VIS-Übungsblatt 1

Hennings, Regorz, Röder, Budde, Warrelmann · WiSe 2017

1 Transparenz

b)

- **Ortstransparenz**
  Ortstransparenz bewirkt, dass der Benutzer eines Systems nicht wissen muss, wo die Informationen, die er benötigt im Netzwerk abgelegt sind und das System den Zugriff auf die Daten mit einem Befehl gewährleistet, der nicht vom Ort der Information im Netzwerk abhängt. Ein Beispiel, in dem Ortstransparenz gewünscht ist, ist wenn ein Facebook-Nutzer Informationen abrufen möchte. Dieser Nutzer soll nur die Adresse von Facebook abrufen müssen und die Informationen bereitgestellt bekommen. Im Hintergrund kann Facebook die Informationen dann auf einem beliegen Server speichern, ohne dass der Nutzer beispielsweise die IP-Adresse des Servers kennen muss. Ein Beispiel, bei dem Ortstransparenz nicht gewünscht ist, ist dem Speichern von Daten auf den einem Computer zur Verfügung stehenden Speichermedien. Würde hier Ortstransparenz gelten, würde der Computer selbst entscheiden, welche Daten er beispielsweise auf einen USB-Stick auslagert und welche er auf der Festplatte speichert. Dies würde zum Beispiel dazu führen, dass nach dem Anschließen eines USB-Sticks plötzlich Daten, die vorher auf der Festplatte des Computers lagen, ohne Interaktion des Nutzers auf den Stick ausgelagert werden. Außerdem wäre es nicht mehr möglich bewusst zu entscheiden, welche Daten mittels des USB-Sticks von einem Rechner auf den anderen übertragen werden sollen.
- **Fehlertransparenz**
  Fehlertransparenz ermöglicht beispielsweise, dass Teile des Systems ausfallen können oder nicht mehr erreichbar sein können, ohne dass der Nutzer dies mitbekommt oder sich um das Wiederherstellen der Daten von den ausgefallenen Servern kümmern muss. Fehlertransparenz ist zum Beispiel beim Betrieb von Servern durch Facebook wichtig. Fällt einer dieser Server aus, muss Facebook weiterhin erreichbar sein, ohne dass der Nutzer etwas von dem Ausfall mitbekommt. Ein Beispiel in dem Fehlertransparenz nicht erwünscht ist, ist beim Abschicken einer Banktransaktion. Wenn beim Erstellen der Transaktion ein Fehler auftritt, soll dieser Fehler dem Nutzer angezeigt werden, damit er reagieren kann und die Transaktion eventuell erneut abschicken kann und die Anwendung nicht in dem Glauben schließt, die Transaktion wäre abgeschickt worden.

4 Globale Zustände

Diese Aufgabe wurde mithilfe des Kapitels 11.5 der vierten Ausgabe des Buchs „Distributes Systems - Concepts and Design“ von George Coulouris, Jean Dollimore und Tim Kindberg gelöst.

a)

Das erste Problem des Ansatzes ist, dass es nicht möglich ist, in einem verteilten System dafür zu sorgen, dass die Prozesse Zugriff auf eine synchrone Uhr haben, d.h. eine Uhr, die für die Prozesse

genau zur gleichen Zeit tickt. Die Uhren der Prozesse werden den vereinbarten Zeitpunkt also nicht gleichzeitig erreichen, sodass sich auf diese Weise kein global konsistenter Zustand abspeichern lässt.

Doch selbst wenn es mögliche wäre, eine solche synchrone Uhr zu realisieren, würde das Speichern des lokalen Zustandes nicht dazu führen, dass sich ein globaler Zustand ableiten ließe. Dies lässt sich gut an dem Beispiel veranschaulichen, in dem versucht wird, über die lokalen Zustände darauf zu schießen, ob ein Algorithmus terminiert hat. Das Problem sind in diesem Falle Nachrichten, die sich bei der Aufnahme des Zustandes zwischen den Prozessen befinden können. Wenn ein Algorithmus die Prozesse durch Nachrichten aktiviert, kann es passieren, dass ein Prozess eine Nachricht an einen anderen Prozess schickt und danach alle Prozesse pausiert sind. Erst wenn die Nachricht angekommen ist, würde der Algorithmus fortfahren. Wenn nun während der Übertragung der Nachricht die lokalen Zustände aufgenommen würden, so würde festgestellt werden, dass alle Prozesse pausiert sind. Hieraus würde mit der beschriebenen Methode fälschlicherweise gefolgert werden, dass der Algorithmus terminiert hat.

b)

Um einen *Cut* zu definieren, muss zunächst der Begriff der *History* erklärt werden. Der Ablauf eines Prozesses kann durch eine *History* angegeben werden. Hierbei ist die *History* die Liste aller Aktionen durch den Prozess. Hierbei lassen sich die Aktionen in interne Aktionen, bei denen der Prozess nicht mit anderen Prozessen interagiert, Aktionen für das Versenden von Nachrichten und Aktionen für das Empfangen von Nachrichten unterteilen. Die bisher ausgeführten Aktionen eines Prozesses lassen sich durch einen Präfix der *History* des Prozesses beschreiben.
Ein *Cut* ist die Vereinigung von solchen Präfixen, wobei für jeden Prozess genau ein Präfix enthalten ist. Ein *Cut* enthält also alle Aktionen der Prozesse bis zu einem bestimmten Zeitpunkt, der durch den *Cut* angegeben ist. Dieser Zeitpunkt wird jedoch für jeden Prozess separat angegeben. Das bedeutet, es ist möglich, dass in einem *Cut* eine Aktion enthalten ist, die für das Empfangen einer Nachricht seht, obwohl die Aktion, die für das Versenden dieser Nachricht steht, nicht im *Cut* enthalten ist. Das Bestimmen eines *Cuts* reicht also nicht aus, um einen globalen Zustand zu bestimmen. Hierfür wird ein konsistenter *Cut* benötigt. Ein konsistenter *Cut* enthält für alle empfangenden Aktionen, eine entsprechende versendende Aktion. In einem inkonsistenten *Cut* ist dies nicht gegeben. Es ist also möglich, dass zu einem Zeitpunkt genau die Aktionen eines konsistenten *Cuts* durchgeführt wurden. Aus einem konsistenten *Cut* lässt sich also ein globaler Zustand ableiten.

