FREIE UNIVERSITÄT BERLIN
FACHBEREICH MATHEMATIK UND INFORMATIK – INSTITUT FÜR INFORMATIK

Epidemische Algorithmen

Seminararbeit von Konrad Rieck
Betreuer: Manuel Scholz
Sommer 2003

Zusammenfassung

Die Synchronisation von replizierten Datenbanken ist ein Problem, bei dem zwischen Leistungsfähigkeit und temporärer Konsistenz abgewogen werden muss. In dieser Arbeit werden epidemische Algorithmen vorgestellt, die temporäre Konsistenz und Serialisierbarkeit gewährleisten und eine gute Leistungsfähigkeit gegenüber klassischen Synchronisationsverfahren zeigen. Zwei dieser Algorithmen basieren auf pessimistischen bzw. optimistischen Annahmen über die Häufigkeit von Konflikten zwischen Transaktionen. Der dritte Algorithmus arbeitet pessimistisch und nutzt zu dem Quoren um Konflikte aufzulösen.

1 Einleitung

In den letzten zwanzig Jahren haben sich die Informationstechnologie und das Internet rasant entwickelt. Populäre Dienste in diesem weltumspannenden Netz aus Rechnersystemen sind mehreren Millionen Anfragen am Tag ausgesetzt und müssen etliche Gigabyte an Daten in angemessener Zeit verarbeiten.

Solche Dienste sind einer erheblichen Last ausgesetzt, die von einzelnen Rechnersystemen kaum getragen werden kann. Es ist daher zweckmäßig die Last auf autonome Rechner, die über ein Kommunikationsnetz miteinander verbunden sind, zu verteilen. Im Falle eines Datenbanksystems entsteht so eine *verteilte Datenbank* [KE01]. Die autonomen Rechner einer verteilten Datenbank werden als *Stationen* oder *Sites* bezeichnet und nutzen unabhängige physikalische Ressourcen, wie zum Beispiel Hardware, Netzanbindung und Stromversorgung, um eine verbesserte Ausfallsicherheit und Fehlertoleranz zu erreichen.

Nach [KE01] besitzen verteilte Datenbanken zwei entscheidende Eigenschaft:

1. Art der Fragmentierung
   Um Relationen auf mehrere Stationen zu verteilen, müssen diese fragmentiert werden. Man unterscheidet horizontale, vertikale und kombinierte Fragmentierung.

2. Art der Allokation
   Die Allokation gibt an, wie die Fragmente auf die einzelnen Stationen verteilt werden. Man unterscheidet redundanzfrei Allokation und Allokation mit Replikation.

Ein Spezialfall der verteilten Datenbanken stellen die *replizierten Datenbanken* oder *Replicated Databases* dar. Ihre Relationen sind nicht fragmentiert und jede Station allokiert und verwaltet im Idealfall eine exakte Kopie aller Relationen. Updates der Daten erfolgen dezentral und lokal an einer Stationen und werden dann zu den anderen Stationen propagiert.

Da jede Information an jeder Station vorgehalten wird, bieten replizierte Datenbanken einen sehr hohen Schutz bei Ausfällen und Datenverlust. Da alle Operationen an den Daten lokal erfolgen, kann die Last auf alle Stationen gleichmäßig verteilt werden. Die Synchronisation der Daten und die Wahrung der Datenkonsistenz stellen jedoch ein schwieriges Problem dar. Prinzipiell muss hier zwischen der Leistungsfähigkeit des Systems und der temporären Konsistenz aller Replikate abgewogen werden.

In [DGH+87, Sch02] werden drei unterschiedliche Ansätze zur Synchronisation von replizierten Datenbanken vorgestellt, die sich vor allem in der Art und Weise der Propagierung unterscheiden. Die Ansätze sind in Abbildung 1 dargestellt.

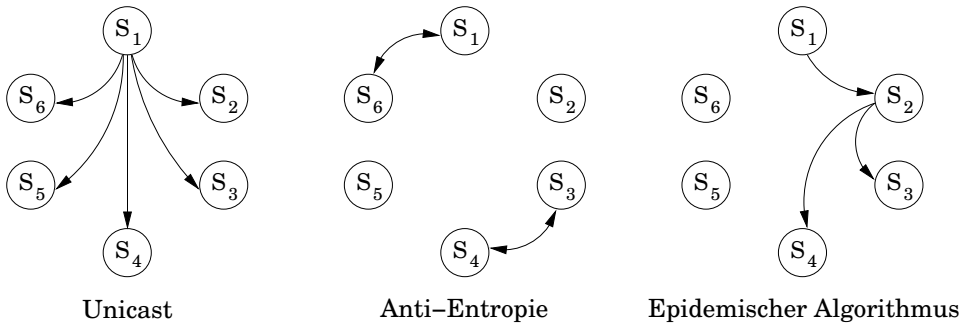

Abbildung 1: Synchronisationsverfahren von replizierten Datenbanken

1. *Unicast* oder *Write-All Locking*
   Bei diesem Verfahren blockiert die Station, auf der Daten lokal geändert wurden, den Schreibzugriff auf allen anderen Stationen, überträgt die Änderungen und gibt anschliessend den Schreibzugriff wieder frei.

