Seminar Verteilte Systeme - Ausarbeitung

Alexander Stein 4551334
Jakob Warkotsch 4587134

24. Juli 2015

1. Beschreibung des FUCoin Systems

   Im Laufe des Seminars wurde von uns eine Version von FUCoin, dem verteilten Wallet-System, entwickelt. Hierfür haben wir die Programmiersprache Scala und das Akka-Framework verwendet. Im Folgenden werden die Objekte, Funktionalitäten und deren Implementierung beschrieben.

   (a) Wallet

   Wallets sind Teilnehmer des FUCoin-Netzwerk. Sie bestehen aus einem `ActualWallet` - einer Geldbörse, die einen Kontostand enthält und auf der Transaktionen ausgeführt werden, `localNeighbors` - einer Liste anderer Wallets zu denen eine direkte Verbindung besteht und `backedUpNeighbors` - eine Map von Wallet-Namen auf `ActualWallets`, die den Zustand eines gesicherten `ActualWallet`-Objekts enthalten.

   (b) Join

   Join besteht aus zwei Schritten und dient dem Eintritt eines Wallets in das System. Im ersten Schritt erhält ein Wallet als Einstiegspunkt eine Adresse eines anderen Wallets und schickt diesem ein `ActionJoin` um den Eintrittswunsch zu signalisieren. Im zweiten Schritt fügt der Empfänger des `ActionJoin` den Sender zu seinen `localNeighbors` hinzu und sendet ihm seine `localNeighbors`, die dann vom neu beigetretenen Wallet übernommen werden.

   (c) StoreOrUpdate

   Beim `StoreOrUpdate` wird das `ActualWallet` eines Teilnehmers bei einem anderen hinterlegt. Das Objekt wird serialisiert versendet und beim Empfänger mit dem Namen des Senders in den `backedUpNeighbors` hinterlegt. Wenn bereits ein Eintrag vorhanden war, wird dieser überschrieben.

   (d) Invalidate

   Der Sender einer Invalidate-Nachricht, löscht seinen Eintrag aus den `backedUpNeighbors` des Empfängers.

   (e) Transaction

   Der Sender einer Transaction zieht den Betrag der Transaction vo seinem eigenen `ActualWallet` ab. Der Empfänger addiert den Betrag auf sein `ActualWallet`. Transactions können sowohl positive, als auch negative Beträge enthalten.

   (f) SearchMyWallet

   Hierbei wird über die Liste der `localNeighbors` iteriert und jedem ein `ActionSearchMyWallet` gesendet. Die Empfänger sehen in ihren `backedUpNeighbors` nach, ob ein Eintrag für den Sender vorhanden ist. Falls ja, wird das Wallet an den Sender gesendet und dieser übernimmt es als sein `ActualWallet`.

   (g) Command Line Interface

   Um unsere Applikation testen zu können haben wir eine kleine Kommandozeilenschnittstelle programmiert, die beim Start des Wallet-Programms zeilenweise Befehle einliest und

beispielsweise beim Befehl `‘‘transfer Foo 20’’` versucht, dem Wallet “Foo” 20 FUCoins zu überweisen.

(h) Statistics Server

Der Statistics Server empfängt `backedUpNeighbors` und `ActualWallet` aller Teilnehmer des Netzwerks und kann zu einem beliebigen Zeitpunkt überprüfen, ob ein Zustand (ein Kontostand) der `backedUpNeighbors` ungleich dem des entsprechenden `ActualWallet` ist.

2. Snapshot Algorithmus

Bei der Wahl des verteilten Algorithmus haben wir uns für den Snapshot-Algorithmus von Chandy und Lamport entschieden. Ein zuvor ausgewählter Initiator sendet Marker mit seiner `ActorRef` an all seine `localNeighbors`, die wiederum den Marker an ihre `localNeighbors` weiter versenden. Gleichzeitig wird das `ActualWallet` gespeichert und eingehende Transaktionen aller `localNeighbors` aufgezeichnet. Sobald ein Wallet von allen `localNeighbors` einen Marker empfangen hat, wird der Zustand des `ActualWallet`, sowie die aufgezeichneten Transkationen an den Initiator zurückgesendet.

