Ausarbeitung

Transaktionen und Nebenläufigkeit

Systemtechnik
5BHITT 2015/16

Melanie Goebel, Tobias Perny

Inhaltsverzeichnis

1 CAP-Theorem

Das CAP-Theorem beschreibt das Verhältnis zwischen Consistency, Availability und Partition Tolerance (Konsistenz, Verfügbarkeit und Partitionstoleranz) indem es besagt, dass nur davon zwei in einem verteilten System gleichzeitig angeboten werden können. Es wurde im Jahre 2000 von Eric Brewer vorgestellt, deswegen wird es auch Brewer-Theorem genannt. Es wurde mit einem Webservice implementiert von mehreren Servern geografisch verteilt vorgestellt. Clients senden Anfragen an einem Server und erwarten von diesen eine Antwort. /citeCAPTheorem2012

Einen Beweis dafür geben Nancy Lynch und Seth Gilbert in ihren Paper [1] Für den Beweis muss man sich ein Szenario vorstellen mit 2 Nodes (N1,N2). N1 enthält einen Algorithmus A, der vertrauenswürdig und fehlerfrei ist. In V wird ein Wert gespeichert, zum Beispiel eine Anzahl für ein Produkt im Onlineshop. In N1 ist v0, dies ist der Initialwert. N2 besitzt einen Algorithmus B, dieser ähnelt sich mit A. A schreibt den Wert V1, B liest den Wert V. Wenn alles nach Plan läuft,

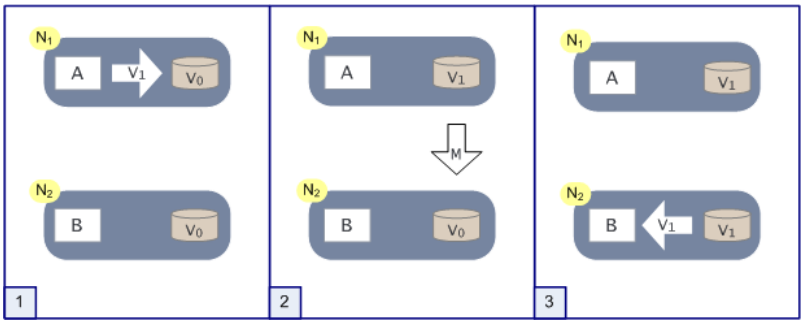

Abbildung 1: Erfolgreich: N2 bekommt die Nachricht und erhälts V1 [3]

schreibt A den Wert in v1. Eine Nachricht M mit den Wert wird von N1 an N2 gesendet. Dieser empfängt die Nachricht und speichert sich den Wert. Somit liest B den selben Wert von V nämlich V1.

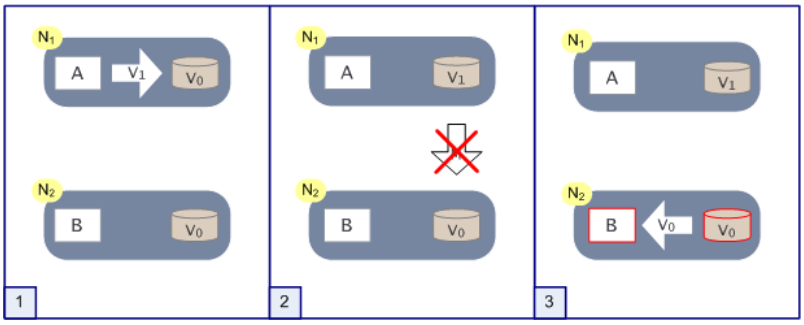

Abbildung 2: Fehlgeschlagen: N2 bekommt nicht die Nachricht und erhält nicht V1 [3]

Wenn sich das Netzwerk partitioniert (N2 erhält die Nachricht von N1 nicht) entsteht eine Inkonsistenz ab Schritt 3. B liest den Wert V0 anstatt den Wert V1, da er keine Änderung erhalten hat.

Wenn man also eine hohe Verfügbarkeit erreichen möchte und Partitionen toleriert, wird der Fall auftreten, dass Nodes als V V0 lesen anstatt den geänderten Wert V1.

