Datenintegration mittels der regelbasierten Replikationsstrategie RegRess

Heiko Niemann
Lieken IT Service GmbH, heiko.niemann@lieken-it.de

Wilhelm Hasselbring
Universität zu Kiel, Software Engineering, wha@informatik.uni-kiel.de

Abstract: Wenn Daten redundant in heterogenen Systemen gespeichert sind, dann wird für die Integration der Daten eine geeignete Replikationsstrategie benötigt. Die hier vorgestellte regelbasierte Replikationsstrategie RegRess erlaubt eine Konfiguration durch Replikationsregeln und eine Adaption zur Laufzeit, weil bei der Inferenz, durch die Schreib- und Lesezugriffe koordiniert werden, Systemzustände berücksichtigt werden. Dadurch kann für ein Anwendungsszenario hinsichtlich der konkurrierenden Ziele Konsistenz, Verfügbarkeit und Performance ein optimaler Kompromiss erreicht werden.

1 Einleitung

Die Integration von Informationssystemen in heterogenen Systemlandschaften bedeutet häufig ein Abgleich von Daten lokaler, autonomer Systeme. Als Beispiel dienen klinische Informationssysteme, die im Allgemeinen aus mehreren heterogenen Systemen bestehen, wobei jedes System Patientendaten speichert [NHW+02]. Diese Patientendaten müssen zwischen den Systemen ausgetauscht werden. Einerseits soll nun die Datenmanipulation auf den verschiedenen Systemen möglichst unter Wahrung der Konsistenz erfolgen, d.h. die Datenspeicherung muss entweder zentral erfolgen oder die Daten werden quasi zeitgleich abgeglichen. Andererseits sollen die beteiligten Systeme im Allgemeinen ihre Autonomie beibehalten, was eine lokale Speicherung erfordert und einen asynchronen Abgleich bedingt, um die Verfügbarkeit und Performance nicht bzw. nicht zu stark einzuschränken.

Für derartige Anforderungen werden geeignete Replikationsstrategien benötigt, die einen optimalen Kompromiss hinsichtlich der Ziele der Replikation bieten, nämlich hinsichtlich der Konsistenz, der Verfügbarkeit und der Performance. Hierbei handelt es sich um konkurrierende Ziele, d.h. die Verbesserung eines dieser Ziele bedingt im Allgemeinen eine Verschlechterung der anderen Ziele. Beispielsweise ist eine Verbesserung der Konsistenz häufig nur zu Lasten der Verfügbarkeit und der Performance zu erreichen oder eine Erhöhung der Performance ist nur dann möglich, wenn Abstriche bei der Verfügbarkeit und der Konsistenz hingenommen werden. In Abbildung 1 ist dieser Sachverhalt als Zielkonflikt der Datenreplikation dargestellt.

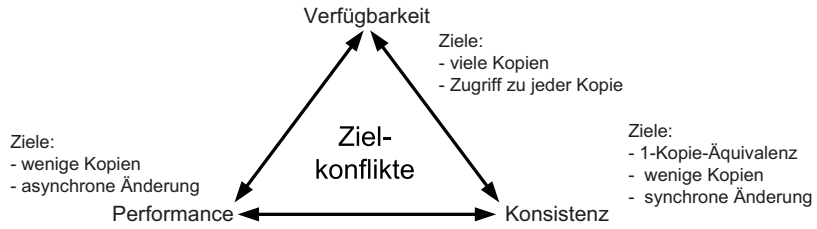

Abbildung 1: Zielkonflikte der Datenreplikation (nach [Has97])

In dieser Arbeit wird die regelbasierte Replikationsstrategie RegRess vorgestellt, die dazu beiträgt, für ein gegebenes Anwendungsszenario einen möglichst optimalen Trade-Off bezüglich der Replikationsziele zu erreichen. RegRess wird durch die Vorgabe einer Menge von anwendungsbezogenen Replikationsregeln konfiguriert. Vor jedem Schreib- oder Lesezugriff werden mittels Inferenz der Regeln die betroffenen Replikate bzw. die Art des Zugriffs bstimmt. Weil bei der Inferenz protokollierte Systemzustände berücksichtigt werden, passt sich RegRess zur Laufzeit dem Systemverhalten an.

Der Aufbau dieser Arbeit gliedert sich wie folgt: In Abschnitt 2 werden zunächst verwandte Arbeiten vorgestellt. Die regelbasierte Replikationsstrategie RegRess wird in Abschnitt 3 erläutert, wobei die Koordination der Zugriffe behandelt und die Regelsprache RRML präsentiert wird. Die RRML erlaubt die Formulierung von Replikationsregeln. In Abschnitt 4 wird auf die Evaluation des Ansatzes eingegangen, die den Entwurf eines Replikationsmanagers enthält, der die Replikationsstrategie RegRess implementiert, sowie den Simulator F4SR, der die Analyse von Replikationsstrategien erlaubt. Abschließend folgt in Abschnitt 5 das Fazit dieser Arbeit.

