Verteilte Algorithmen

Aufgabenblatt 9

(zum 30. Januar 2013)

Gruppe 08
Björn Stabel 222128
Tim Strehlow 316594
Friedrich Maiwald 350570

Aufgabe 9.1: Transaktionen

a) Wozu sind Transaktionen notwendig? Geben Sie mindestens ein Beispiel.

Transaktionen bezeichnen mehrere Zugriffe, die zusammen genommen entweder fehlerfrei und vollständig, oder im Fehlerfall gar nicht ausgeführt werden, und somit als logische Einheit betrachtet werden können. Damit wird nach deren Ausführung das System in einem konsistenten Zustand hinterlassen.
Ein typisches Beispiel sind Zugriffe auf Datenbanksysteme. Tritt während eines Datenbankzugriffs ein Fehler auf, sodass die Transaktion nicht fehlerfrei zu Ende geführt wird, werden alle bis dahin während der Transaktion vorgenommen einzelnen Änderungen wieder rückgängig gemacht, um den Zustand vor Beginn der Transaktion wieder herzustellen.
Ein anderes abstrakteres Beispiel wäre eine Überweisung, wobei die Transaktion aus dem Abziehen des Betrages von einem Konto und dem Gutschreiben auf einem anderen Konto besteht. Die Transaktion muss sowohl korrekt als auch als komplett durchgeführt werden. Wenn es etwa bei der Gutschrift auf dem Empfängerkonto zu Problemen kommt, muss der Betrag wieder auf das Absenderkonto gezahlt werden, beziehungsweise von diesem erst gar nicht offiziell abgebucht werden.

b) Erklären Sie die ACID Eigenschaften und deren Relevanz.

A – Atomicity: Alles oder Nichts – Die atomaren Operationen einer Transaktion werden entweder komplett ausgeführt oder gar nicht.
C – Consistency: Eine Transaktion überführt das System von einem konsistenten Status wieder in einen konsistenten Status.
I – Isolation: Zwischenstände innerhalb einer Transaktion, die durch die atomaren Operationen erreicht werden, sind außerhalb der Transaktion nicht sichtbar.
D – Durability: Der Status nach erfolgreichem Abschluss einer Transaktion ist auf jeden Fall dauerhaft gespeichert.

c) Wodurch können diese Eigenschaften gefährdet werden? Nennen Sie Beispiele, die zur Verletzung führen.

Gleichzeitig ablaufende Transaktionen können zum Beispiel die genannten Eigenschaften gefährden, wenn diese auf dieselben Daten zugreifen. Laufen die Transaktionen nicht korrekt isoliert von einander, greifen sie auf zwischenzeitlich geänderte Daten zu. Das kann zu falschen Ergebnissen und im Fall eines Rollbacks zu inkonsistenten Datenzuständen führen.
Auch fehlerhafte Umgebungen können Transaktionen gefährden. Wenn die Software- oder Hardware-Umgebung nicht fehlerfrei arbeitet, können beispielsweise Ergebnisse von eigentlich erfolgreichen Transaktionen nicht auf die Festplatte geschrieben und somit permanent gespeichert werden.
In verteilten Datenbanken ist es schwieriger, die ACID-Eigenschaften zu garantieren, da hier bei einer verteilten Transaktion mehrere Instanzen über diese wachen müssen und zur korrekten

Abwicklung der Transaktion sich möglicherweise auf eine vergleichsweise unzuverlässige Netzwerkverbindung verlassen müssen.

Aufgabe 9.2: Eigenschaften von Plänen

a) Was ist der Unterschied zwischen einem serialisierbaren Plan (serializable schedule) und einem seriellen Plan (serial schedule)?

Ein serieller Plan führt die einzelnen Operation pro Transaktion zusammengefasst hintereinander aus (zuerst alle Operationen von Prozess 1, anschließend alle von Prozess 2,...).
In einem serialisierbaren Plan muss das so nicht gegeben sein, allerdings muss zu ihm ein äquivalenter serieller Plan existieren, so dass sein Ergebnis das selbe ist.

b) Zeigen Sie, dass der Einsatz von Two Phase Locking (2PL) nicht immer zu einem striktem Plan (strict schedule) führt!

Ein Plan ist strikt, wenn keine Transaktion Daten liest/überschreibt, die noch nicht commited wurden.
Im 2PL werden einzelnen Transaktionen Locks zugeteilt (Lese- oder Schreib-Locks). Wenn eine Transaktion ein Lock freigibt, kann es keinen neuen Lock beantragen. Lese-Locks können gleichzeitig mehrere vergeben werden (Shared Lock), wenn aber ein Schreib-Lock zugesichert wurde, können keine weiteren Locks mehr vergeben werden (exclusive lock).
Beim einfachen 2PL kann eine Transaktion den Schreib-Lock schon vor dem Ende wieder freigeben, wodurch wieder Lese-Locks an andere Transaktionen vergeben werden können. Dadurch können von verschiedenen Transaktionen Lesezugriffe auf beschriebene Daten, die noch nicht commited wurden ausgeführt werden, was die Eigeschaft eines strikten Planes verletzen würde.

c) Warum resultiert aus dem Einsatz des strikten 2PL immer ein strikter Plan (strict schedule)?