   Dieses Verfahren ist problematisch, da alle Stationen ständig verfügbar seien müssen und die Blockade des Schreibzugriffs die Leistungsfähigkeit des Systems sehr stark beeinflusst. Die Konsistenz des Datenbestandes ist allerdings zu jedem Zeitpunkt an jeder Station gewährleistet.

2. *Anti-Entropie*
   Bei dem Verfahren der Anti-Entropie werden lokale Änderungen nicht sofort übertragen. Jede Station kontaktiert in periodischen Abständen eine zufällige andere Station und gleicht mit dieser ihren gesamten Datenbestand ab.

   Das Abgleichen der Datenbestände erzeugt eine hohe Kommunikationslast und beeinträchtigt die Leistungsfähigkeit des Systems. Nach einem gewissen Zeitraum werden alle Replikate durch dieses Verfahren synchronisiert.

3. *Epidemische Algorithmen*
   Diese Algorithmen, die in [DGH+87] auch als *Rumor Mongering* bzw. Gerüchtekrämerei bezeichnet werden, propagieren lokale Änderungen ähnlich einer virulenten Epidemie. Ein lokales Update wird analog zu einem Virus oder Gerücht von der Quellstation zufällig an andere Stationen versandt, die daraufhin das Update an weitere zufällige Stationen übertragen.

   Die epidemischen Algorithmen erzeugen nur wenig Kommunikationslast und können in unzuverlässigen Netzen eingesetzt werden. Ähnlich zu dem anti-entropischen Verfahren wird die Konsistenz der Replikate nach einer gewissen Zeitperiode durch die Propagierung erreicht.

Da die epidemische Algorithmen im Vergleich zu Unicast und dem anti-entropischen Verfahren eine gute Leistungsfähigkeit und eine geringe Kommunikationslast zeigen, werden in dieser Arbeit verschiedene Arten von epidemischen Algorithmen vorgestellt.

Abschnitt 2.1 führt in die epidemische Propagierung von Nachrichten ein. In Abschnitt 2.2 und 2.3 werden dann ereignis- und transaktionsorientierte epidemische Algorithmen vorgestellt und deren Leistungsfähigkeit und Skalierbarkeit in Abschnitt 2.4 zusammengefasst.

2 Epidemische Algorithmen

2.1 Propagierung

Charakteristisch für die epidemischen Algorithmen ist die Art und Weise in der lokale Updates im Kommunikationsnetz propagiert werden. Hierbei synchronisiert ein epidemischer Algorithmus eine replizierte Datenbank, die aus n Stationen $S_1, S_2, \ldots S_n$ besteht. Jede dieser Stationen verfügt ursprünglich über eine vollständige Kopie des gesamten Datenbestandes.

Wird nun an einer Station S_i eine Operation ausgeführt, die lokale Daten modifiziert, so müssen die anderen Stationen entsprechend synchronisiert werden. Die Station S_i beginnt nun periodisch zufällig andere Stationen zu kontaktieren und diesen das Update zu übermitteln. Die kontaktierten Stationen beginnen darauf hin ebenfalls weitere zufällige Stationen zu benachrichtigen. Während dieses Prozesses lassen sich drei Zustände für jede Station unterscheiden [DGH+87]. In Abbildung 2 sind beispielhaft diese Zustände für die Stationen S_1, S_2 und S_3 dargestellt.

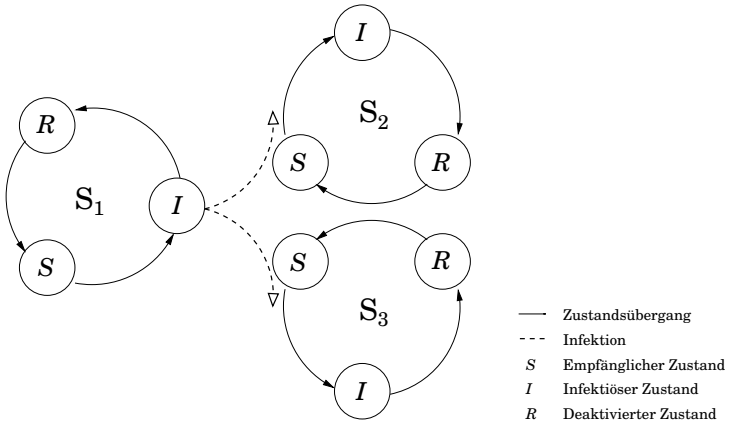

Abbildung 2: Zustandsdiagramm für die Stationen S_1, S_2 und S_3

1. Der *empfängliche* (susceptible) Zustand
   Solange die Replikate der Stationen identisch sind, befindet sich jede Station in diesem Zustand. Der Anteil der empfänglichen Stationen des Systems wird mit s bezeichnet.