2 Verwandte Arbeiten

Eine Replikationsstrategie, auch Replikationsverfahren, ist für die Koordination der Zugriffe auf die Replikate verantwortlich [ASC85]. Dabei wird ein Zugriff auf ein logisches Datenobjekt in eine Menge von Zugriffen auf physikalische Datenobjekte, d.h. auf Replikate, übersetzt. Ein wichtiges Korrektheitskriterium für Replikationsstrategien ist die 1-Kopien-Serialisierbarkeit: Eine nebenläufige Ausführung von verteilten Transaktionen ist 1-Kopien-serialisierbar, wenn sie zu einer sequentiellen Ausführung dieser Transaktionen auf einer nicht-replizierten Datenbank äquivalent ist.

Wenn eine Replikationsstrategie die 1-Kopien-Serialisierbarkeit gewährleistet, dann wird bei Zugriffen auf die Replikate die Konsistenz eingehalten. Insbesondere werden Schreib-/Lesekonflikte und Schreib-/Schreibkonflikte vermieden. Wie bereits erwähnt, wird zu Gunsten der Verfügbarkeit und/oder der Performance auch abgeschwächte Konsistenz toleriert, z.B. wird das Lesen veralteter Replikate zugelassen. Ein Replikat R ist genau dann veraltet, wenn eine Änderung auf das zugehörige logische Datenobjekt noch nicht an R

propagiert ist. Dadurch können zumindest temporär Inkonsistenzen auftreten. Wenn alle Replikate alle Schreiboperationen empfangen haben und dann die Replikate desselben logischen Datenobjekts identische Werte repräsentieren, dann wird von letztendlicher Konsistenz gesprochen [TPST98].

Ein ausführlicher Überblick zu Replikationsstrategien ist z.B. in [SS05] zu finden. An dieser Stelle soll kurz auf adaptive Replikationsstrategien eingegangen werden, die sich dadurch auszeichnen, dass die Koordination der Zugriffe sich zur Laufzeit dem Systemverhalten anpasst. Das Mischverfahren „Missing Updates“ (auch „Missing Writes“, [ES83]) ist eine Kombination aus dem Verfahren Read One Write All [TGGL82] und einem Votierungsverfahren, dass im Wesentlichen dem Majority Consensus entspricht [DGMD85].

Die Replikationsstrategie ASPECT [Len97]) hat eine Analogie zum Verfahren „Primary Copy“ [Sto79]. Bei ASPECT wird aber nicht eine feste Primärkopie verwendet, sondern eine Menge von Replikaten, die als Konsistenzinsel bezeichnet wird. Für die „Sekundärkopien“, also die Replikate, die nicht der Konsistenzinsel angehören, können Prädikate für Konsistenzgarantien spezifiziert werden, die sowohl eine zeitliche Dimension als auch räumliche Dimension aufweisen. Die zeitliche Dimension entspricht den Aktualisierungszeitpunkten von Snapshots [AL80] und die räumliche Dimension den Kohärenzbedingungen der Quasi-Copies [ABGMA88].

Auch die Replikationsstratgie „FAS“ [RBSS02] ist an Primary Copy angelehnt. Ein Schreibzugriff wird zunächst an dem so genannten OLTP Node durchgeführt und dann asynchron an so genannte OLAP Nodes propagiert. Für die Replikate wird ein „Frischeindex“ definiert, mittels dem das Alter der Replikate auf den OLAP Nodes bestimmt wird. Bei „Fracs“ [ZZ03] wird ein so genanntes „update window“ für jedes Replikat definiert, mit dem der Umfang der Inkonsistenz bestimmt wird, wobei als Beispiel in [ZZ03] die Anzahl fehlender Updates verwendet wird. Änderungen an einem lokalen Replikat werden so lange gepuffert, bis die Fenstergrenze erreicht ist. Dann werden keine weiteren Änderungen des Replikats angenommen, bis die gepufferten Änderungen propagiert sind.

Demgegenüber verwendet „TACT“ [YV02]) einen dreidimensionalen Vektor für verschiedene Abstandsmaße, um Inkonsistenzen zu begrenzen. „Refresco“ [PGV04, PG06] nutzt unterschiedliche Aktualisierungsstrategien, um insbesondere die Verfügbarkeit von Lesezugriffen zu erhöhen. Auf sehr große, skalierbare Systeme im Umfeld von Webdiensten zielen die Replikationsstrategien „Google’s File System“ (GFS, [GGL03]), „Chain Replication“ [vRS04] und „Niobe“ [MTJ+08], bei denen neben der Erhöhung der Verfügbarkeit auch die Lokalisation der Replikate eine Rolle spielt.