1.1 Consistency

Zählt zu den Sicherheitseigenschaften und beschreibt im Groben, dass eine Anfrage eine richtige Antwort erhält. Abhängig was Konsistenz im Detail ist liegt von der Service Spezifikationen ab. Diese Eigenschaft wird auch „Atomic Consistency“ genannt. Um dies zu Erhalten benötigt man eine Reihenfolge für alle Operationen, sodass zu einem Moment nur eine Operation fertiggestellt wird. [1] [2] Als Beispiel für Consistency kann ein Onlineshop gesehen werden. Wenn 2 Personen gleichzeitig das letzte Exemplar beantragen, dürfen nicht beide die Antwort bekommen, dass die Bestellung erfolgreich war und somit zur Zahlung weiter gelangen.

1.2 Availability

In verteilten Systemen muss für eine kontinuierliche Verfügbarkeit, jede Anfrage, die ein nicht fehlerhaften Node erhält, eine Antwort bekommen. Das heißt auch, dass ein Algorithmus schlussendlich beendet werden muss, obwohl in dieser Definition nicht beschrieben wird wie lange ein Algorithmus läuft bevor er beendet wird. [1] [2]

1.3 Partition Tolerance

Dieser Teil des Theorems widmet sich nicht der verteilten Applikation, sondern den Eigenschaften des Systems dahinter. Wenn sich der Server wie ein atomic processor verhält (Applikation läuft nur auf einer Node) funktioniert er oder nicht. Wenn man jedoch die Daten und Logik auf mehrere Nodes verteilt besteht die Gefahr von Partition forming. Dies passiert, wenn sie in verschiedenen Gruppen aufgeteilt werden und wegen eines Fehlers (vereinfacht ein Kabel wird ausgesteckt) nicht mehr miteinander kommunizieren können. Eine solche Trennung wird dabei als üblich eingestuft und wird somit verkraftet. Das bedeutet, dass keine Fehler außer ein Totalausfall eine Antwort verfälschen soll. Wenn man somit eine Partition im Netzwerk hat, verliert man entweder Consistency, weil man Änderungen auf beiden Partitionen erlaubt oder Availability, weil man einen Fehler entdeckt und man das System abstellen muss um dies zu richten. [1] [2] [3]

1.4 Consistency und Availability

Dafür muss darauf geachtet werden, dass Partition Tolerance nicht stattfinden kann. Dies kann erfüllt werden, wenn man die Applikation auf nur einen Server beziehungsweise eher nur auf einen atomically-failing unit zur Verfügung stellt. Für diesen „Verlust“ können 2-Phase-Commits verwendet werden. [2]

1.5 Consistency und Partition Tolerance

Wenn Consistency garantiert wird, können auf Daten zugegriffen werden. Dies schließt Pessimistic Locking ein, da Daten geschützt/in Anspruch genommen werden müssen bis jede Node die Änderung der Daten erhält. Dabei tritt ein Ausfall der Verfügbarkeit der Partitionen auf, da erst darauf zugegriffen werden kann wenn Consistency garantiert werden kann. Ein Beispiel für diese Kombination ist Distributed locking und Quorums. [2]

1.6 Availability und Partition Tolerance

Wenn eine Partition auftritt können die Daten weiterhin genutzt werden, jedoch gibt es keine Garantie, dass es eine richtige Antwort auf die Anfrage ist. Ein Beispiel für diese Nutzung ist Web caching. [2]

2 Zwei-Phasen-Commit

Der Zwei-Phasen-Commit ist eine Methode zur Koordination einer Transaktion zwischen mindestens zwei Ressourcenmanager. Es gewährleisten Datenintegrität bei der Sicherstellung, dass entweder eine Transaktion von allen Ressourcemanager committed werden oder bei allen ein Rollback stattfindet. Die 1. Phase des Zwei-Phasen-Commit wird Abstimmungsphase oder Vorbereitungsphase genannt, die 2. Phase Entscheidungsphase oder Commit/Abort Phase. [4]