2. Der *infektiöse* (infective) Zustand
   Jede Station, deren lokalen Daten modifiziert wurden oder die über ein Update informiert wurde, wird infektiös. Der Anteil der infektiösen Stationen des Systems wird mit i bezeichnet.

3. Der *deaktivierte* (removed) Zustand
   Jede infektiöse Station, die eine andere Station über ein bereits bekanntes Update benachrichtigt, wechselt bei einer solchen Übertrag zufällig mit einer Wahrscheinlichkeit von $\frac{1}{k}$ in den deaktivierten Zustand. Der Anteil der deaktivierten Stationen des Systems wird mit r bezeichnet.

Ein epidemischer Algorithmus kann also nur dann erfolgreich eine replizierte Datenbank synchronisieren, wenn ein lokales Update zu allen Stationen propagiert wird, d.h. keine Station für dieses Update mehr empfänglich bleibt. Nach [DGH+87] lässt sich der Anteil s an Stationen, die von einem Update nicht erreicht werden, wie folgt berechnen.

$$s = e^{-(k+1)(1-s)} \tag{1}$$

Dieser Anteil s hängt exponentiell von dem Wahrscheinlichkeitswert k ab. Während bei $k = 1$ circa 20% der Stationen von einem Update nicht erreicht werden, so erhalten bei $k = 2$ nur noch 6% der Stationen das Update nicht. Ist k entsprechend groß gewählt, so lässt sich der Anteil s stark verringern. Allerdings steigt mit wachsendem k die Kommunikationslast, da die Stationen länger infektiös bleiben. Eine vollständige Ausbreitung kann mit epidemischen Algorithmen nicht garantiert werden. Es verbleibt stets eine Wahrscheinlichkeit, dass Stationen nicht erreicht werden.

2.2 Transfer von Ereignissen

Hat eine infektiöse Station S_i eine noch empfängliche Station S_j ausgewählt, so muss das lokale Update von S_i zu S_j transferiert werden. Da epidemische Algorithmen auch in unzuverlässigen Netzen operieren sollen, kann nicht davon ausgegangen werden, dass Nachrichten in der gleichen Reihenfolge eintreffen, wie sie abgesendet werden. So könnte zum Beispiel die Station S_i drei Updates in der Reihenfolge U_1, U_2, U_3 absenden und die Station S_j diese in der Reihenfolge U_2, U_3, U_1 empfangen.

Um den kausalen Zusammenhang der übertragenen Updates zu erhalten, wird in [AAS97] ein Ereignismodel $\langle E, \rightarrow \rangle$ eingeführt. E entspricht hierbei einer Menge von Operationen auf der lokalen Datenbank, z.B. $E = \{\text{READ}, \text{WRITE}\}$, und $\rightarrow$ einer zeitlichen Relation, die angibt dass ein Ereignis, also eine Operation auf der Datenbank, *vor* einem anderen eingetreten ist, z.B. $e, f \in E : \ e \rightarrow f$ bestimmt, dass e vor f eingetreten ist.

Epidemische Algorithmen verwenden so genannte *Vektoruhren* [Mat89] und lokale Zeiten, um diese zeitliche Relation zu gewährleisten. Die lokale Zeit einer Station ist unabhängig von einer globalen Zeit und wird zum Beispiel mit jeder ein- und ausgehenden Nachricht verändert. Eine Vektoruhr verwaltet nicht nur die lokale Zeit einer Station, sondern die lokalen Zeiten aller Stationen in einem Vektor. Zu einem Ereignis e wird dieser Vektor mittels $\text{TIME}(e)$ berechnet, wobei $\text{TIME}_i(e)$ der lokalen Zeit der Station S_i entspricht. Es ergibt sich folgende Eigenschaft:

$$\forall e, f \in E : \ e \rightarrow f \ \Leftrightarrow \ \text{TIME}(e) \ < \ \text{TIME}(f) \tag{2}$$

Eine Station verwaltet allerdings nicht nur die eigene Vektoruhr, sondern erhält mit jeder Nachricht über Updates auch die aktuellen Vektoruhren der anderen Stationen. Es entsteht eine Matrixuhr, die mit jedem Update um die Vektoruhr der sendenden Station aktualisiert wird.