3 Regelbasierte Replikationsstrategie RegRess

Die Idee der Replikationsstrategie RegRess ist es, eine möglichst optimale Adaption zur Laufzeit dadurch zu erreichen, dass vor jedem Schreib- oder Lesezugriff eine Inferenz der Replikationsregeln durchgeführt wird und damit die Koordination der Zugriffe bestimmt wird. Die für ein Anwendungsszenario verwendeten Regeln werden von einem Adminis-

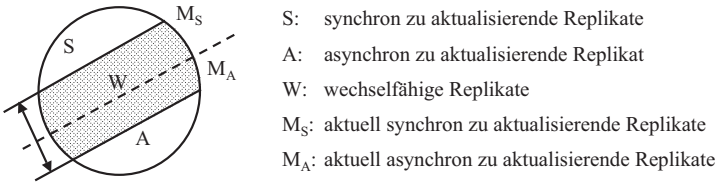

Abbildung 2: Partitionierung in synchron und asynchron zu aktualisierende Replikat

trator oder Anwender als Konfiguration vorab festgelegt. Somit handelt es sich bei RegRess um eine konfigurierbare, adaptive Replikationsstrategie.

Im Folgenden wird zunächst in Abschnitt 3.1 die Koordination der Zugriffe erläutert, um anschließend in Abschnitt 3.2 die Regelsprache RRML vorzustellen, mittels derer Replikationsregeln formuliert werden können. Abschließend wird in Abschnitt 3.3 die Definition des Konsistenzgrades präsentiert, der einerseits als Korrektheitskriterium für Replikationsstrategien dient und andererseits als Metrik bei der Ermittlung des Trade-Offs zwischen Konsistenz, Verfügbarkeit und Performance verwendet werden kann.

3.1 Koordination der Zugriffe

Bei RegRess wird die Koordination von Zugriffen auf die Replikate eines logischen Objekts durch eine vorhergehende Inferenz der Replikationsregeln bestimmt, genauer gesagt, vor dem Zugriff führt ein Regelinterpreter [HR85] eine Inferenz der Regeln durch, um die betroffenen Replikate sowie die Art des Zugriffs auf die Replikate zu ermitteln. Der Zugriff selbst wird nicht durch den Regelinterpreter oder durch Aktionen der Regeln (siehe Abschnitt 3.2) durchgeführt, sondern erfolgt durch einen Replikationsmanager, der das Koordinationsprotokoll implementiert. Weil bei Schreib- und Lesezugriffen unterschiedlich koordiniert wird, werden folgende Inferenzarten unterschieden:

- **InferenzSynchron**: Die betroffenen Replikate bei einem Zugriff werden in synchron und asynchron zu aktualisierende Replikate partitioniert.
- **InferenzAsynchron**: Für ein asynchron (zeitversetzt) zu aktualisierendes Replikat wird der Aktualisierungszeitpunkt festgelegt.
- **InferenzLesen**: Für einen Lesezugriff wird ein Replikat bestimmt, das vorgegebenen Eigenschaften genügt.

zu **InferenzSynchron**: Wenn ein Schreibzugriff auf ein logisches Datenobjekt erfolgt, dann werden bei RegRess grundsätzlich alle Replikate des logischen Datenobjekts geschrieben, aber ein Teil der zugehörigen Replikate wird synchronisiert geschrieben und ist damit konsistent, und der andere Teil wird zeitversetzt (asynchron) geschrieben und

ist damit zumindest temporär inkonsistent. In Abbildung 2 ist die Partitionierung der betroffenen Replikate bei einem Schreibzugriff auf ein logisches Datenobjekt illustriert. Der Ansatz von RegRess ist es nun, dass von diesen Replikaten eine Teilmenge S stets synchron aktualisiert wird, um immer den aktuellen Wert zu repräsentieren, eine Teilmenge A stets asynchron aktualisiert wird, weil hier die Aktualität nicht erste Priorität hat, und eine Teilmenge W die restlichen Replikate enthält, die „wechselfähig“ sind, d.h. die entweder synchron oder asynchron aktualisiert werden können.

Die Mengen S, A und W sind disjunkt und können auch leer sein und W sollte zumindest für einige Replikationseinheiten ungleich der leeren Menge sein, damit überhaupt eine Adaption stattfindet. Die Adaption wird durch Inferenz der Replikationsregeln erreicht, d.h. die wechselfähigen Replikate werden in Abhängigkeit der aktuellen Zustände entweder den synchron oder den asynchron zu aktualisierenden Replikaten zugeschlagen. Das Ergebnis der InferenzSynchron ist also eine Partitionierung der betroffenen Replikate in die Menge M_S für die synchron zu aktualisierenden Replikate und M_A für die asynchron zu aktualisierenden Replikate. Somit sind M_S und M_A disjunkt.