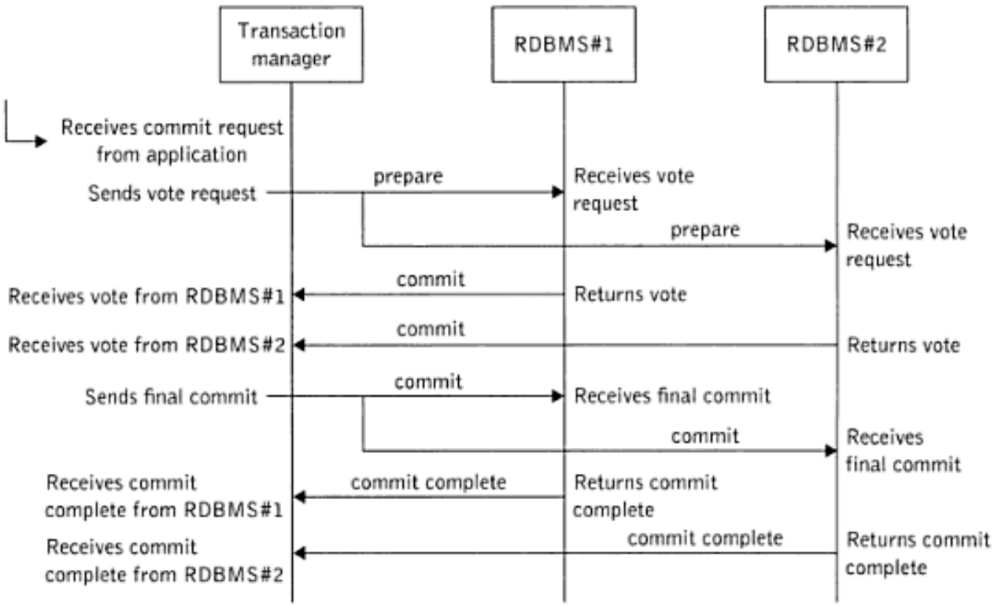

Abbildung 3: Zwei-Phasen-Commit [5]

2.1 Zustände

Insgesamt gibt es drei Zustände in denen ein Koordinator oder Teilnehmer blockiert, weil er auf eine eingehende Nachricht wartet. Der Teilnehmer könnte im INIT-Zustand eine VOTE-REQUEST-Nachricht vom Koordinator erhalten, wenn diese nicht nach einer bestimmten Zeit empfangen wird, bricht der Teilnehmer lokal die Transaktion ab und versendet eine VOTE-ABORT-Nachricht an den Koordinator. Ebenfalls ist der Koordinator im wartenden Zustand, wenn er auf alle Abstimmungen der Teilnehmer warten muss. Dieses wird nach einer bestimmten Zeit abgebrochen und der Koordinator stimmt für einen Abbruch der Transaktion indem er einen GLOBAL-ABORT an alle Teilnehmer sendet. Im Ready-Zustand wartet der Teilnehmer auf die globale Abstimmungsnachricht vom Koordinator. Nach einer bestimmten Zeit jedoch kann der Teilnehmer nicht mehr

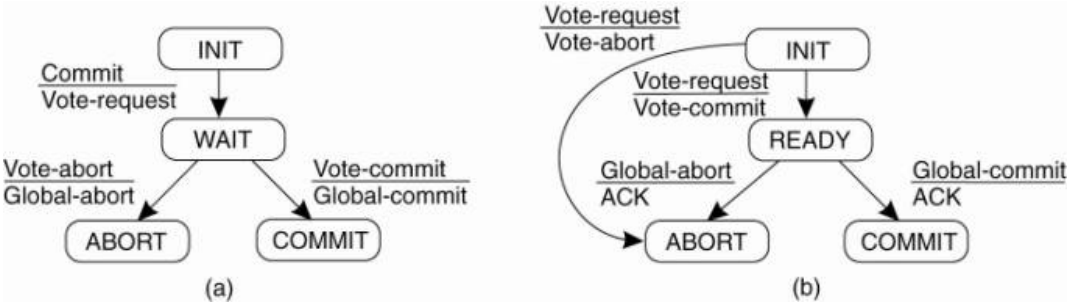

Abbildung 4: Finite Zustandsautomat in 2PC (a) Koordinator, (b) Teilnehmer [4]

entscheiden die Transaktion durchzuführen, eine Lösung ist es den Teilnehmer solange zu blockieren bis der Koordinator die Nachricht senden kann.