Im strikten 2PL werden Schreib-Locks immer bis zum Ende einer Transaktion (commit/abort) gehalten. Dadurch kann der in b) beschriebene Fall nicht auftreten, wodurch sich ein strikter Plan ergibt.

d) Warum müssen beim strikten 2PL nur Schreib-Sperren (write locks) bis zum Ende der Transaktion gehalten werden?

Reine Lese-Zugriffe ändern den Zustand des Systems nicht, Schreib-Zugriffe schon, deswegen muss der Lock bis zum Commit, erhalten bleiben, da zu diesem Zeitpunkt erst die Änderungen komplett übernommen werden.

Aufgabe 9.3: Lock Escalation

Was ist Lock Escalation und wie kann sichergestellt werden, dass die verschiedenen Locks stets konsistent sind und sich nicht widersprechen?

Mit *Lock Escalation* wird ein Verfahren genannt, um die Granularität von Locks anzupassen. Wenn normalerweise eine feine Granularität für Locks benutzt wird (bei Festplattenzugriffen beispielsweise auf Datei-Ebene), dann kann das bei größeren Transaktionen zu vielen Locks und damit einem hohen Aufwand führen. Als Gegenmaßnahme wird die Lock-Granularität dynamisch erhöht (beispielsweise auf Ordner-Ebene) und damit der verbundene Aufwand wieder gesenkt.
Damit Locks sich generell nicht widersprechen, müssen sie zu einander kompatibel sein. Nach einem read lock für ein Datum dürfen danach nur weiteren read locks erteilt werden, keine write locks (*shared lock*). Wenn ein write lock gewährt wurde, darf kein anderes Lock mehr für das entsprechende Datum erteilt werden (*exclusive lock*). Außerdem können in bestimmten Fällen read locks zu write locks hochgestuft werden beziehungsweise umgekehrt heruntergestuft werden. Diese Regln sorgen dafür, dass die Locks insgesamt konsistent bleiben und Transaktionen auch bei mehreren parallelen Locks immer fehlerfrei durchgeführt werden können.

Aufgabe 9.4: 2PC

Implementieren Sie das 2PC Protokoll für verteilte Transaktionen. Testen Sie ihre Implementierung mit folgenden Vorgaben:

a) Der Koordinator erhält ein Commit sowie ein Abort Command.

b) Alle Teilnehmer entscheiden sich für Vote Commit sowie ein Teilnehmer entscheidet sich für Vote Abort.

Nehmen Sie den fehlerfreien Fall an. Der Koordinator muss nicht durch ein Wählverfahren bestimmt werden, sondern darf statisch vorgegeben werden. Geben Sie zusätzlich zu ihrer Implementierung auch eine Beschreibung ab wie sich ihr Algorithmus unter den Vorgaben verhält.

Da laut Algorithmus alle Knoten zumindest den Koordinator kennen, haben wir uns für ein voll vermaschtes Netz und gegen zusätzlichen Routing-Aufwand entschieden.
Alle Knoten können dem Koordinator (Knoten 0) ein initiales Commit- oder Abort-Kommando schicken. Der Koordinator färbt sich darauf bis zum Ende der verteilten Transaktion grau und schickt im Fall vom einem Commit-Kommando eine Prepare-Nachricht an alle beteiligten Knoten. Diese versuchen intern ihren Teil der Transaktion bis zum Commit durchzuführen, und antworten bei Erfolg mit einem Vote-Commit und bei Fehlern mit einem Vote-Abort (wir haben Vote-Abort nicht einem Knoten statisch zugewiesen, sondern alle Knoten schicken mit 75% Wahrscheinlichkeit ein Vote-Commit – mit mehreren Durchläufen sollten so alle Fälle abgedeckt sein). Wenn alle Votes bei ihm eingetroffen sind, schickt der Koordinator je nach Ergebnis den Befehl zum Commit oder zum Zurückrollen an alle Knoten und wartet auf deren Bestätigungen. Sind diese alle eingetroffen, ist die verteilte Transaktion beendet und der Koordinator färbt sich wieder schwarz.
Bekommt der Koordinator initial ein Abort-Kommando, sendet er sofort den Abort-Befehl an alle Knoten weiter und wartet auf deren Bestätigungen.
Da wir den fehlerfreien Fall annehmen, funktioniert unser Algorithmus sowohl bei erfolgreichen als auch bei nicht erfolgreichen Transaktionen wie erwartet. Die Implementierung der Knoten ist aber sehr simpel, daher können sie theoretisch gleichzeitig laufende Transaktionen nicht auseinander halten, der exakte Ablauf des Algorithmus ist in diesem Fall nicht gesichert.