Die Koordination des Schreibzugriffs erfolgt dann derart, dass innerhalb einer Transaktion T die Replikate von M_S synchronisiert geschrieben werden und für die Replikate von M_A entsprechende Aufträge in einer so genannten Replica Queue abgelegt werden, wodurch eine Pufferung ermöglicht wird. T ist im Allgemeinen eine verteilte Transaktion, weil sowohl Replikate auf verschiedenen Rechnern als auch die Replica Queue manipuliert wird. Für T müssen die ACID-Eigenschaften gelten. Dabei kann T selbst in einer globalen Transaktion, die z.B. aus mehreren Schreibzugriffen auf logische Datenobjekte besteht, eingebunden sein, d.h. T ist eine Teiltransaktion. Um die Verfügbarkeit von Schreibzugriffen zu erhöhen, ordnet RegRess ein wechselfähiges Replikat aus der Menge M_S der Menge M_A zu, falls das Replikat beim synchronisierten Schreiben einen Fehler verursacht hat, und setzt die Transaktion T erneut auf. Damit T erneut aufgesetzt werden kann, muss T zurückgerollt werden, d.h. RegRess benötigt als Transaktionskonzept so genannte geschachtelte Transaktionen [Mos85], die das Zurückrollen und erfolgreiche Beenden von Teiltransaktionen erlauben.

zu **InferenzAsynchron**: Durch die InferenzAsynchron wird gesteuert, in welchem Maß ein Replikat veralten darf. Wenn ein Auftrag aus der Replica Queue bearbeitet werden soll, d.h. es soll ein Schreibzugriff auf ein Replikat nachgeholt werden, dann kann mittels Regeln festgelegt werden, ob derzeit ein Schreibzugriff erlaubt ist. Derzeitiges Schreiben steht z.B. entgegen, wenn ein Replikat frühestens nach 15 Minuten aktualisiert werden darf (vergleiche Börsenticker). Wenn der Schreibzugriff durchgeführt werden darf, dann wird bei RegRess unterschieden, ob alle Aufträge des Replikats aus der Replica Queue bearbeitet werden dürfen, um z.B. wieder auf einen konsistenten Zustand des Replikats zu kommen, oder ob genau ein Auftrag bearbeitet wird und bei einem folgenden Auftrag des gleichen Replikats erneut eine Inferenz durchgeführt werden muss, um eine Bedingung wie den oben genannten Zeitverzug von 15 Minuten zu prüfen. Somit liefert die InferenzAsynchron an das Koordinationsprotokoll, das die Bearbeitung der Replica Queue spezifiziert, einen Status, der einen Schreibzugriff verbietet, die Bearbeitung eines Auftrags oder aller Aufträge eines Replikats gestattet.

Bei der Verarbeitung von Schreibzugriffen, ob synchronisiert oder zeitversetzt, koordiniert RegRess die Zugriffe derart, dass Schreib-/Schreibkonflikte verhindert werden. Gleichzeitiges Schreiben auf ein logisches Datenobjekt wird verhindert und die chronologische Verarbeitung von Schreibzugriffen auf dasselbe Replikat wird gewährleistet. So kann ein Replikat nicht synchronisiert geschrieben werden (siehe oben), solange Aufträge in der Replica Queue für dieses Replikat anstehen. Durch diese Restriktionen wird das Korrektheitskriterium „letztendliche Konsistenz“ von RegRess gewährleistet.

zu **InferenzLesen**: RegRess erlaubt einerseits das Lesen eines beliebigen Replikats oder andererseits die Ermittlung einer Menge von Replikaten, die vorgegebenen Eigenschaften genügen. Wenn ein beliebiges Replikat gelesen wird, dann wird keine explizite Koordination benötigt. Andererseits sind dann keine Angaben zur Aktualität bzw. zum Alter des Replikats möglich. Erfolgt die Koordination über RegRess, so können Eigenschaften vorgeben werden, die das Replikat erfüllen muss. Dabei können fachliche Bedingungen berücksichtigt werden, z.B. darf der Versionsabstand zu einem aktuellen Replikat höchstens drei entsprechen, und/oder technische Bedingungen, z.B. darf die Antwortzeit des Replikats 50 Millisekunden nicht überschreiten. Derartige Eigenschaften werden mittels Regeln formuliert. Die InferenzLesen liefert als Ergebnis eine Menge von Replikaten, die den gewünschten Eigenschaften genügen. Wenn mehrere passende Replikate ermittelt werden, dann kann bei einem gescheiterten Lesezugriff ggf. auf ein anderes Replikat ausgewichen werden.