Eine andere Lösung wäre es, dass sie der Teilnehmer P beim Teilnehmer Q erkundigen kann um festzustellen ob er den aktuellen Status erhalten hat. Wenn Q die Nachricht COMMIT erhalten hat bevor der Koordinator abgestürzt ist und die Nachricht nicht an P senden konnte, kann P die Information entnehmen und die Transaktion durchführen. Wenn Q die Nachricht ABORT erhalten hat, kann P die Transaktion abbrechen. Die Tabelle zeigt die Aktionen von P, wenn er im Zustand READY ist und Teilnehmer Q kontaktiert:

Zustand von Q	Aktion von P
COMMIT	Gehe zu COMMIT über
ABORT	Gehe zu ABORT über
INIT	Gehe zu ABORT über
READY	Kontaktiere anderen Teilnehmer

Tabelle 1: Aktionen eines Teilnehmer P (Status: READY) und einen Teilnehmer Q kontaktiert. [4]

Wenn die letzte Zeile der Tabelle stattfindet, also der Zustand von P und Q READY ist, kann keine Entscheidung getroffen werden. Deswegen werden die Teilnehmer nach einer bestimmten Zeit blockiert, bis der Koordinator wiederhergestellt ist.

Der Prozess kann nur wiederhergestellt werden wenn sich der Status in einen nicht flüchtigen Speicher ablegen lassen kann. Somit kann ein Teilnehmer, wenn er im Zustand INIT war, kann er sich dazu entscheiden die Transaktion abzubrechen und muss zusätzlich den Koordinator Bescheid sagen. Wenn er sich schon entschieden hat, im Zustand COMMIT oder ABORT, kann er wenn er wiederhergestellt ist die Entscheidung nochmal an den Koordinator senden. Problematisch wird es, wenn er im Zustand READY war. Somit ist er, ähnlich wie schon beschrieben, dazu gezwungen andere Teilnehmer zu kontaktieren. Wenn der Koordinator während er das Zwei-Phasen-Commit-Protokoll startet, sollte er aufzeichnen, dass er in den Zustand WAIT übergeht. Somit kann er die Nachricht VOTE-Request erneut an alle Teilnehmer versenden. Wenn er nachdem er eine Entscheidung getroffen hat (2.Phase) wiederhergestellt ist muss er nur die Entscheidung aufzeichnen und gegebenfalls an alle Teilnehmer erneut senden. [5] [4]

2.2 Abstimmungsphase

Die Abstimmungsphase beginnt beim Senden einer VOTE-REQUEST-Nachricht von dem Koordinator an alle Teilnehmer. Diese erhalten die Nachricht und senden eine VOTE-COMMIT-Nachricht zurück, wenn der Teilnehmer lokal dazu bereit ist seinen Teil der Transaktion mit Commit zu bestätigen oder er sendet eine VOTE-ABORT-Nachricht.

2.3 Entscheidungsphase

Der Koordinator bekommt alle Nachrichten und wertet diese aus. Wenn nur ein Teilnehmer eine VOTE-ABORT-Nachricht verschickt hat, beschließt der Koordinator die Transaktion abzubrechen und verschickt eine GLOBAL-ABORT-Nachricht als Multicast. Wenn alle Teilnehmer jedoch dafür bereit ist, beschließt der Koordinator eine Durchführung der Transaktion und sendet eine GLOBAL-COMMIT-Nachricht an alle Teilnehmer.

3 Drei-Phasen-Commit

Der Drei-Phasen-Commit verhindert das Problem des Zwei-Phasen-Commit, dass Teilnehmer nach einem Absturz des Koordinators teilweise nicht mehr zu einer eindeutig richtigen Entscheidung kommen können. In der Praxis wird der 3PC nicht häufig eingesetzt, da der Zustand bei dem 2PC selten vorkommen. Beim 3PC genügen die Zustände des Koordinators und der Teilnehmer folgender Bedingungen:

- Kein einzelner Zustand ist es möglich um direkt in COMMIT/ABORT-Zustand zu gelangen.
- Kein einzelner Zustand ist es unmöglich eine endgültige Entscheidung zu treffen (Übergang in den COMMIT-Zustand) [4]