Hierfür wurde eine entsprechende Nachrichtenklasse erstellt, die als einzige Instanzvariable die `ActorRef` des Initiators enthält. Im Folgenden werden die wichtigsten Methoden des Snapshots in Pseudo-Scala dokumentiert:

(a) initiateSnapshot

```
startSnapshot(self)
```

(b) startSnaphot(initiator: ActorRef)

```
walletSnapshot := actualWallet.copy()
for ((_, neighbor) <- localNeighbors) neighbor ! ActionSnapshot(
    initiator)
```

(c) makeSnapshot(initiator: ActorRef)

```
if (isRecording()):
  startSnaphot(initiator)

receivedMarkers += 1
respondIfSnapshotComplete(initiator: ActorRef)
```

(d) respondIfSnapshotComplete(initiator: ActorRef)

```
if (receivedMarkers == localNeighbors.size):
  sendSnapshot(initiator)
```

(e) isRecording

```
return walletSnapshot != null
```

(f) recordTransactionIfRecording(walletName: String, amount: Int)

```
// wird beim Empfang einer Transaction ausgefuehrt
if (isRecording()):
  channelSnapshots[name] += amount
```

(g) sendSnaphot(initiator: ActorRef)

```
initiator ! ActionSnapshotResponse(walletSnapshot, channelSnapshots)
walletSnapshot = null
receivedMarkers = 0
```

3. Herausforderungen und Probleme

   Ein Problem stellte die steile Lernkurve von Akka dar. Bei der Programmierung des Systems haben wir einen Großteil der Zeit mit Herumprobieren und periphären Aufgaben wie beispielsweise der Konfiguration von TCP für Akka-Systeme verbracht. Es wäre hilfreich gewesen, hierfür ein gegebenes Grundgerüst verwenden zu können und sich bei der Implementierung auf die wesentlichen Komponenten zu beschränken.

   Eine weitere Herausforderung bestand in der korrekten Verknüpfung und dem Nachrichtenaustausch zwischen Wallets. Vor allem die Kommunikation zweier Teilnehmer, die keine direkte Verbindung zu einander haben, d.h. nicht `localNeighbors` sind, stellte eine Hürde dar, weshalb wir uns schlussendlich aus Zeitgründen auf eine vereinfachte Variante ohne Gossipping reduzieren mussten.

4. Auswertung

   In der folgenden Abbildung sieht man einen Durchlauf des Programms mit manuellen Eingaben von der Kommandozeile. Es werden zwei Prozesse gestartet, von denen sich einer zum anderen verbindet und dann bei Nachricht 2 eine Transaktion über 12 FUCoins ausführt. Prozess “Foo” speichert anschließend seinen Zustand bei “Bar”, verlässt das Netzwerk und stellt seinen vorherigen Zustand in Nachricht 5 wieder her.