3.2 Regelsprache RRML

Die Regelsprache RRML (Replication Rule Markup Language) dient zur Formulierung von Replikationsregeln und wurde speziell für die Replikationsstrategie RegRess entwickelt. Es werden so genannte Reaktionsregeln verwendet, die hier in der ON-IF-THEN-Darstellung präsentiert werden (vgl. aktive Datenbanken mit ECA-Regeln [WC95]). Das folgende Beispiel zeigt eine mögliche Regel, wobei das Ereignis, die Bedingung und die Aktion umgangsspachlich formuliert sind:

ON	*Objekt O soll geschrieben werden*
IF	*wochentag = Samstag OR wochentag = Sonntag*
THEN	*aktualisiere Replikat R asynchron*

In Abbildung 3 ist die Syntax der RRML vereinfacht dargestellt.

Ereignisse sind entweder Zugriffe, die durchgeführt werden sollen, oder durch Aktionen ausgelöste Ereignisse. Aktionen lösen ein Ereignis gleichen Namens aus und bestimmen das Ergebnis der Inferenz (siehe Abschnitt 3.1). Sowohl Ereignisse als auch Aktionen verwenden Parameter, um Regeln auf unterschiedlichen Granularitätsstufen formulieren zu können. Folgende Parameter sind möglich:

Regeln für die synchrone Aktualisierung

ON	update(...) sync_update(...) async_update(...) changeableTrue(...) changeableFalse(...)	**IF**	*Bedingung*	**THEN**	sync_update(...) async_update(...) changeableTrue(...) changeableFalse(...)

Regeln für die asynchrone Aktualisierung

ON	write(...) later(...) one(...) all(...)	**IF**	*Bedingung*	**THEN**	later(...) one(...) all(...)

Regeln für Lesezugriffe

ON	read(...)	**IF**	*Bedingung*	**THEN**	getConsistency(...) getPerformance(...)

Abbildung 3: Regeln der Replication Rule Markup Language RRML

D	Default-Regeln, die für alle logischen Objekte gelten
E_i	Replikationseinheit, i = 1,..,s; Menge von logischen Objekten
O^o	o-tes logische Datenobjekt; o = 1,..,m; m Anzahl logischer Objekte
K_k	k-te Komponente mit Replikat; k = 1,..,n; n Anzahl Komponenten
R_k^o	Physisches Datenobjekt bzw. Replikat; o,k wie oben

Eine Bedingung in einer Regel ist eine Formel, wobei eine Formel entweder ein Prädikat ist oder durch Verknüpfung von Formeln mit logischen Operatoren entsteht. Ein Prädikat ist zweistellig, wobei eine Funktion mit einer Konstanten durch Vergleichsoperatoren zu einem logischen Ausdruck verknüpft wird. Durch die Funktionen können Gültigkeitszeiträume, z.B. Tageszeit und/oder Datum, fachliche Konsistenzbedingungen, z.B. Versionsabstand, Zeitverzug, Wertdifferenz und/oder Konsistenzgrad, sowie technische Konsistenzbedingungen, z.B. Verfügbarkeit, Datengröße der Objekte und/oder Antwortzeit, ausgedrückt werden.

Folgendes Beispiel realisiert für Werktage auf der Komponente K_2 einen „Börsenticker“, der die logischen Objekte der Replikationseinheit E_1 betrifft:

(1)	ON	*update(E_1)*	IF	*weekday* < 6	THEN	*async_update(K_2)*
(2)	ON	*write(E_1)*	IF	*diff_time* < 900	THEN	*later(K_2)*
(3)	ON	*write(E_1)*	IF	*diff_time* ≥ 900	THEN	*one(K_2)*

Mit der Regel (1) wird erreicht, dass bei Schreibzugriffen auf logischen Objekten, die zur Replikationseinheit E_1 gehören, die Replikate, die auf Komponente K_2 gespeichert sind, asynchron (zeitversetzt) aktualisiert werden, sofern der Wochentag ein Werktag ist. Mit der Regel (2) wird gesetzt, dass die Replikate beim Nachholen der Aktualisierungen später aktualisiert werden, wenn der Zeitverzug kleiner als 15 Minuten (900 Sekunden) ist. Durch Regel (3) wird gesteuert, dass genau ein Auftrag aus der Replica Queue bearbeitet wird. Bei einem weiteren Auftrag für das gleiche Replikat muss eine erneute Inferenz durchgeführt werden, bei dem der Zeitverzug geprüft wird.

Angemerkt sei, dass widersprüchliche Regeln auftreten können. Wenn z.B. eine Regel ein Replikat den synchron zu aktualisierenden Replikaten zuordnet und eine weitere Regel dasselbe Replikat den asynchron zu aktualisierenden Replikaten, dann muss ein derartiger Konflikt behandelt werden. In der RRML können hierfür Widerspruchsregeln formuliert werden, die z.B. Prioritäten nutzen.

