel $ID = (\frac{\text{highestProposalNr}}{\text{networkSize}} + 1) * \text{networkSize} + \text{nodeId}$ berechnet. So wird gewährleistet, dass jeder Knoten auf eindeutige IDs zugreift.

2. Der Proposer versendet eine PREPARE Nachricht mit der berechneten ID an alle Knoten des Netzwerks, die als Acceptors arbeiten.

3. Alle Acceptors prüfen nun, dass die ID größer als alle zuvor verwendeten IDs ist, speichern diese ID ab und antworten mit einer PROMISE Nachricht. Falls sie bereits eine höhere ID kennen, wird keine Nachricht versendet.

4. Der Proposer startet einen gesonderten Thread, der die ankommenden Nachrichten verarbeitet und wartet dann ein zuvor festgelegtes Timeout. Nach dem Timeout prüft der Proposer ob er von mehr als der Hälfte der Knoten im Netzwerk (sich selbst eingeschlossen) ein PROMISE erhalten hat.

5. Nun sendet der Proposer die vorgeschlagene Transaktion mit einer PROPOSE Nachricht an alle Acceptors.

6. Diese prüfen nun, ob die ID der PROPOSE Nachricht mit der vorherigen übereinstimmt und ob die Transaktion valide ist. Wenn die Bedingungen erfüllt sind, antworten sie mit einer ACCEPT Nachricht.

7. Der Proposer zählt nun erneut in einem gesonderten Thread mit einem vorgegebenen Timeout, ob die Mehrheit der Knoten für die Transaktion stimmen.

8. Sollte die Transaktion erfolgreich sein, fügt der Proposer die Transaktion in die Liste ein und speichert sie auf der Festplatte ab. So ist gewährleistet, dass die Transaktionen auch

nach Absturz der Anwendung oder Hardware wieder geladen werden kann. Nun sendet er die Transaktion mit einer LEARN Nachricht an alle Knoten, die die Transaktion ebenfalls auf der Festplatte abspeichern.

9. Zuletzt informiert der Proposer den Client über den Erfolg der Transaktion. Wenn die Transaktion zu einem früheren Zeitpunkt fehlschlug, wurde zuvor ein Misserfolg an den Client übermittelt.

Die Reihenfolge der versendeten Nachrichten zur Konsensfindung wird in Abbildung 1 dargestellt. Die Entwicklung der Anwendung erfolgte mit Hilfe von ChatGPT[3] in der kostenlosen Version, wobei die Anwendung zur Recherche nach Funktionen / Methoden verwendet sowie zu Code-Feedback durch Einfügen eigenem Codes verwendet. Eine komplexe und funktionierende Paxos Implementierung, oder Teile davon, konnte ChatGPT nicht erstellen.

2.2 Simulation

Nach der Implementierung wurde eine Simulation mit 5 `BankNodes` durchgeführt und für 30 Sekunden simuliert. Die gespeicherten Bank-Logs und der Simulations-Log befinden sich in Ordner `SIM1`. Der aufgezeichnete Simulations-Log zeigt, dass es zu keinen Fehlschlägen bei der Simulation kam und die aufgezeichneten Bank-Log für alle Knoten gleich sind.
Im Anschluss wurde eine zweite Simulation ausgeführt. Für diese Simulation wurden die Logs von `Node3` und `Node4` weiterverwendet und die Logs von `Node0 - Node2` gelöscht. Durch das automatische Laden eines bestehenden Bank-Logs bei Start eines Knotens soll gewährleistet werden, dass ein Knoten nach Absturz weiterarbeiten kann. Durch diese Startbedingungen werden zwei Logs geladen und drei neu erstellt. Hierdurch sollte es lediglich zu einem Konsens kommen, wenn die Knoten `Node0 - Node2` eine Transaktion initialisieren. Die Ergebnisse der Simulation befinden sich in Ordner `SIM2`. Die Logs zeigen das erwartete Ergebnis. Bei Vergleich der Bank-Logs kann festgestellt werden, dass die Logs von der Knoten `Node0 - Node2` gleich sind und die übrigen Logs unverändert blieben, da es in den Fällen zu keinem Konsens kam.
In einem dritten Simulationslauf wurden die Logs aus `SIM2` wieder verwendet und der Bank-Log