3.1 Zustände

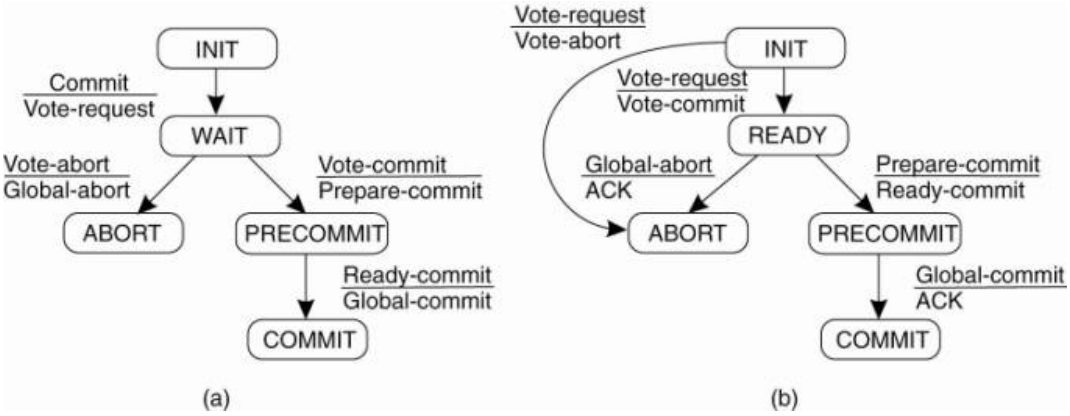

Abbildung 5: Finite Zustandsautomat in 3PC (a) Koordinator, (b) Teilnehmer [4]

Als erstes sendet der Koordinator eine VOTE-REQUEST-Nachricht an alle Teilnehmer, dannach wird auf Antworten der Teilnehmer gewartet. Wenn ein Teilnehmer gegen die Transaktion stimmt, ist auch seine Entscheidung die Transaktion abzubrechen und ein GLOBAL-ABORT wird versendet. Wenn jedoch die endgültige Entscheindung für die Transaktion ist, wird eine PREPARE-COMMIT-Nachricht gesendet. Erst wenn wieder alle Teilnehmer bestätigen wird eine GLOBAL-COMMIT-Nachricht gesendet worauf der Commit durch die Transaktion festgeschrieben wird.

Auch beim 3PC gibt es wenige Zustände bei denen ein Prozess blockiert, während er auf eine Nachricht wartet. Wenn ein Teilnehmer auf eine Aufforderung zur Abstimmung vom Koordinator wartet (Zustand INIT), der Koordinator (Zustand WAIT) auf Stimmen der Teilnehmer wartet und wenn der Koordinator im Zustand PRECOMMIT sich befindet.

Ein Teilnehmer P kann im Zustand READY oder im Zustand PRECOMMIT blockieren. Nach einer Zeitüberschreitung kann P davon ausgehen, dass der Koordinator ausgefallen ist. Genau wie beim 2PC erkundigt sich P bei einem anderen Teilnehmer Q. Wenn sich Teilnehmer Q im Zustand COMMIT/ABORT befindet kann er ebenfalls die Aktion durchführen. Wenn alle Teilnehmer sich im Zustand PRECOMMIT befinden, kann die Transaktion sicher mit COMMIT festgeschrieben werden.

Wichtig ist, dass sich Teilnehmer Q nur dann im Zustand INIT befinden kann, wenn sich kein anderer Teilnehmer im Zustand PRECOMMIT befindet. Ein Teilnehmer kann nur dann den Zustand PRECOMMIT erreichen, wenn der Koordinator sich ebenfalls im Zustand PRECOMMIT befindet. Dies passiert, wenn alle Teilnehmer für das festschreiben des Commits gestimmt haben.

Ein weiteres besonderes Merkmal des 3PC ist die die Behandlung eines Teilnehmers, wenn er abgestürzt ist, er sich im Zustand PRECOMMIT befunden hat und die anderen Teilnehmer den Zustand READY besitzten. In diesem Fall richtet es keinen Schaden an die Transaktion dennoch abzubrechen. Dann könnten die restlichen funktionierenden Prozesse keine endgültige Entscheidung treffen und müssen warten, is der abgestürzte Prozess wiederhergestellt wird. Wenn sich ein funktionierender Teilnehmer in dem Zustand READY befindet, wird kein abgestürzter Teilnehmer der sich in einen anderen Zustand als INIT, ABORT oder PRECOMMIT befindet wiederhegestellt. Deswegen können die anderen Prozesse stets eine entgültige Entscheidung treffen. [4]

3.2 Abstimmungsphase

Selbige Methode der Abstimmungsphase wie beim Zwei-Phasen-Commit. Nach dieser Phase kommt die Vorbereitungsphase.