c)

Mit dem *Snapshot*-Algorithmus lässt sich ein globaler Zustand in einem Netzwerk bestimmen. Damit der Algorithmus eine korrekten globalen Zustand bestimmen kann und terminiert, müssen folgende Eigenschaften für das Netzwerk gelten:

- Keiner der Prozesse kann fehlschlagen
- Keine bestehende Verbindung zwischen zwei Prozessen kann während der Ausführung getrennt werden

- Alle verschickten Nachrichten kommen korrekt an
- Jede Verbindung zwischen zwei Prozessen lässt sich nur in eine Richtung verwenden
- Nachrichten werden in der Reihenfolge übertragen, in der sie abgeschickt wurden
- Es gibt von jedem Prozess einen Pfad über die Verbindungen zwischen den Prozessen zu jedem anderen Prozess

Der *Snapshot*-Algorithmus wird von jedem Prozess des Netzwerks ausgeführt. Zunächst soll beschrieben werden, wie sich ein Prozess verhält. Der *Snapshot*-Algorithmus wird auf einem Prozess durch den Erhalt einer sogenannten *Marker*-Nachricht aktiviert. Diese Nachricht kann mehrfach bei einem Prozess eintreffen. Wenn eine solche Nachricht das erste Mal eintrifft, sichert der Prozess seinen lokalen Zustand und initialisiert für jeden eintreffenden Kanal, also für jede Verbindung, die von einem anderen Prozess zu diesem Prozess führt, eine leere Menge. In dieser leeren Menge sollen die Nachrichten gespeichert werden, die sich im aufgenommenen globalen Zustand zwischen den beiden Prozessen befinden. Für den Kanal, über den die erste *Marker*-Nachricht eingetroffen ist, wird diese leere Menge bereits gespeichert. Direkt im Anschluss sendet der Prozess (ebenfalls nur, wenn er das erste Mal eine *Marker*-Nachricht empfangen hat) eine Nachricht über jeden ausgehenden Kanal. Wenn nun eine normale Nachricht über einen Kanal eintrifft, wird diese Nachricht in die entsprechende Menge für den Kanal gelegt. Wenn nun über einen Kanal eine weitere *Marker*-Nachricht eintrifft, wird auch die Menge der Nachrichten für diesen Kanal gespeichert.

Auf diese Weise, wird der Zustand von jedem Prozess und jedem Kanal aufgenommen. Diese Zustände können anschließend zu dem globalen Zustand zusammengefügt werden. Dies ist jedoch nicht Teil des *Snapshot*-Algorithmus. Anzumerken ist, dass es sich bei dem aufgenommen Zustand nicht um einen tatsächlich aufgetretenen globalen Zustand handeln muss, sondern nur um einen korrekten Zustand, der hätte auftreten können.
Der *Snapshot*-Algorithmus kann von einem beliebigen Prozess begonnen werden, in dem dieser beginnt so zu agieren, als hätte er eine *Marker*-Nachricht empfangen. Der Algorithmus kann auch von mehreren Prozessen gleichzeitig begonnen werden, wenn sich die *Marker*-Nachrichten unterscheiden lassen.

Für die Korrektheit des Algorithmus muss gezeigt werden, dass immer ein konsistenter *Cut* erzeugt wird. Dies lässt sich leicht dadurch zeigen, dass es - bei erfüllten Vorbedingungen für den Algorithmus - unmöglich ist einen Zustand abzuleiten, in dem eine empfangene Nachricht von einem Prozess B aufgenommen wurde, aber die entsprechende Sendung des sendenden Prozesses A nicht im *Snapshot* abgebildet wird. Angenommen es gäbe eine Nachricht m, die vor dem Aufnehmen des Zustandes von Prozess B bereits verarbeitet wurde, aber nach dem Aufnehmen des Zustandes von Prozess A verschickt wurde. Die Nachricht muss bei B vor der ersten *Marker*-Nachricht eingetroffen sein, da B seinen Zustand beim Eintreffen noch nicht aufgenommen hat. Außerdem muss A zunächst eine *Marker*-Nachricht an B geschickt haben und dann die Nachricht m, da A vor dem Verschicken von m seinen Zustand aufgenommen hat und nach dem *Snapshot*-Algorithmus im Anschluss direkt die *Marker*-Nachricht an alle ausgehenden Kanäle schicken muss. Da sich m und die *Marker*-Nachricht aber aufgrund der Eigenschaften des Netzwerkes nicht überholen können, ergibt sich ein Widerspruch. Also ist der vom *Marker*-Algorithmus aufgenommene globale Zustand korrekt.

Der Algorithmus terminiert, wenn durch das Aktivieren des Algorithmus auf einem Prozess nach endlicher Zeit jeder Prozess von jedem eingehenden Kanal eine *Marker*-Nachricht empfängt. Da eine Eigenschaft des Netzwerkes ist, dass es von jedem Prozess einen Prad über die Verbindungen zwischen den Prozessen zu jedem anderen Prozess gibt, ist dies gegeben und des Algorithmus muss terminieren.