FUCoin: Design und Implementierung eines Bitcoin-Systems

Praxisseminar Verteilte Systeme SS2015
Larissa Zech(4594149) und Benjamin Zengin(4553500)

I. Einleitung

Im Rahmen des Praxisseminars Verteilte Systeme im Sommersemester 2015 haben wir uns intensiv mit der Entwicklung von *FUCoin*, einem an diese Lehrveranstaltung angepassten Bitcoin-System, befasst. Ein Bitcoin ist eine digitale Geldeinheit, die in einem weltweit verfügbaren Peer-to-Peer-Netzwerk gehandelt wird. Jeder Teilnehmer am Bitcoin-System benötigt die Bitcoin Core-Software und erhält digitale Brieftaschen (*Wallets*) zur Speicherung seines Guthabens.

Die gesamte Transaktions-Historie des Netzwerks wird in einer dezentralen Datenbank in Form einer Block-Chain gespeichert. Jeder Block umfasst eine oder mehrere Transaktionen, die mit einer Prüfsumme versehen sind, und wird in einem rechenintensiven kryptographischen Prozess, dem sogenannten *Mining*, erzeugt. Nach der Erzeugung eines neuen Blocks wird dieser im Netzwerk verbreitet. Hierbei kann es zu einer Verzweigung der Block-Chain kommen, wenn mehrere Bitcoin Cores gleichzeitig neue gültige Blöcke erzeugen. Da angenommen wird, dass hinter der längeren Block-Chain auch die Mehrheit der Teilnehmer steht, setzt sich in diesem Fall der Zweig mit der längeren Block-Chain durch.

Jede Transaktion an die dem Zahlungssender bekannte Bitcoin-Adresse des Zahlungsempfängers wird mit dem privaten Schlüssel des Zahlungssenders signiert und anschließend durch Flooding im Netzwerk verbreitet. Empfängt ein Teilnehmer diese Transaktion, prüft er sie durch Verifikation der Signatur auf Gültigkeit, übernimmt sie in die noch offenen Transaktionen und sendet sie weiter an die ihm bekannten Bitcoin Cores. Jede Transaktion ist mit einer Gebühr von mindestens 0,00001BTC belegt. Die Transaktionsgebühren werden dem Teilnehmer gutgeschrieben, der diese Transaktion bei der Erzeugung eines neuen gültigen Blocks berücksichtigt. Daher kann die Gebühr vom Zahlungssender erhöht werden, um eine höhere Priorisierung seiner Transaktion zu erzielen.

Wenn ein Teilnehmer einen neuen Block für die Block-Chain berechnet, so erhält er eine Belohnung für das Schöpfen des Blocks, sowie die Gebühren aller im Block enthaltenen Transaktionen. Folglich werden Transaktionen mit höheren Gebühren beim Mining-Prozess bevorzugt. Die Belohnung, die für die Berechnung eines neuen Blockes ausgeschüttet wird, wird alle vier Jahre halbiert. Aktuell beträgt sie 25 Bitcoins.

Das Mining funktioniert so, dass der Teilnehmer vor den zu berechnenden Block Zufallszahlen setzt und anschließend den kryptografischen Hashwert dieses Blocks berechnet. Dies wird solange wiederholt, bis der Hashwert kleiner als der vom Netz vorgegebene Schwellwert (Schwierigkeitsgrad) ist. Der Schwierigkeitsgrad des Netzes wird höher, je mehr Teilnehmer sich am Mining beteiligen, sodass ungefähr alle zehn Minuten ein neuer Block veröffentlicht wird.

Das Ziel dieses Praxisseminars war die Entwicklung und Implementierung eines stark vereinfachten Bitcoin-Systems. Zu Beginn der Lehrveranstaltung wurde die Verwendung des Akka-Frameworks beschlossen, mit dem in Skala, Java und C# implementiert werden kann und das die Entwicklung verteilter Anwendungen vereinfacht.

II. Unser FUCoin-System

Im Unterschied zur quelloffenen, weltweit genutzten Bitcoin-Software verzichten wir in unserer Implementierung auf eine Block-Chain und aufwändige kryptographische Berechnungen und setzen den Schwerpunkt auf die Implementierung verschiedener verteilter Algorithmen.