3.3 Vorbereitungsphase

Wenn nach der Abstimmungsphase die engültige Entscheidung für den Commit ist, wird eine PREPARE-COMMIT-Nachricht gesendet.

Nehmen wir an der Koordinator blockiert in der Vorbereitungsphase nach dem Absenden der PREPARE-COMMIT-Nachricht. Dabei geht er bei einer Zeitüberschreitung davon aus, dass ein Teilnehmer abgestürzt ist, aber von der Abstimmungsphase bekannt ist, dass er für das Festschrei-

ben des Commits gestimmt hat. Der Koordinator sendet den funktionierenden Teilnehmer eine GLOBAL-COMMIT-Nachricht. Dabei verlässt er sich auf ein Wiederherstellungsprotokoll, sodass er seinen Commit festschreiben kann, wenn er wiederhergestellt wurde.

3.4 Entscheidungsphase

Selbige Methode der Entscheidungsphase wie beim Zwei-Phasen-Commit.

4 Zwei-Phasen-Sperrprotokoll

Das Zwei-Phasen-Sperrprotokoll ist eine Methode zur Synchronisierung von Zugriff auf aufgeteilte Daten. Das Zwei-Phasen-Sperrprotokoll funktioniert nur dann, wenn sich alle Transaktionen an das Protokoll halten. Die 1. Phase des Zwei-Phasen-Sperrmodell wird Wachstumsphase oder Expanding-Phase genannt. Die 2. Schrumpfphase oder Shrinking-Phase. Dabei muss man zwei Sperren unterscheiden.

- Schreibsperre/Write-Lock (exclusive)
- Lesesperre/Read-Lock (shared)

Bei einer Schreibsperre kann nur der sperrende Prozess auf das Objekt zugreifen und somit dieses auch ändern. Bei einer Lesesperre können mehrere Prozesse das gesperrte Objekt lesen.

Von dem Zwei-Phasen-Sperrprotokoll gibt es 2 Arten:

- Strikte 2PL
- Konservative/Starke 2PL

Die konservative Art verhindert einen Deadlock. Dies wird umgesetzt indem bei Beginn der Transaktion alle benötigten Sperren auf einmal gesetzt werden. Die strikte Art ist die häufiger verwendete Art von 2PL. Hier werden alle gesetzten Write-Locks erst am Ende der Transaktion freigegeben. Das Kaskadierende Zurücksetzen von sich gegenseitig beeinflussten Transaktionen wird verhindert (auch bekannt als Schneeballeffekt). Der Nachteil ist, dass bei dieser Art Sperren oftmals länger anhalten als sie benötigt werden. [6]

4.1 Wachstumsphase

In der Wachstumsphase dürfen Sperren nur gesetzt werden. Während dieser Phase dürfen Sperren jedoch nicht freigelassen werden.

4.2 Schrumpfphase

In der Schrumpfphase dürfen Sperren frei gelassen werden. Während dieser Phase dürfen nicht neue Sperren gesetzt werden.

5 Long-duration Transaction

Sehr viele Businesstransaktionen sind Long-term-/Long-running-/Long-duration-Tranaktionen. Diese laufen meist wie der Name sagt für eine lange Zeit, sogar für Stunden oder Tage. Daher werden meistens Timelimits gesetzt. Wenn also bei einem Onlineshop zum Beispiel eine Reservierung stattfindet, besitzt die Reservierung ein Timeout. Dannach wird die Transaktion nicht durchgeführt (Rollback).