[3] `https://chatgpt.com/`

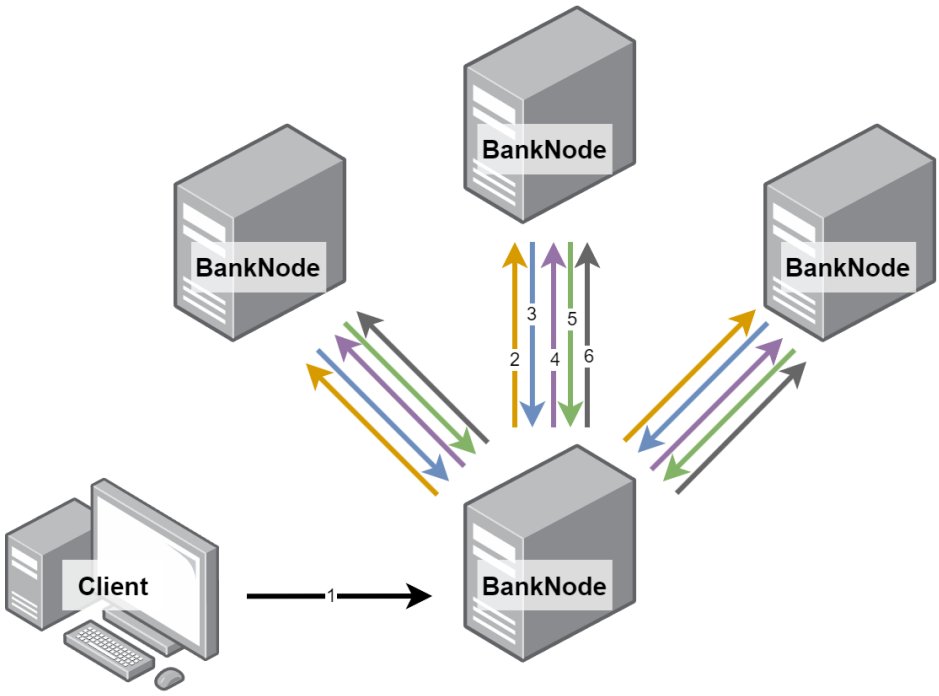

Abbildung 1: Paxos Implementierung als Banking Anwendung mit den Nachrichten (1) CHANGE, (2) PREPARE, (3) PROMISE, (4) PROPOSE, (5) ACCEPT, (6) LEARN, Quelle: Eigene Darstellung

von `Node0` gelöscht. Somit hat ein Knoten keinen initialen Log und von den anderen vier Knoten besitzen jeweils zwei die gleichen Logs. Erwartungsgemäß wird für keine Transaktion ein Konsens gefunden, da nicht die Mehrheit aufgrund des vorhanden Logs eine Transaktion verifizieren und eine ACCEPT Nachricht versenden kann.

2.3 Änderungen an sim4da

Eine Änderung wurde am Simulator vorgenommen. In der Methode `reveice` der Klasse `NodeProxy` wurde der try / catch Block entfernt und die Exceptpion an die aufrufende Methode weitergereicht. Dies war notwendig, um den Thread der zum Empfang eingehenden PROMISE und ACCEPT Nachrichten in der Methode `countMessages` in `BankNodes` sicher beenden zu können. Ohne diese Änderung war dies so nicht möglich.

3 Fazit

Die Ergebnisse der Simulation zeigen, dass die implementierte simple Bankanwendung funktioniert und durch den Paxos Algorithmus ein Konsens über die Transaktionen erzielt werden kann. Die zweite durchgeführte Simulation zeigt, dass es möglich ist zu einem Konsens zu kommen, wenn mehr als die Hälfte der Knoten für eine Transaktion stimmen.
Durch die Implementierung kann auch nur zu jeweils einer Transaktion zur gleichen Zeit ein Konsens gefunden werden. Sollte eine neue Transaktion mit einer höheren ID mit einer PREPARE Nachricht eingehen, wird die vorherige Abstimmung in der Folge ignoriert. Zudem ignoriert der Proposer während der Prüfung der zurückkehrenden Nachrichten in der Methode `countMessages` Nachrichten des falschen Typs, die in der Zwischenzeit eingehen. Durch diese Implementierung ist die Konsistens des Gesamtsystems gewährleistet, auch wenn einzelne Knoten einen falschen oder aufgrund eines zeitweise Systemausfalls einen unvollständigen Bank-Log besitzen.
Aufgrund der Ergebnisse aus der Simulation in Abschnitt 2.2 wird davon ausgegangen, dass die Knoten mit einem Systemausfall oder bei Verlust von Nachrichten weiter arbeiten können und das Netzwerk in seiner Gesamtheit grundsätzlich funktionsfähig bleibt. Da jedoch bisher keine Methoden für einen Reset oder eine entsprechende Simulation implementiert wurde, wäre dies eine künftige Verbesserung des Systems. Zudem zeigen auch die Tests von Simulation 2, dass das Gesamtnetz noch konsensfähig ist, auch wenn zwei Knoten einen falschen Log besitzen. Im nächsten Schritt müssten Methoden implementiert werden, wobei falsche Logs oder Logs mit fehlenden Einträgen wieder auf den aktuellen Stand gebracht werden können.

Literatur

[Holt23] D. Holt. Comparing RAFT, PAXOS, and Calvin Consensus Algorithms: Why CockroachDB Chose RAFT. https://dantheengineer.com/raft-paxos-and-calvin/, 2023. Stand: 13.03.2023, Aufruf am: 20.09.2024.

[Lamp98] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.* 16(2), 1998, S. 133–169.

[ompr24] omprakashkumar7079. PAXOS Consensus Algorithm. https://www.geeksforgeeks.org/paxos-consensus-algorithm/, 2024. Stand: 21.06.2024, Aufruf am: 20.09.2024.

[Torr18] L. Q. Torres. The Paxos Algorithm. https://www.youtube.com/watch?v=d7nAGI_NZPkab_channel=GoogleTechTalks, 2018. Stand: 02.02.2018, Aufruf am: 20.09.2024.