Wir gehen in unserer Implementierung davon aus, dass jeder Teilnehmer des Netzwerks über nur eine Wallet verfügt und diese für die gesamte Dauer der Teilnahme behält. Als Eintritt in das Netzwerk (*Join*) dienen entweder der optionale Statistikserver, dessen Adresse dem Teilnehmer bekannt sein muss, oder, im Falle des Verzichts auf einen Statistikserver, ein dem Teilnehmer bekannter anderer Teilnehmer am Netzwerk. Der Statistikserver speichert Namen, Adresse und initialen Kontostand des eintretenden Wallets und antwortet mit einer Liste von Nachbarknoten. Erfolgt der Eintritt in das Netzwerk durch einen gleichberechtigten Teilnehmer, so antwortet auch dieser mit einer Liste von Nachbarknoten.

Die weitere Kommunikation erfolgt dann hauptsächlich zwischen aktiven Wallets. Jedes Wallet verfügt über eine Liste bekannter Nachbarn und speichert zudem die Zustände einiger synchronisierender Wallets. Der Benutzer hat die Möglichkeit, Transaktionen über die Konsole anzustoßen. Der Ablauf einer Transaktion ist in den Abschnitten III-B

Transaktionen zu unbekannten Wallets und III-C 2-Phase Distributed Commit beschrieben.

Auch der Austritt aus dem Netzwerk kann durch eine Konsoleneingabe geschehen. Findet ein kontrolliertes Verlassen des Netzwerkes (*Leave*) statt, werden sowohl die Zustände aller synchronisierten Wallets als auch der eigene Zustand an alle bekannten Nachbarn verbreitet. Diese aktualisieren dann die von ihnen gespeicherten Zustände entsprechend. Auf das Vorgehen bei Wiedereintritt einer bereits bekannten Wallet wird im Abschnitt III-A Rejoin eingegangen.

Der Statistikserver ist ein optionaler zusätzlicher Teilnehmer im Netzwerk, dessen Aufgabe es ist, Daten über den Zustand des Netzwerkes zu sammeln. Hierzu werden die Namen, Adressen und Kontostände aller Wallets im Netzwerk gespeichert. Wenn eine Wallet eine Änderung ihres eigenen Kontostandes oder des Kontostandes einer synchronisierten Wallet vornimmt, wird eine entsprechende Nachricht an den Statistikserver gesendet. Dieser nimmt eine Aktualisierung der gespeicherten Kontostände vor und prüft den globalen Zustand des Netzwerks auf Konsistenz, indem die Summe aller Kontostände mit der initialen Geldmenge verglichen wird. Im Abschnitt V-B Snapshot wird dargestellt, wie mit Hilfe eines Snapshots des Netzwerks eine glaubwürdigere Aussage über den Zustand des Netzwerks gemacht werden kann. Das Verhalten des Statistikservers haben wir in mehreren Testläufen beobachtet. Die Ergebnisse werden im Abschnitt IV-A Statistikserver diskutiert.

III. Herausforderungen

A. Rejoin

Bei einem Wiedereintritt eines im Netzwerk bereits bekannten Teilnehmers soll zunächst überprüft werden, ob sich der letzte gültige Zustand dieser Wallet noch im Netzwerk befindet. Zu diesem Zweck stößt jeder Teilnehmer, der das Netzwerk betritt und vom Statistikserver eine Liste seiner Nachbarknoten erhalten hat, eine Suche nach seiner Wallet an.

Um eine Änderung des zur Verfügung stehenden Interfaces zu vermeiden, wird hierzu zunächst ein zufälliger Nachbarknoten aus der Liste der bekannten Nachbarn ausgewählt und eine *ActionSearchMyWallet*-Nachricht entsprechend der Vorgaben des Interfaces gesendet. Dieser zufällige Nachbar überprüft, ob sich der Name der gesuchten Wallet in der Liste seiner synchronisierten Wallets befindet. Falls ja, sendet er die gesuchte Wallet umgehend an den Initiator der Suche zurück.

Andernfalls erzeugt er eine neue UUID und startet eine *Gossiping*-Suche nach dieser Wallet. Die verwendete Gossiping-Nachricht enthält den Namen der gesuchten Wallet, den Initiator der Suche sowie die UUID des Gossiping-Durchgangs und wird an alle bekannten Nachbarn gesendet.