Aufgabe 9.5: Selbststabilisierender Spannbaum

Betrachten Sie den in der Vorlesung vorgestellten Algorithmus zur selbststabilisierenden Spannbaumkonstruktion.

a) Welches Verhalten zeigt der Algorithmus beim Auftreten folgender Fehler:

- *Ausfall bzw. Wiedereingliederung von Knoten oder Verbindungen,*
- *Korrumpierung lokaler Datenstrukturen,*
- *Verfälschung/Verlorengehen von Nachrichten?*

Bei *Ausfall bzw. Wiedereingliederung von Knoten oder Verbindungen* wird der Spannbaum unter den neuen Umständen neu konstruiert, sobald der Timeout in der Wurzel abgelaufen ist. Sollte die Wurzel ausgefallen sein oder eine neuer Knoten mit niedriger ID dazugekommen sein, so wird in jedem Fall nach einer bestimmten Zeit der Knoten mit der niedrigsten ID einen Heartbeat aussenden. Dieser Heartbeat erreicht nach und nach über die vorhandenen Verbindungen alle Knoten, und nach dem Algorithmus wird dadurch der Spannbaum neu konstruiert. Mit einem angepassten Timout (siehe b) ist der Algorithmus gut auf Änderungen bei Knoten oder Verbindungen vorbereitet.
Bei der *Korrumpierung lokaler Datenstrukturen* ist es hingegen anders. Sollte ein Knoten die eigene ID falsch gespeichert haben, verhält er sich wie ein Knoten mit dieser ID. Es kommt aber zu Problemen, wenn dadurch zwei Knoten mit scheinbar gleicher ID mit demselben dritten knoten verbunden sind, da dieser nun nicht mehr anhand der Knoten-ID unterscheiden und entscheiden kann. Falsche lokale Informationen über das Level oder den Vater-Knoten führen in der Regel zu einem weniger optimalen Spannbaum, da dadurch die Pfadlänge zu den Blättern steigt. Sollte der Timout zu klein sein, führt das zu erhöhter Netzwerklast. Ist nur bei der eigentlich Wurzel der Timout zu groß gesetzt, führt es dazu, dass andere Knoten zwischenzeitlich Heartbeats aussenden und somit auch unnötige Netzwerklast erzeugen. Korrumpierung lokaler Datenstrukturen lässt den

Algorithmus in der Regel normal ausgeführen, aber der Ergebnis-Spannbaum ist womöglich weniger effektiv konstruiert.
Das *Verfälschung/Verlorengehen von Nachrichten* ist dagegen ein echtes Problem für den Algorithmus. Je nach Fehlertoleranz in der Implementierung können verfälschte Nachrichten dafür sorgen, dass eigentlich korrekt funktionierende Knoten abstürzen (Knoten A schickt an Knoten B, dass Knoten B die Wurzel sei). Das Verlorengehen von Nachrichten kann im schlimmsten Fall dazu führen, dass ein Knoten ewig auf die verlorengegangene Nachricht wartet, da er schon von allen anderen Nachbarn eine Nachricht bekommen hat.

b) In größeren, dynamischen Systemen ist zumindest das Auftreten von Topologieänderungen der Normalfall. Was bedeutet das für den vorgestellten Algorithmus?
Das Zeitintervall für den Timeout sollte den Topologieänderungen angepasst werden. Wenn Topologieänderungen wahrscheinlich häufig auftreten, sollte der Timeout hinreichend klein gewählt werden, damit die Änderungen möglichst schnell berücksichtigt werden und ein neuer aktualisierter Spannbaum konstruiert werden kann. Natürlich sollte der Timout nicht zu gering sein, so dass die Netzwerk-Last durch den selbststabilisierenden Spannbaum-Algorithmus zu groß wird.

c) Wie könnte ein geeigneterer selbststabilisierender Spannbaum konstruiert werden? Skizzieren Sie eine eigene Idee oder recherchieren Sie entsprechende wissenschaftliche Arbeiten.
Um einen geeigneteren Spannbaum zu konstruieren, muss man etwas an dem Ergebnis des bisherigen Algorithmus verbessern können. Spontan ist uns dazu eingefallen, dass der entstehende Spannbaum in manchen Fällen optimiert werden könnte. So ist es für Spannbäume meist wünschenswert, wenn die von der Wurzel ausgehen Pfade möglichst gleich lang sind, die Wurzel also in der „Mitte“ des Spannbaums liegt. Dazu könnten die Blätter nach dem Heartbeat auf den Spannbaum-Kanten ihre Level an die Wurzel zurückschicken. Die Wurzel erhält dann von allen Seiten die höchsten Level zugeschickt, und kann bei unterschiedlichen Werten entscheiden, dass es das „Amt der Wurzel“ an ihren Nachbarn in Richtung des höchsten Levels abgibt. Das kann fortgeführt werden, bis eine Wurzel nicht mehr unterschiedlich hohe Level-Werte an ihren Kanten hat. Sollte durch Topologieänderungen die so gefundene optimierte Wurzel nicht mehr in der „Mitte“ des Netzwerks liegen, muss die ganze Prozedur mit vorherigem Finden der kleinsten ID wiederholt werden.