Die Matrixuhr der Station S_i wird mit T_i bezeichnet und das Element $T_i[j, k]$ entspricht der lokalen Zeit der Station S_k, die der Station S_j bekannt ist. Ist $T_i[j, k] = v$, dann weiß die Station S_i, dass die Station S_j alle Updates von Station S_k bis zur Zeit v erhalten hat. Abbildung 3 verdeutlicht die Beziehungen zwischen lokaler Zeit, Vektor- und Matrixuhr.

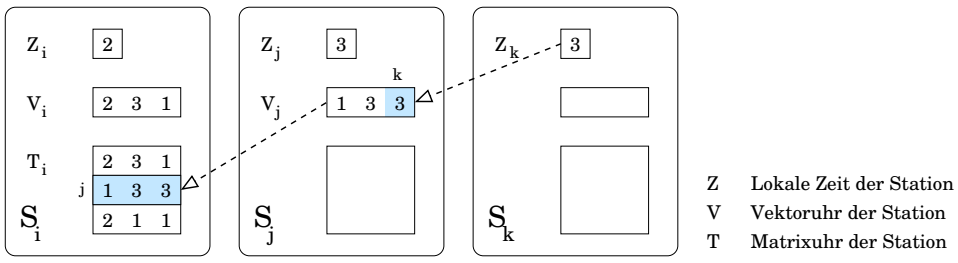

Abbildung 3: Lokale Zeit, Vektor- und Matrixuhr

Mittels der Matrixuhr kann nun eine Station bestimmen, ob eine andere Station schon ein bestimmtes Ereignis erhalten hat. Hierzu kann das folgende Prädikat $\text{HASRECVD}(T_i, t, S_k)$ definiert werden.

$$\text{HASRECVD}(T_i, t, S_k) \equiv T_i[k, \text{SITE}(t)] \geq \text{TIME}_{\text{SITE}(t)}(t) \tag{3}$$

$\text{SITE}(t)$ entspricht der Station an der das Ereignis t aufgetreten ist und $\text{TIME}_{\text{SITE}(t)}(t)$ ist die lokale Zeit der Station $\text{SITE}(t)$ beim Auftreten von t. $\text{HASRECVD}(T_i, t, S_k)$ prüft also von der Station S_i aus, ob die Station S_k das Ereignis t schon erhalten hat. Dieses Prädikat wird auch als *Timetable Eigenschaft* bezeichnet [AAS97, HSAA03].

Wenn die Station S_i nun die Station S_k kontaktiert, um ein Update zu transferieren, übermittelt sie alle Ereignisse für die $\text{HASRECVD}(T_i, t, S_k)$ falsch ist. Die Station S_k übernimmt in einer atomischen Operation alle übermittelten Ereignisse in den lokalen Datenbestand und trägt die Änderungen in das lokale Log L_k ein.

Es gilt dann folgende Eigenschaft für das Log L_k der Station S_k, die auch als *Log Eigenschaft* bezeichnet wird [AAS97, HSAA03].

$$\forall e, f \in E: \quad \text{IF } (e \rightarrow f) \wedge (f \in L_k) \text{ THEN } e \in L_k \tag{4}$$

Wenn also die Station S_k ein Update für das Ereignis f in das Log L_k protokolliert hat und e vor f aufgetreten ist, muss auch das Ereignis e bereits lokal vorgehalten werden und sich ein Eintrag darüber im Log L_k befinden.

2.3 Transfer von Transaktionen

Das bisher beschriebene Verfahren erlaubt die epidemische Verbreitung von einzelnen Ereignissen, die an den jeweiligen Stationen entsprechend ihrer Kausalität eingefügt werden. Es ist allerdings ohne weitere Mechanismen nicht möglich das Verfahren für den Transfer von Transaktionen zu nutzen.

Werden an zwei verschiedenen Stationen gleichzeitig die Transaktionen t_1 und t_2 ausgelöst, wobei $t_1 \equiv \text{READ}(x) \rightarrow \text{WRITE}(y)$ und $t_2 \equiv \text{READ}(y) \rightarrow \text{WRITE}(x)$ entspricht, so entsteht ein Konflikt. Ein solcher Konflikt zweier nicht serialisierbarer Transaktionen wird von dem bisherigen Verfahren nicht berücksichtigt, da nur einzelne Ereignisse betrachtet werden.

In [HSAA03, HAA00, HSAA99] werden daher pessimistische, optimistische und quorum-basierte epidemischen Algorithmen vorgestellt, die derartige Konflikte berücksichtigen. Eine kurze Übersicht über diese Algorithmen ist in [HAA99] aufgeführt.