Erhält ein Teilnehmer eine Gossiping-Nachricht, prüft er die UUID und verwirft die Nachricht umgehend, falls er diese UUID bereits kennt. Andernfalls überprüft er die Liste seiner synchronisierten Wallets und sendet entweder die gesuchte Wallet an den Initiator der Suche oder leitet die Gossiping-Nachricht an alle bekannten Nachbarn weiter.

Somit verbreitet sich die Suche nach der Wallet im gesamten Netzwerk, wobei aber jeder Teilnehmer Nachrichten derselben Gossiping-Suche nur ein Mal weiterleitet. Das reduziert die Anzahl der Nachrichten im Netzwerk erheblich und sichert zudem die Terminierung der Suche. Der Initiator erfährt jedoch nicht, wenn der Algorithmus terminiert und seine Suche somit fehlgeschlagen ist. Darum haben wir einen Timeout gesetzt, nach welchem der Initiator davon ausgeht, dass der Zustand seiner Wallet dem Netzwerk nicht bekannt ist.

Sofern im Netzwerk auch ein Statistikserver vorhanden ist, könnte die Wallet des eintretenden Teilnehmers alternativ direkt vom Statistikserver abgefragt werden.

Im Abschnitt IV-B Rejoin überprüfen wir die korrekte Funktionsweise unseres Rejoin-Verfahrens mit Hilfe einiger Testläufe.

B. Transaktionen zu unbekannten Wallets

Möchte ein Benutzer eine Transaktion durchführen, muss er den Namen des Zahlungsempfänger kennen und angeben. Es ist aber möglich, dass diese Wallet nicht in der Liste der bekannten Wallets enthalten ist. In diesem Fall kann keine direkte Transaktion zum Zahlungsempfänger vorgenommen werden.

Um eine einheitliche Vorgehensweise anwenden zu können, die unabhängig ist von der Bekanntheit des Zahlungsempfängers, beginnen wir jede Transaktion mit einer *TryTransaction*-Nachricht mit festgelegter *Time To Live (TTL)* an die eigene Wallet. Bei Erhalt einer TryTransaction-Nachricht wird geprüft, ob der in der Nachricht eingetragene Zahlungsempfänger mit dieser Wallet übereinstimmt. Falls ja, wird in Vorbereitung auf den sich anschließenden *Distributed Commit* eine *TransactionCommitInfo*-Nachricht mit Namen und Adressen aller mit dieser Wallet synchronisierten Wallets an den Zahlungssender gesendet. Der von uns implementierte Distributed Commit wird im folgenden Abschnitt III-C 2-Phase Distributed Commit beschrieben.

Stimmt der gesuchte Zahlungsempfänger jedoch nicht mit dieser Wallet überein, wird in der Liste der bekannten Nachbarn nach dem Zahlungsempfänger gesucht. Ist er dort vorhanden, wird die TryTransaction-Nachricht direkt an den Zahlungsempfänger weitergeleitet. Falls

nicht, wird die TryTransaction-Nachricht an einen zufällig ausgewählten Nachbarn weitergeleitet, sofern die TTL noch nicht abgelaufen ist. Wenn die TTL bereits abgelaufen ist, wird eine *TransactionFailed*-Nachricht an den Zahlungssender geschickt, um ihn über das Scheitern seiner Transaktion zu informieren.

Zur Sicherstellung eines konsistenten Zustands der Wallet ist nur eine offene Transaktion zulässig. Erst, wenn diese durchgeführt wurde oder gescheitert ist, hat der Benutzer die Möglichkeit, eine weitere Transaktion anzustoßen.