Weil Transaktionsisolation erreicht wird indem man Ressourcen sperrt, können lang laufende Transaktionen die Nebenläufigkeit auf ein inakzeptables Level bringen. Ebenfalls kann in einer Businesstransaktion die Aktion eines Mitgliedes die Ressourcen aller anderer Transaktionsmitglieder sperren und somit Denial-of-service Attacken provozieren. Es ist also logisch, dass das Isolation-property (welches für das sperren verantwortlich ist) in diesem Fall gelockert werden muss. [7]

6 Anwendung: Java Transaction API

Die Java Transaction API gibt an wie die Kommunikation zwischen Transaktionsmanager und externen Komponenten funktioniert. JTA definiert somit Schnittstellen für den Vertrag zwischen den Transaktionsmanager und 3 Applikationskomponenten[7]:

- Application Program
- Application Server
- Resource Manager

6.1 Transaktionsmanager und Applikationsprogramm

Eine Clientapplikation oder ein Server EJB bei der die Transaktionsrichtslinien in TX_BEAN_MANAGED verwaltet wird können explizit Transaktionen abzugrenzen. Dafür bekommt die Applikation ein e Referenz von einen User Transaktions Objekt. Dieses Objekt implementiert das Interface javax.transaction.UserTransaction. Es definiert unter anderen Methoden damit die Applikation commit, roll back und suspend an Transaktionen benutzen kann.

Die Java-Client-Applikation kann mit der von dem Applikationsserver JNDI-basierten Register erworben werden. Da es keinen Standard-JNDI-Namen dafür gibt, muss dieser schon im vorhinein bekannt sein. In einer EJB wird das User-Transaction Objekt mit den EJBContext bereitgestellt. [7]

6.2 Tranaktionsmanager und Applikationsserver

In JEE werden Container dazu benötigt um CMT (Container-managed transaction) zur Verfügung zu stellen. Ein Container grenzt Transaktionen ab und regelt Threads und gemeinsam genutzte Ressourcen. Dieses benötigt der Applikationsserver um eng mit den Transaktionsmanager zu arbeiten. Die Kommunikation des Transaktionsmanager und Applikationsserver wird durch javax.transaction.TransactionManager Interface definiert.[7]

6.3 Transaktionsmanager und Ressourcenmanager

Der Resourenmanager bietet einen Applikationszugang zu Datenbanken, JMS Queues oder jegliche andere Ressourcen an. Damit der Transaktionsmanager verschiedene Ressourcenmanager koordinieren und synchronisieren, muss ein standardbasiertes, namhaftes Transaktionsprotokoll verwendet werden (z.B.: X/Open XA). [7]

6.4 Java Transaction Service

JTA stellt X/Open Standard Interfaces zur Verfügung. Deswegen kann ein JTA-compliant Transaction Manager eine Transaktion die sich auf mehrere Ressourcenmanager verteilt kontrollieren und koordinieren, dies nennt sich auch verteilte Transaktion. Java Transaction Service definiert wie sich die Ressourcenmanager verständigen und spezifiziert wie der Transaktionsmanager implementiert sein soll.

JTS ist das Java-mapping von CORBA Object Transaction Service (OTS) Spezifikationen. Diese definieren einen Standardmechanismus für das Generieren und Verbreiten von Transaktionskontext zwischen Transaktionsmanagern mittel IIOP (Internet Inter-ORB Protocol). JTA benutzt OTS interfaces für Interoperabilität und Portabilität. [7]

Literatur

[1] Nancy A. Lynch Seth Gilbert. Perspectives on the cap theorem. *IEEE Computer Society*, Februar:30–36, 2012.

[2] Nancy A. Lynch Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, Volume 33 Issue 2:51–59, 2002.

[3] Julian Browne. Brewer's cap theorem. `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem`. zuletzt besucht: 10.1.2016.

[4] A.S. Tanenbaum and M. Van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. Pearson Studium. Addison Wesley Verlag, 2008.

[5] Michael P. Papazoglou. *Web Services: Principles and Technology*. Pearson Education. Pearson, 2008.

[6] Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*, chapter 3, pages 47–111. ADDISON-WESLEY PUBLISHING COMPANY, 2007.

[7] Michael E. Stevens James McGovern, Sameer Tyagi and Sunil Mathew. *Java Web Services Architecture*, chapter 14, pages 583–619. Morgan Kaufmann, 2007.

Tabellenverzeichnis

Listings

Abbildungsverzeichnis