2.3.1 Pessimistischer Epidemischer Algorithmus

Bei diesem in [HAA00] vorgestellten epidemischen Algorithmus wird eine Transaktion t lokal an einer Station S_i initialisiert, die zu lesenden und zu schreibenden Objekte werden entsprechend

des Zwei-Phasen-Sperrprotokolls (2PL) [KE01] gesperrt und anschließend modifiziert. Sind die Modifikationen abgeschlossen, erzeugt die Station eine so genannte *Precommit-Nachricht*, die neben der Transaktion t, einen Zeitstempel der aktuellen Vektoruhr $\mathrm{TS}(t) = T_i[i, *]$, sowie die Menge der gelesenen $\mathrm{RS}(t)$ und geschriebenen $\mathrm{WS}(t)$ Objekte enthält. $T_i[i, *]$ ist als die i-te Reihe der Matrixuhr zu verstehen.

Diese Nachricht wird nun in das lokale Log L_i eingetragen und über die epidemische Propagierung an die anderen Stationen übermittelt. Mit der Precommit-Nachricht werden alle Lesesperren aufgehoben, die Schreibsperren bleiben bestehen, bis sicher ist, dass t keinen Konflikt auslöst.

Erhält eine Station S_j eine solche Precommit-Nachricht für die Transaktion t, so überprüft sie, ob es im lokalen Log L_j eine weitere Precommit-Nachricht für eine andere Transaktion t' gibt, die die folgenden beiden Bedingungen erfüllt.

1. Die Schnittmengen von $\mathrm{RS}(t)$ und $\mathrm{WS}(t')$ oder $\mathrm{WS}(t)$ und $\mathrm{RS}(t')$ oder $\mathrm{WS}(t)$ und $\mathrm{WS}(t')$ sind nicht leer, d.h. die beiden Transaktionen operieren auf den gleichen Objekten. Die Schnittmenge von $\mathrm{RS}(t)$ und $\mathrm{RS}(t')$ muss nicht berücksichtigt werden, da beide Transaktionen auf diesen Objekten nur lesen.

2. Die beiden Zeitstempel $\mathrm{TS}(t)$ und $\mathrm{TS}(t')$ sind inkompatibel, d.h. es existiert ein r mit $\mathrm{TS}_r(t) > \mathrm{TS}_r(t')$ und ein s mit $\mathrm{TS}_s(t) < \mathrm{TS}_s(t')$. Zwei Zeitstempel sind inkompatibel, wenn Ereignisse der beiden Transaktionen gleichzeitig ausgeführt wurden. Die folgende Schreibweise drückt Inkompatibilität aus $\mathrm{TS}(t) <> \mathrm{TS}(t')$.

Diese zwei Bedingungen lassen sich auch in einem Prädikat $\textsc{Aborted}(t, S_j)$ formulieren, welches prüft, ob ein Konflikt durch die Transaktion t an der Station S_j erzeugt wird.

$$\textsc{Aborted}(t, S_j) \equiv \exists t' \in L_j : \mathrm{TS}(t) <> \mathrm{TS}(t') \wedge \begin{pmatrix} \mathrm{RS}(t) \cap \mathrm{WS}(t') \neq \emptyset & \vee \\ \mathrm{WS}(t) \cap \mathrm{WS}(t') \neq \emptyset & \vee \\ \mathrm{WS}(t) \cap \mathrm{RS}(t') \neq \emptyset & \end{pmatrix} \quad (5)$$

Wenn eine solche Transaktion t' in dem lokalen Log L_j existiert, so stellt die Station S_j einen Konflikt fest und versendet über die epidemische Propagierung eine *Abort-Nachricht* für beide Transaktionen. Alle Stationen einschließlich der Quellstation S_i, die diese Nachricht erhalten brechen die Transaktion t und t' ab, führen wenn nötig ein *Rollback* aus und geben die gesperrten Objekte frei. Kann an der Station S_j kein Konflikt festgestellt werden, so fügt sie die Precommit-Nachricht der Transaktion t in das lokale Log L_j ein, sperrt die Objekte aus $\mathrm{WS}(t)$ und propagiert die Precommit-Nachricht weiter im Netz.

Die Station S_i kann mittels $\textsc{HasRecvd}(T_i, t, S_k)$ feststellen, ob die Transaktion t die Station S_k erreicht hat. Ist dies für alle Stationen der Fall, d.h. es gilt $\forall k : \textsc{HasRecvd}(T_i, t, S_k)$, und S_i

hat keine Abort-Nachricht erhalten, dann kann S_i die Precommit-Nachricht aus dem Log L_i entfernen, ein *Commit* auslösen und die Schreibsperre lösen. Das folgende Prädikat $\text{COMMIT}(t, S_i)$ gibt an, wann die Station S_i die Transaktion permanent in den Datenbestand übernehmen kann.

$$\text{COMMIT}(t, S_i) \equiv \forall k : T_i[k, \text{SITE}(t)] \geq \text{TS}_{\text{SITE}(t)}(t) \wedge \neg\text{ABORTED}(t, S_i) \quad (6)$$

Das beschriebene Verfahren verhindert zuverlässig Konflikte zwischen Transaktionen und wird als pessimistisch bezeichnet, da es unabhängig von der Anzahl der auftreten Konflikte ist und stets beide Transaktionen eines Konfliktes abbricht. Durch den pessimistischen epidemische Algorithmus entstehen zwei Nachteile:

1. Die von einer Transaktion geschriebenen Objekte $\text{WS}(t)$ werden so lange gesperrt, bis alle Stationen die Precommit-Nachricht erhalten haben und keine Abort-Nachricht eingetroffen ist. Diese Sperrung kann die Leistungsfähigkeit des Systems beeinträchtigen.