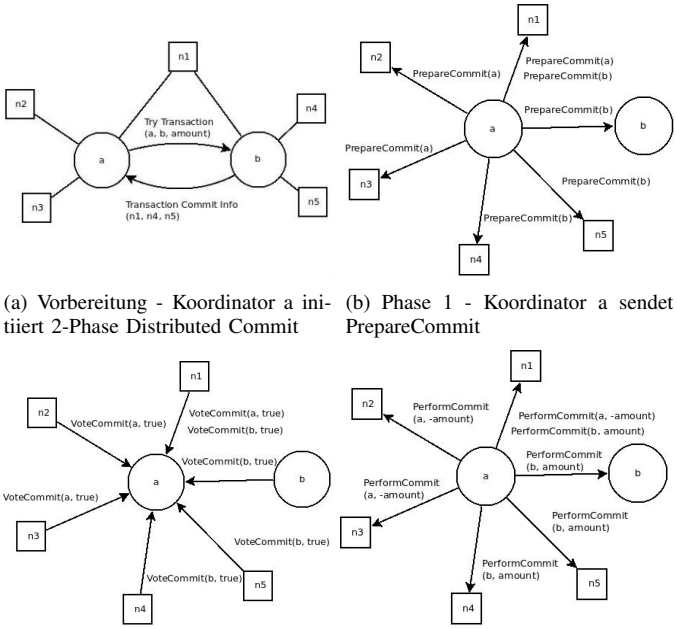
In unserer Implementierung wird also die TryTransaction-Nachricht stets nur an einen zufälligen Nachbarn weitergeleitet. Das Netzwerk wird nicht mit einer Suchanfrage geflutet. Dies kann dazu führen, dass die Suche nach dem Zahlungsempfänger durch eine ungünstige Wahl der zufälligen Nachbarn fehlschlägt, obwohl der Zahlungsempfänger aktiv und im Netzwerk bekannt ist. Im Abschnitt IV-C Transaktionen zu unbekannten Wallets wird näher auf Testläufe und deren Auswertung eingegangen.

Wenn das Netzwerk mit einem zentralen Statistikserver betrieben wird, dann verfügt dieser über eine vollständige Liste aller im Netzwerk bekannten Wallets. In diesem Fall könnte die TryTransaction-Nachricht direkt an den Statistikserver gesendet werden, der diese dann an den Zahlungsempfänger weiterleitet.

C. 2-Phase Distributed Commit

Der Kontostand jeder Wallet wird mit einer Anzahl von Nachbarknoten synchronisiert. Dadurch wird sichergestellt, dass der letzte gültige Zustand der Wallet bei einem Wiedereintritt vom Netzwerk bereitgestellt werden kann. Hierzu muss jedoch der auf verschiedenen Teilnehmern gespeicherte Kontostand konsistent gehalten werden. Eine Möglichkeit, den gegenseitigen Ausschluss in einem Verteilten System sicherzustellen, bieten Distributed Commit-Verfahren. Diese Verfahren erhöhen die Wahrscheinlichkeit, dass die Änderung einer Variablen entweder von allen Teilnehmern vorgenommen wird oder von keinem.

Wir haben uns in unserer Implementierung für den 2-Phase Distributed Commit entschieden. Wie im vorigen Abschnitt beschrieben und in Abbildung 1(a) dargestellt, antwortet der Zahlungsempfänger auf eine TryTransaction-Nachricht mit einer TransactionCommitInfo-Nachricht an den Zahlungssender, die alle mit dieser Wallet synchronisierten Wallets enthält. Der Zahlungssender übernimmt stets die Rolle des Koordinators und unterscheidet zwischen den eigenen synchronisierten Wallets und denen des Zahlungsempfängers. Der Zahlungsempfänger selbst verhält sich wie die mit ihm synchronisierten Wallets.



(a) Vorbereitung - Koordinator a initiiert 2-Phase Distributed Commit

(b) Phase 1 - Koordinator a sendet PrepareCommit

(c) Phase 1 - Teilnehmer antworten mit VoteCommit

(d) Phase 2 - Koordinator a sendet PerformCommit

Fig. 1. 2-Phase Distributed Commit

In der ersten Phase des Protokolls sendet der Koordinator wie in Abbildung 1(b) und 1(c) gezeigt eine *PrepareCommit*-Nachricht mit seinem eigenen Namen an alle mit ihm synchronisierten Teilnehmer und eine PrepareCommit-Nachricht mit dem Namen des Zahlungsempfängers an alle mit dem Zahlungsempfänger synchronisierten Teilnehmer. Die Teilnehmer prüfen daraufhin, ob sie bereits eine offene Commit-Anfrage für diese Wallet gespeichert haben. Wenn nicht, senden sie eine *VoteCommit*-Nachricht mit Wahrheitswert true an den Koordinator und speichern die offene Anfrage für diese Wallet.