3.3 Konsistenzgrad

RegRess erlaubt das Lesen veralteter Replikate, d.h. abgeschwächte Konsistenz wird toleriert. Um ein Maß für die Konsistenz zu erhalten, die durch RegRess, genauer gesagt, durch die jeweiligen Regelmengen, erreicht wird, wird an dieser Stelle der Konsistenzgrad definiert. Dabei werden die beiden bekannten Maße Versionsabstand [ABGMA88] und Frische einer replizierten Datenbank [CGM00] kombiniert, um ein genaueres Maß zu erhalten. Der Konsistenzgrad kann auf folgende Arten ermittelt werden:

- Der **gemessene Konsistenzgrad** wird im realen System gemessen.
- Der **simulierte Konsistenzgrad** wird im Simulationsmodell gemessen.
- Der **berechnete Konsistenzgrad** wird als Vorhersage berechnet.

Der Einfachheithalber wird im Folgenden angenommen, dass eine vollreplizierte Datenbank vorliegt. Es gebe n Komponenten und m logische Objekte, d.h. jede der n Komponenten speichert m Replikate.

zu **gemessener Konsistenzgrad**: Es sei angenommen, dass jedes Replikat eine Versionsnummer speichert. Eine Versionsnummer wird bei jedem Schreibzugriff inkrementiert. Wenn $v(R_a^o, t)$ die Versionsnummer eines aktuellen Replikats zum Zeitpunkt t ist, dann ist der Versionsabstand wie folgt definiert:

$$\delta(R_k^o, t) = v(R_a^o, t) - v(R_k^o, t) \tag{1}$$

Mit dem Versionsabstand wird der Konsistenzgrad eines Replikats $KG_{Rep}(R_k^o, t)$ zum Zeitpunkt t definiert:

$$KG_{Rep}(R_k^o, t) = \frac{1}{1 + \delta(R_k^o, t)} \tag{2}$$

Der KG_{Rep} ist ein normierter Wert im Bereich $[0..1]$. Durch die Verwendung einer Hyperbelfunktion fallen Änderungen des Versionsabstands bei kleinen Versionsabständen beim KG_{Rep} stärker ins Gewicht als bei großen Versionsabständen. Der Konsistenzgrad der replizierten Datenbank $KG_{DB}(t)$ zum Zeitpunkt t wird nun als Mittelung der Konsistenzgrade aller Replikate definiert:

$$KG_{DB}(t) = \frac{1}{n \cdot m} \sum_{k=1}^{n} \sum_{o=1}^{m} KG_{Rep}(R_k^o, t) \tag{3}$$

Die Ermittlung des KG_{DB} bzw. KG_{Rep} kann dadurch vereinfacht werden, dass nicht die Versionsnummern der Replikate gelesen werden, sondern die Aufträge je Replikat in der Replica Queue gezählt werden. Die Anzahl an Aufträgen eines Replikats R entspricht dem Versionsabstand von R.

zu **simulierter Konsistenzgrad**: Um vorab die gewährleistete Konsistenz von RegRess zu bestimmen, kann der Konsistenzgrad entweder analytisch berechnet werden (siehe unten) oder durch Simulation ermittelt werden. Der simulierte Konsistenzgrad entspricht dem gemessenen Konsistenzgrad, jedoch wird anstatt im realen System im Simulationsmodell gemessen. Damit hängen die Ergebnisse von der Affinität des Simulationsmodells zur realen Systemlandschaft ab.

zu **berechneter Konsistenzgrad**: Der berechnete Konsistenzgrad ergibt sich durch ein probabilistisches Modell. Zunächst werden für ein Replikat R_k^o Versionsabstandswahrscheinlichkeiten $p_i(R_k^o)$ benötigt, die die Wahrscheinlichkeit dafür angeben, dass das Replikat R_k^o einen Versionsabstand von i hat. Es gilt:

$$\sum_{i=0}^{\infty} p_i(R_k^o) = 1 \tag{4}$$

Des Weiteren wird eine Zufallsvariable definiert, die dem gemessenen Konsistenzgrad eines Replikats KG_{Rep} entspricht:

$$KG_{ZV}(Versionsabstand = i) = KG_{ZV}(i) = \frac{1}{1+i} \tag{5}$$

Der erwartete Konsistenzgrad ergibt sich nun als Skalarprodukt aus Versionsabstandswahrscheinlichkeiten mit der Zufallsvariable:

$$KG_{ErwRep}(R_k^o) = \sum_{i=0}^{\infty} p_i(R_k^o) \cdot KG_{ZV}(i) = \sum_{i=0}^{\infty} \frac{p_i(R_k^o)}{1+i} \tag{6}$$