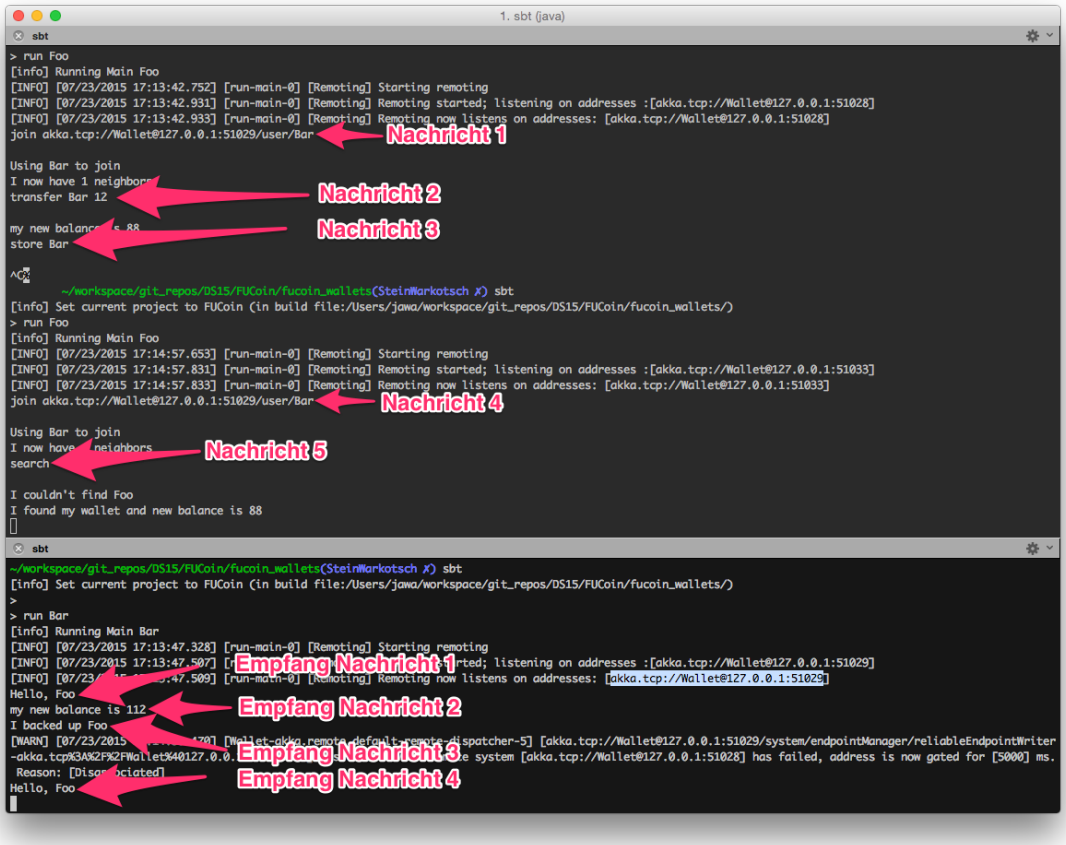

Abbildung 1: Beispiel von Backup und Transaktion

Im nächsten Beispiel wird der Snapshot-Algorithmus mit drei Prozessen (Proc1, Proc2, Proc3) ausgeführt, die jeweils alle miteinander verbunden sind. Proc1 initiiert einen Snapshot und bekommt von allen Netzwerkteilnehmern den lokalen Snapshot geschickt und gibt diesen aus. Anschließend werden einige Transaktionen durchgeführt und ein zweiter Snapshot, diesmal mit den neuen Werten, gestartet und ausgegeben.

Mit der momentanen Implementierung kommt es noch zu Problemen, wenn während des Snaps-

hots neue `localNeighbors` hinzukommen oder ein Wallet das andere kennt, aber nicht umgekehrt. Das Problem könnte gelöst werden, indem entweder für die Initiierung ein anderer Nachrichtentyp als für die Marker verwendet wird oder indem Snapshots eine UID bekommen.

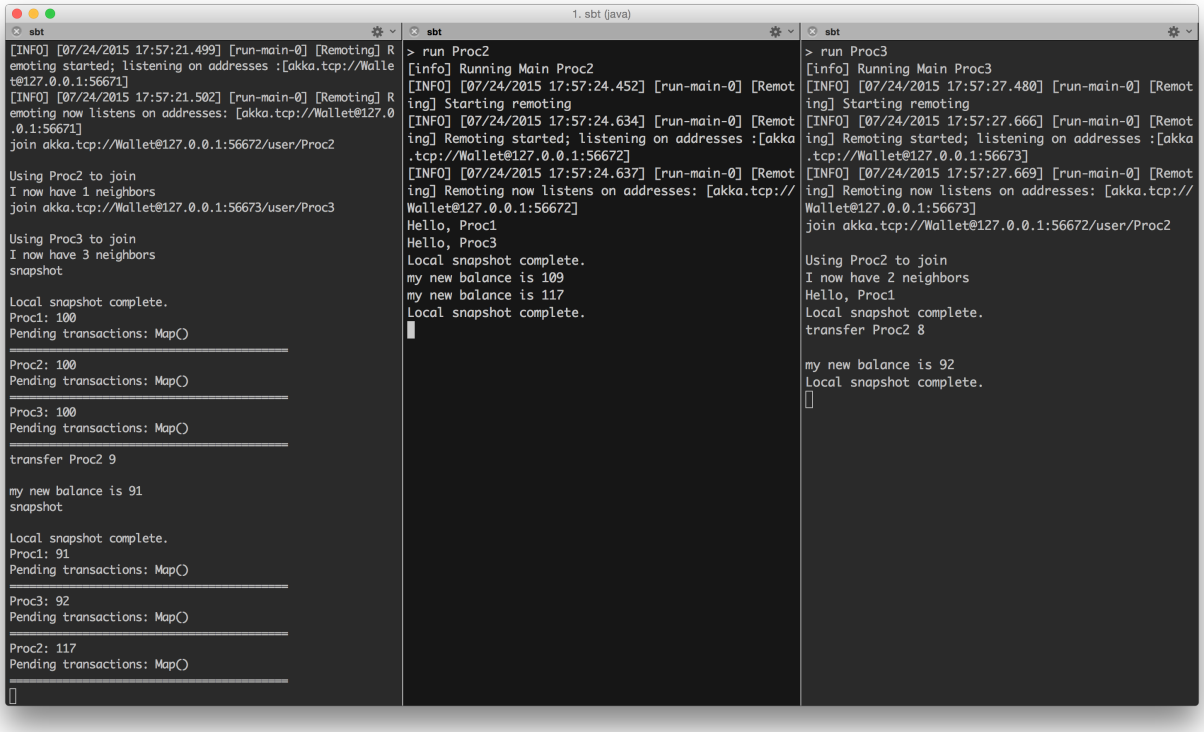

Abbildung 2: Beispiel eines Snapshot