In der zweiten Phase des Protokolls sendet der Koordinator entsprechend der Abbildung 1(d) eine PerformCommit-Nachricht mit dem Namen der zu verändernden Wallet und der zu übertragenen Geldmenge an alle Teilnehmer. Bei Erhalt der PerformCommit-Nachricht führen alle Teilnehmer einen lokalen Commit durch, führen also die Transaktion für dieses Wallet lokal aus.

Die Verwendung des 2-Phase Distributed Commit bewirkt eine höhere Sicherheit gegenüber Ausfällen von Teilnehmern und Nachrichtenübertragungsfehlern. Durch die Phase des Global Commits, wird sichergestellt, dass alle Teilnehmer mit dem Commit einverstanden sind und ihn auch nur dann ausführen, wenn dieses Einverständnis von allen bestätigt wurde.

IV. Experimente und Auswertung

A. Statistikserver

Auf ein Experiment wird an dieser Stelle verzichtet, da die Masse an Log-Ausgaben nicht übersichtlich wäre. Stattdessen wird ein Eindruck vermittelt, der beim Testen des Statistikservers entstanden ist.

Da der Test des Statistikservers mit periodischen Zeitintervallen arbeitet, führen die verschiedenen Wallets ihre Aktionen mehr oder weniger simultan aus. Dies führt zu schwer vorhersagbarem Verhalten und vielfältigem Fehlverhalten unserer Implementierung. Das genaue Analysieren der Fehlerquelle ist aufgrund der Parallelität der Ausführung folglich schwierig.

Wirkt dieses Ergebnis auf den ersten Blick wenig aussagekräftig, ist es jedoch genauer betrachtet ein Hinweis auf die fehlende Robustheit des Programms. Die Spezifikation des Protokolls ist leider zu unscharf, um alle Eventualitäten in Betracht zu ziehen. Es müsste also noch ein Handling von Fehlern eingebaut werden, sodass eine Wallet nicht einfach abstürzt, oder in einen Deadlock gerät. Optimal wäre eine Lösung, bei der der letzte funktionierende Zustand wiederhergestellt wird und die Konsistenz dabei gewahrt wird.

B. Rejoin

Experimentaufbau und -ablauf:

1) Wallet A initialisiert das Netz und weist sich selbst einen zufälligen Betrag zu
2) Wallet B tritt dem Netz über A bei und weist sich ebenfalls einen zufälligen Betrag zu
3) B tritt aus dem Netz aus und speichert sein Wallet bei A
4) C tritt über A dem Netz mit einem zufälligen Betrag bei
5) A verlässt das Netz und speichert sein Wallet, sowie das von B, bei C
6) B tritt dem Netz über C bei und erhält den Betrag seines Wallets von C
7) A tritt dem Netz über C bei und erhält den Betrag seines Wallets von C

Ausgabe (Auszug):

1) A: Wallet amount is: 79
2) B: Wallet amount is: 49
   B: Checking endpoint for join.
   B: endpoint is connected, performing Join...
   A: Join Received.
   A: Sending 1 neighbors.
   B: Join successful.
3) B: Left network in safe state.
   A: Received Update for wallet B
4) C: Wallet amount is: 51
   C: Checking endpoint for join.
   C: endpoint is connected, performing Join...
   A: Join Received.
   A: Sending 2 neighbors.
   C: Join successful.
5) A: Left network in safe state.
   C: Received Update for wallet B
   C: Received Update for wallet A
6) B: Checking endpoint for join.
   B: endpoint is connected, performing Join...
   C: Join Received.
   C: Sending 3 neighbors.
   B: Join successful.
   C: Received request to find wallet...
   C: Found it! Sending Back.
   B: Found Wallet! Balance: 49
   C: Received Update for wallet B
7) A: Checking endpoint for join.
   A: endpoint is connected, performing Join...
   C: Join Received.
   C: Sending 3 neighbors.
   A: Join successful.
   C: Received request to find wallet...
   C: Found it! Sending Back.
   A: Found Wallet! Balance: 79
   C: Received Update for wallet A

Die von uns durchgeführten Experimente zum Test der Rejoin-Funktion bestätigen eine korrekte Funktionsweise unserer Implementierung wie oben ersichtlich.

C. Transaktionen zu unbekannten Wallets

Experimentaufbau:

Für dieses Experiment wird die Anzahl der beim Join versendeten Nachbarn auf 1 begrenzt.