2. Existiert ein Konflikt zwischen zwei Transaktionen t und t', so werden t und t' abgebrochen. Theoretisch ist es allerdings nur nötig, dass eine Transaktion abgebrochen wird.

2.3.2 Optimistischer Epidemischer Algorithmus

Bei dem pessimistischen epidemischen Algorithmus werden alle geschriebenen Objekte einer Transaktion so lange blockiert, bis festgestellt werden kann, dass an keiner Station ein Konflikt vorliegt. In einer Umgebung in der Konflikte äußerst selten auftreten, kann die Leistungsfähigkeit stark erhöht werden, wenn anstelle der pessimistischen Blockade, optimistisch angenommen wird, dass eine Transaktion zu keinem Konflikt führt.

So werden bei dem optimistischen epidemischen Algorithmus aus [HSAA03] mit den Lesesperren auch alle Schreibsperren gelöst, wenn alle lokalen Modifikation am Datenbestand abgeschlossen sind. Durch diese Änderung werden keine nachfolgenden Operationen mehr blockiert. Auf der anderen Seite stehen nun Daten anderen Stationen zur Verfügung, die möglicherweise einen Konflikt hervorrufen.

Aus diesem Grund wird das Feld *readFrom* zu der Precommit-Nachricht hinzugefügt. Das Feld enthält alle Transaktionen von der die aktuelle Transaktion liest. Meldet eine Station nun einen Konflikt, der über das Prädikat $\text{ABORTED}(t, S_j)$ bestimmt wird, so wird nicht nur die Transaktion t abgebrochen, sondern auch alle neueren Transaktionen, die t in ihrem *readFrom* Feld enthalten. Es entsteht eine Kaskade von Abbrüchen, die üblicherweise bei Datenbanksystemen vermieden wird (Avoid Cascading Aborts).

Ist ein Konflikt aufgetreten, so werden die betroffenen Transaktionen abgebrochen und das Feld *inConflict* gesetzt. Der optimistische Algorithmus kontaktiert dann die Applikationsschicht und löst hier eine *ResolveConflict* Ausnahme aus.

Der optimistische epidemische Algorithmus verlagert also die Konfliktbehebung in die Applikationsschicht und ist anfällig für kaskadierende Abbrüche, daher kann er nur in einer Umgebung eingesetzt werden in der äußerst wenig Konflikte auftreten. Er erreicht allerdings hier auf Grund der kürzeren Sperrungen eine deutlich bessere Leistungsfähigkeit gegenüber der pessimistischen Variante.

2.3.3 Epidemischer Quorum-Algorithmus

Der pessimistische epidemische Algorithmus ist ineffizient, da bei einem Konflikt zwischen zwei Transaktionen, beide Transaktionen abgebrochen werden. Um diese Situation zu verbessern wird in [HSAA99, HSAA03] der epidemische Quorum-Algorithmus eingeführt, der bei einem Konflikt per Votum nur eine der beiden Transaktionen abbricht und auf dem pessimistischen epidemischen Algorithmus basiert.

Grundlage für dieses Votum sind so genannte *Quoren*. Quoren sind Teilmengen von Stationen, die so gewählt werden, dass jede Schnittmenge von zwei Quoren nicht leer ist. Ein einfaches Beispiel ist das Mehrheitsquorum, welches die Mehrheit aller Stationen enthält. Zusätzlich zu dem Begriff Quorum wird der Begriff *Antiquorum* definiert, wobei ein Antiquorum jede Menge ist, die Elemente aus allen Quoren enthält. Abbildung 4 zeigt ein Beispiel mit sechs Stationen für Mehrheitsquoren und ein Antiquorum.