Der erwartete Konsistenzgrad der replizierten Datenbank kann wie beim gemessenen Konsistenzgrad durch Mittelung berechnet werden. Vereinfachungen können hier dadurch erreicht werden, dass gleiche Erwartungswerte für Gruppen von Replikaten, z.B. der Replikate einer Replikationseinheit, angenommen werden. Die Versionsabstandswahrscheinlichkeiten können mittels zeitstetiger Markov-Kette berechnet werden, wobei die Versionsabstände als Zustände abgebildet werden. Weil bei einem Versionsabstand von i Aufträge

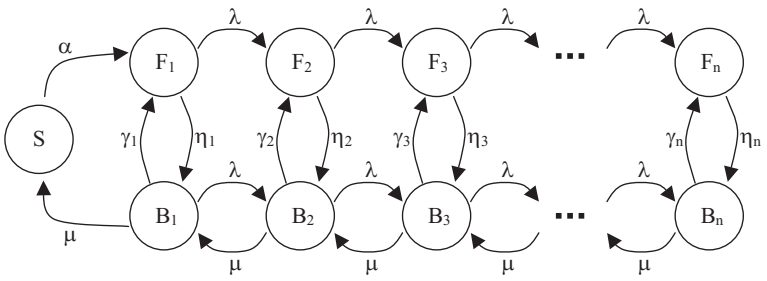

Abbildung 4: Versionsabstandswahrscheinlichkeiten mittels zeitstetiger Markov-Kette

aus der Replica Queue bearbeitet werden können oder derzeit nicht, handelt es sich um eine inhomogene Markov-Kette. Die inhomogene Markov-Kette wird hier in eine homogene Markov-Kette überführt, indem für jeden Versionsabstand zwei Zustände kodiert werden.

Die Abbildung 4 zeigt ein Muster einer hier benötigten Markov-Kette. Dieses Muster muss für konkrete Regeln angepasst werden. Dabei werden die Schreibrate λ, mit der auf das Replikat geschrieben wird, die Bearbeitungsrate μ der Replica Queue und die Umschaltraten α, γ_i und η_i für den konkreten Fall benötigt. Damit kann die stationäre Verteilung der Markov-Kette berechnet werden. Die Versionsabstandswahrscheinlichkeiten p_i ergeben sich dann dadurch, dass die entsprechenden stationären Verteilungen der beiden Zustände F_i und B_i addiert werden bzw. p_0 gleich der stationären Verteilung des Zustandes S gesetzt wird.

4 Evaluation

4.1 Software-Architektur KARMA

In Abbildung 5 ist die grobe Softwarearchitektur des KARMA (konfigurierbarer, adaptiver Replikationsmanager) dargestellt, die insbesondere die Schnittstellen des Replikationsmanagers zeigt. Dabei ist die Architektur an das X/Open-Modell eines zentralisierten Transaktionssystems angelehnt, wobei der KARMA nach dem X/Open-Modell ein Anwendungsprogramm ist, also die Anwendung, die auf die Ressourcen und den Transaktionsmanager zugreift. Demzufolge wird hier auf eine explizite Darstellung der Lokalisation der Replikate verzichtet und analog zu dem X/Open-Modell angenommen, dass jede Komponente mit Replikat einen Ressourcenmanager vorgeschaltet hat, der entsprechende Schnittstellen für Zugriffe auf die Replikate anbietet.

Wie in Abbildung 5 zu sehen ist, bietet die KARMA-Komponente bzw. eine Instanz der KARMA-Komponente mit dem Namen RegRess die Schnittstelle `ILogicalAccess` an, über die ein Client logische Zugriffe, d.h. Zugriffe auf logische Objekte, durchführt. Der KARMA benötigt eine Schnittstelle `IExtendedTx` zu einem Transaktionsmanager,