Experimentablauf:

1) Wallet A initialisiert das Netz und weist sich selbst einen zufälligen Betrag zu
2) Wallet B tritt dem Netz über A bei und weist sich ebenfalls einen zufälligen Betrag zu (A sendet sich selbst als Nachbarn)
3) Wallet C tritt dem Netz über B bei und weist sich ebenfalls einen zufälligen Betrag zu (B sendet A als Nachbarn)
4) Wallet C überweist Wallet B einen Betrag, obwohl Wallet B nicht in den known Neighbours ist

Ausgabe (Auszug):

1) A: Wallet amount is: 23
2) B: Wallet amount is: 78
   B: Checking endpoint for join.
   B: endpoint is connected, performing Join...
   A: Join Received.
   A: Sending 1 neighbors.
3) C: Wallet amount is: 71
   C: Checking endpoint for join.
   C: endpoint is connected, performing Join...

B: Join Received.
B: Sending 1 neighbors.

4) C: Perform Transaction to B, amount 20
C: Forwarding transaction randomly.
A: Forwarding transaction to recipient.
B: Received transaction request.

Auch hier wird demnach die korrekte Funktionsweise unserer Implementierung durch die von uns durchgeführten Experimente bestätigt. Die Transaktion selbst wird mit Hilfe des 2-Phase Distributed Commit durchgeführt. Ein entsprechendes Experiment wird im folgenden Abschnitt beschrieben.

D. 2-Phase Distributed Commit

Experimentaufbau:
Da sich der 2-Phase Distributed Commit direkt an den Erhalt der Transaction Request anschließt, entspricht der Experimentaufbau demjenigen im vorigen Abschnitt.

Experimentablauf:

1) Wallet C startet eine Überweisung zu Wallet B
2) Wallet C erhält die Commit Infos von B und leitet den Commit ein
3) Alle Teilnehmer voten für die Transaktion
4) Der globale commit wird durchgeführt

Ausgabe (Auszug):

1) B: Received transaction request.
2) C: Ask vote from A for B
C: Ask vote from B for B
C: Ask vote from A for C
C: Received Transaction Commit Info. Waiting for 3 responses.
3) A: Received vote request for B
A: Commit, all right.
A: Received vote request for C
A: Commit, all right.
B: Received vote request for B
B: Commit, all right.
4) C: Received Responses from all participants.
C: Global Commit.
A: Perform commit for B
A: Perform commit for C
B: Perform commit for B
B: Received Transaction. Balance before: 78
B: 20 FuCoins have been transferred to your account.
B: Balance is now: 98

In den von uns durchgeführten Testdurchläufen konnte auch hier kein Fehlverhalten festgestellt werden.

V. Erweiterungsmöglichkeiten

A. Ping

Da wir in unserer Implementierung darauf verzichtet haben, die Aktivität der bekannten Nachbarn durch einen regelmäßigen Ping zu prüfen, werden Wallets, die das Netzwerk verlassen haben, nicht aus der Liste der synchronisierten Wallets entfernt. Dies wird dazu führen, dass inaktive Wallets Transaktionen der mit ihnen synchronisierten Wallets blockieren. Da sie nicht mehr im Netzwerk verfügbar sind, antworten sie auch nicht auf die PrepareCommit-Nachricht des Koordinators und somit wird jede offene Transaktion abgebrochen.

Dieses Problem ließe sich sehr leicht durch zwei Erweiterungen lösen. Einerseits sollten Wallets, die das Netzwerk geplant verlassen, aus den Listen der benachbarten Wallets entfernt werden. Zudem sollten die synchronisierten Wallets durch einen Ping regelmäßig überprüft und gegebenenfalls ebenso aus den Listen entfernt werden.

Die Wahl der Nachbarn sowie der Abstand zwischen den Pings bestimmen, wie sehr das Netzwerk durch diese Nachrichten belastet wird. Möglicherweise wäre es bereits ausreichend, einen Ping an alle Wallets zu senden, von denen der Koordinator bei einer gescheiterten Transaktion keine Antwort erhalten hat.

B. Snapshot

Wie unsere Testdurchgänge in Abschnitt IV-A Statistikserver gezeigt haben, meldet der Statistikserver deutlich häufiger einen inkonsistenten Zustand des Netzwerkes, der dann jedoch wieder zu einem konsistenten Zustand wird.