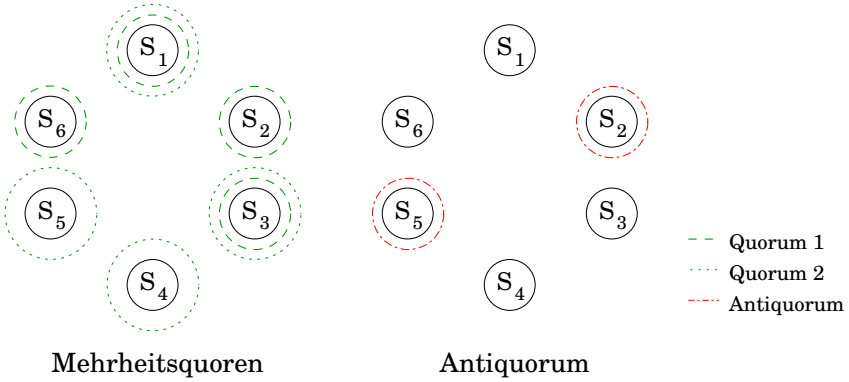

Abbildung 4: Mehrheitsquoren und Antiquorum

Das quorum-basierte Verfahren ermöglicht es nun jeder Station mit *Ja* oder *Nein* für eine Transaktion zu votieren. Das jeweilige Votum wird hierbei einfach mit anderen Nachrichten epidemisch propagiert. Keine Station darf allerdings für zwei im Konflikt stehende Transaktionen mit *Ja* stimmen. Es ergeben sich nun drei Bedingungen.

1. Erhält eine Station ein Quorum von *Ja*-Stimmen für eine Transaktion, so wird ein Commit ausgeführt und die Transaktion permanent in den Datenbestand übernommen.

2. Erhält eine Station ein Antiquorum von *Nein*-Stimmen für eine Transaktion, d.h. es ist nicht mehr möglich, dass ein Quorum von *Ja*-Stimmen gebildet wird, so wird die Transaktion abgebrochen.

3. Erhält eine Station die Nachricht, dass eine Transaktion von einer Station übernommen wurde, weil ein Quorum von *Ja*-Stimmen vorlag, so muss sie die Transaktion auch übernehmen und alle zu dieser Transaktion im Konflikt stehenden Transaktionen abbrechen.

Das so entstehende Verfahren garantiert, dass stets eine von zwei im Konflikt stehenden Transaktionen abgebrochen und eine übernommen wird. Allerdings liegt nicht für jede Transaktion gleich ein Quorum bzw. Antiquorum vor. Daher wird ein neues Feld *uncertain* für jede Transaktion eingeführt, welches angibt, ob der Zustand der Transaktion ungewiss ist, d.h. weder ein Quorum noch ein Antiquorum vorliegen.

Es können nun mehrere ungewisse Transaktionen existieren, die auf die gleichen Objekte schreiben, daher muss zusätzlich eine *Schreibabsichtssperre* gegenüber der Lese- und Schreibsperre eingeführt werden. Jede Transaktion, die in einem ungewissen Zustand ist, hält nun eine Schreibabsichtssperre und keine Schreibsperre auf den Objekten, die sie modifizieren möchte.

Während Lese- und Schreibsperren exklusiv genutzt werden, können mehrere Transaktionen Schreibabsichtssperren für gleiche Objekte verwenden. Eine Schreibabsichtssperre wird allerdings von Lese- und Schreibsperren blockiert. Wird ein Commit für eine ungewisse Transaktion ausgelöst, so wandelt sich die Schreibabsichtssperre in eine einfache Schreibsperre und die Modifikationen können durchgeführt werden.

Die Umwandlung von einer Schreibabsichts- zu einer Schreibsperre kann stets erfolgen, da zwei Transaktionen zeitlich geordnet sind und die ältere als erstes den Zugriff erhält. Gleichzeitig auftretende Transaktionen bilden einen Konflikt, wobei mindestens eine der Transaktionen mittels des quorum-basierten Verfahrens abgebrochen wird.

2.4 Leistungsfähigkeit und Skalierbarkeit

In den technischen Berichten [HAA00, HSAA99] werden alle drei Varianten der epidemischen Algorithmen in komplexen Simulationen evaluiert. Jeder Algorithmus wird hierbei in unterschiedlichen Szenarien, die die Netz-, Precommit- und Commit-Geschwindigkeit variieren, auf replizierten Datenbanken von 5 bis zu 25 Stationen simuliert.

Es zeigt sich das epidemische Algorithmen geeignet sind, um in kleinen bis mittelgroßen Netzen replizierte Daten zu synchronisieren. Bei großen Datenbanksystemen mit tausenden von Stationen können epidemische Algorithmen nicht eingesetzt werden, da zu viel Zeit vergeht bis alle Stationen über die epidemische Propagierung synchronisiert werden.

Der quorum-basierte epidemische Algorithmus zeigt die beste Leistungsfähigkeit in einem Netz mit häufigen Konflikten und ist hier wesentlich schneller als eine einfache Unicast-Variante,

während der optimistische Algorithmus in einem Netz mit wenigen Konflikten dominiert. Ein Vergleich zu anti-entropischen Algorithmen wird in der Literatur nicht aufgeführt.

3 Schlussbemerkungen

Epidemische Algorithmen sind geeignet, um Replikate von verteilten Datenbank in kleinen bis mittelgroßen Netzen zu synchronisieren. Die vorgestellten Algorithmen erlauben sowohl das Synchronisieren von einfachen Operationen als auch von komplexen Transaktionen. Die Konsistenz der Daten und die Serialisierbarkeit bleiben in allen transaktionsorientierten Algorithmen über die Zeit erhalten.

Je nach der konkreten Anwendung und der damit gegebenen Häufigkeit von Konflikten können pessimistische, optimistische oder quorum-basierte epidemische Algorithmen eingesetzt werden. Diese Wahl der konkreten Variante der epidemischen Algorithmen erlaubt einen guten Kompromiss zwischen Leistungsfähigkeit und Konsistenz.

Auch wenn in den zitierten Arbeiten zu epidemischen Algorithmen ausschließlich replizierte Datenbanken synchronisiert werden, kann das Konzept der epidemischen Propagierung auch auf fragmentierte Relationen übertragen werden. Es ist so zum Beispiel denkbar, dass Replikate von Fragmenten an verschiedenen Stationen vorgehalten werden und diese über epidemische Algorithmen synchronisiert werden.

Da epidemische Algorithmen die Ausbreitung von lokalen Updates nur mit einer Wahrscheinlichkeit garantieren, können sie durch den zusätzlichen Einsatz von anderen Synchronisationsverfahren, wie z.B. dem Anti-Entropie Verfahren, verbessert werden und in Kombination eine vollständige Ausbreitung garantieren.

Der große Vorteil der epidemischen Algorithmen liegt aber neben ihrer Leistungsfähigkeit in der „Natürlichkeit“ mit der sich die Daten verbreiten. Keine der Stationen muss alle anderen Stationen kennen, Stationen können zu jedem Zeitpunkt ausfallen und eingefügt werden und die Kommunikationsverbindungen müssen nicht stabil sein. Die epidemischen Algorithmen nutzen genau die Eigenschaften, die auch den natürlichen Viren eine ständige Vermehrung und stetiges Überleben garantieren.

Epidemische Algorithmen eignen sich daher ganz besonders für den Einsatz im Internet und können Datenbestände weltweit synchronisieren, die über Anbindungen unterschiedlichster Qualität miteinander verbunden sind.

Literatur

[AAS97] Divyakant Agrawal, Amr El Abbadi, Robert Steinke. *Epidemic Algorithms in Replicated Databases (Extended Abstract)*. In: Proceedings of the 16th ACM Symposium on Principles of Database Systems, Seite 161–172, Mai 1997. *(http://www.cs.cornell.edu/vogels/Epidemics/epidemic-agrawal.pdf)*

[DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry. *Epidemic Algorithms for Replicated Database Maintenance*. In: Proceedings of the 6th ACM Symposium on Principals of Distributed Computing, Seite 1–12, August 1987. *(http://www.cs.cornell.edu/vogels/Epidemics/epidemic-demers.pdf)*

[HAA99] JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi. *Database Replication: If You Must be Lazy, be Consistent*. In: Proceedings of IEEE Symposium on Reliable Distributed Systems, Seite 304–305, Oktober 1999. *(http://www.cse.scu.edu/~jholliday/srds99.ps)*

[HAA00] JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi. *Database Replication Using Epidemic Update*. Technical Report TRCS00-01, UCSB Computer Science Department, 2000. *(http://www.cs.ucsb.edu/research/trcs/docs/2000-01.ps)*

[HSAA99] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, Amr El Abbadi. *Epidemic Quorums for Managing Replicated Data*. Technical Report TRCS99-32, UCSB Computer Science Department, 1999. *(http://www.cs.ucsb.edu/research/trcs/docs/1999-32.ps)*

[HSAA03] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, Amr El Abbadi. *Epidemic Algorithms for Replicated Databases*. IEEE Transactions on Knowledge and Data Engineering, 15(3), Mai 2003. *(http://www.cse.scu.edu/~jholliday/112609-2.pdf)*

[KE01] Alfons Kemper, Andre Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg Wissenschaftsverlag, München, 4. Edition, 2001. ISBN 3-486-25706-4

[Mat89] Friedemann Mattern. *Virtual Time and Global States of Distributed Systems*. In: Proceedings of the international Workshop on Parallel and Distributed Algorithms, Seite 215–226, 1989.

[Sch02] Christian Schindelhauer. *Epidemische Informationsausbreitung*. In: Algorithmische Grundlagen des Internets, Kapitel 4, Seite 71–96. Universität Paderborn, Juni 2002. Vorlesungsskript. *(http://www.upb.de/cs/ag-madh/vorl/AlGInt02/skript/skript-IV.pdf)*