Wie in Abschnitt II Unser FUCoin-System erläutert, prüft der Statistikserver nach jeder erhaltenen Aktualisierung den globalen Zustand des Netzwerkes und meldet ihn als inkonsistent, wenn die Summe aller derzeitigen Kontostände nicht mit der initialen Geldmenge übereinstimmt. Dies ist insofern wenig aussagekräftig, als dass die betrachteten Kontostände nur einen Teil der Transaktion berücksichtigen. Sobald auch der andere Transaktionsteilnehmer seinen neuen Kontostand an den Statistikserver gesendet hat, korrigiert sich auch der globale Zustand.

Ein glaubwürdigeres Verfahren zur Ermittlung des globalen Zustands eines Netzwerkes bieten Snapshot-Algorithmen. Hierbei existiert ein Initiator, der den globalen Zustand anhand von verteilten Zuständen ermitteln möchte, beispielsweise unser Statistikserver. Bei Verwendung des in der Vorlesung besprochenen Snapshot-Algorithmus von Chandy und Lamport wäre es dann nicht mehr nötig, dass der Statistikserver alle Teilnehmer des Netzwerks kennt.

Der Initiator beginnt das Verfahren damit, Marker-Nachrichten an alle ihm bekannten Teilnehmer zu senden und Transaktions-Nachrichten auf diesen Kanälen aufzuzeichnen. Bei einer Kombination des Snapshot-Algorithmus mit dem 2-Phase Distributed Commit wie in unserem Fall, müssten ausschließlich PerformTransaction-Nachrichten betrachtet werden, da nur aufgrund dieser Nachrichten eine Änderung des Kontostandes durchgeführt wird.

Jeder Teilnehmer speichert bei Erhalt einer Marker-Nachricht seinen aktuellen Kontostand, sendet ebenfalls Marker-Nachrichten an alle bekannten Nachbarn und auch an den Sender der erhaltenen Marker-Nachricht und zeichnet dann ebenfalls Transaktions-Nachricht auf diesen Kanälen auf. Diese Aufzeichnung endet, sobald der Teilnehmer auf diesem Kanal einen Marker zurückgesendet bekommen hat.

Wenn jeder Teilnehmer auf allen Kanälen eine Marker-Nachricht zurückbekommen hat, sind alle Aufzeichnungen beendet und der globale Zustand liegt verteilt bei allen Teilnehmern vor. Die gespeicherten Kontostände aller Teilnehmer werden dann zusammen mit allen aufgezeichneten Transaktions-Nachrichten an den Initiator gesendet.

Da in unserem Fall der Statistikserver den Initiator darstellt, setzt er aus den Nachrichten der Teilnehmer die im Netzwerk vorhandene Geldmenge zusammen, vergleicht sie mit der initialen Geldmenge und meldet den globalen Zustand des Netzwerks.

VI. Fazit

Die Projektarbeit im Laufe des Semesters hat sehr gut funktioniert und alle Funktionalitäten wurden termingerecht implementiert. Die von uns durchgeführten und zum Teil in dieser Abgabe dokumentierten Testdurchläufe haben an verschiedenen Stellen Fehler aufgedeckt, die wir fast ausnahmslos beheben konnten, um anschließend einen finalen, fehlerfreien Test durchzuführen. Lediglich bei der Implementierung des Statistikservers ist Fehlerverhalten aufgetreten, dessen Quelle wir nicht ausfindig machen konnten.

Nach einer kurzen Einarbeitungsphase konnten wir uns in dem verwendeten Akka-Framework sehr gut zurechtfinden. Die Kommunikation zwischen den verschiedenen Teilnehmern des verteilten Systems wurde hierdurch zwar erleichtert, im Umgang mit dem Fehlverhalten des Statistikservers war es aber eher hinderlich, keinen tiefen Einblick in die konkrete Kommunikation zu haben.

Insgesamt stellt dieses Praxisseminar eine sehr gute Ergänzung zu den in der Vorlesung behandelten theoretischen Aspekten Verteilter Systeme dar und bietet eine ideale Möglichkeit, verteilte Algorithmen wie das 2-Phase Distributed Commit in der Praxis anzuwenden.