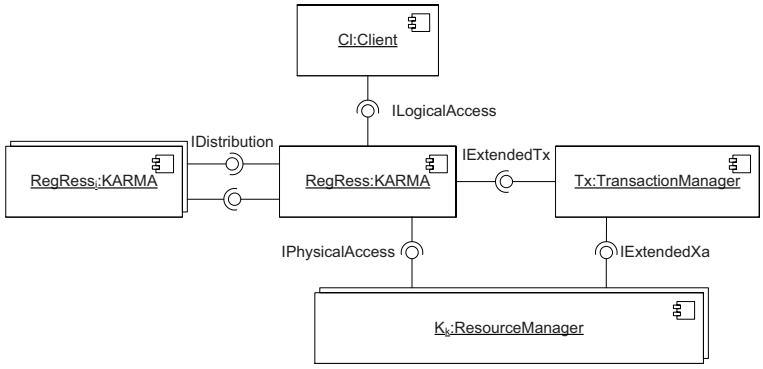

Abbildung 5: Grob-Architektur des KARMA

der die verteilten Transaktionen bzw. Teiltransaktionen synchronisiert. Da gegenüber dem X/Open-Modell die dortige TX-Schnittstelle erweitert wurde, wird hier von der Schnittstelle `IExtendedTx` gesprochen. Die wesentlichen Methoden der Schnittstelle sind den Beginn und das Ende von Transaktionen sowie, als Erweiterung, den Beginn und das Ende von Teiltransaktionen mitzuteilen. Der Transaktionsmanager steuert die jeweiligen lokalen Transaktionen, die auf den Komponenten mit Replikat durchgeführt werden, über die erweiterte XA-Schnittstelle, d.h. ein Ressourcenmanager einer Komponente mit Replikat muss die Schnittstelle `IExtendedXa` anbieten. Des Weiteren bietet ein Ressourcenmanager die Schnittstelle `IPhysicalAccess`, worüber der KARMA Schreib- oder Lesezugriffe auf die jeweiligen Replikate einer Komponente mit Replikat abwickelt.

Wenn es sich um einen verteilten KARMA handelt, d.h. der Replikationsmanager ist mehrfach auf verteilten Systemen vorhanden, dann müssen gemeinsame Daten wie z.B. die Replica Queue verteilt werden. Dafür benötigt der KARMA die Schnittstellen `IDistribution` zu anderen Replikationsmanagern, die RegRess implementieren (in der Abbildung 5 sind diese durch den Index i im Instanzennamen gekennzeichnet). Daher bietet ein KARMA eine entsprechende Schnittstelle an, um gemeinsame Daten entgegenzunehmen.

4.2 Simulator F4SR

Zur Evaluation der Replikationsstrategie RegRess wurde der Konfigurierbare, adaptive Replikationsmanagers KARMA implementiert, welcher in den Simulator F4SR eingebunden wurde. Der F4SR (Framework for Simulation of Replication Strategies [Fro06]) ist ein Simulator, der mittels diskreter, ereignisgesteuerter Simulation die Evaluation von Replikationsstrategien erlaubt. Der F4SR kann durch Anwendungsentwickler erweitert oder angepasst werden, z.B. indem neue Replikationsstrategien mittels Plugin-Mechanismus hinzugefügt werden. Das Simulationsmodell des F4SR bildet eine replizierte Datenbank ab und simuliert Schreib- und Lesezugriffe auf die Replikate und somit auf die Kompo-

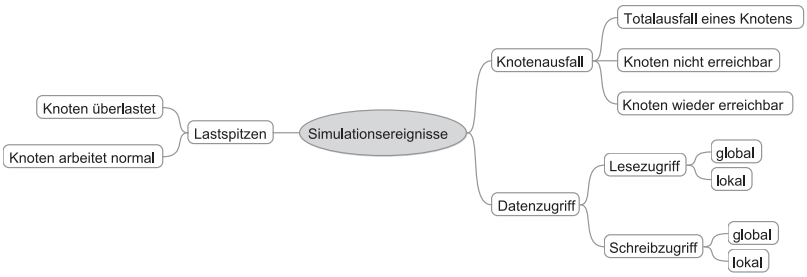

Abbildung 6: Simulationsereignisse des F4SR [Fro06]

nenten mit Replikat. Die Komponenten mit Replikat können sich in der Verarbeitungsgeschwindigkeit z.B. eines Schreibzugriffs unterscheiden. Im Simulationsmodell lassen sich auch Lastspitzen abbilden, die für Verzögerungen bei der Ausführung von Zugriffen verantwortlich sind. Des Weiteren können Komponenten mit Replikat ausfallen, d.h. eine Komponente mit Replikat ist entweder temporär oder dauerhaft während der Simulation nicht verfügbar. Dabei wird nicht zwischen Ausfall der Komponente und Netzwerkproblemen unterschieden.

Während der Simulation wird eine implementierte Replikationsstrategie verwendet, die vor Beginn der Simulation vom Analytiker ausgewählt wurde, wobei je nach Replikationsstrategie spezielle Einstellungen vom Analytiker konfiguriert werden. Beim Ablauf der Simulation werden vom F4SR alle Aktionen und deren Auswirkungen protokolliert, sodass am Ende der Simulation entsprechende Reports ausgegeben werden können. Hierfür wurden entsprechende Metriken definiert: Antwortzeit eines Datenzugriffs, Anzahl Schreib- bzw. Lesezugriffe, Anzahl veralteter Lesezugriffe, Anzahl erfolgreicher Schreib- bzw. Lesezugriffe, Anzahl abgebrochener Schreib- bzw. Lesezugriffe und Alter der Replikate.

Mögliche Simulationsereignisse, die während eines Simulationslaufs eintreten können, sind in der Abbildung 6 zu sehen. Die Simulationsereignisse betreffen entweder Lastspitzen, Knotenausfall (hier: Ausfall einer Komponente) oder Datenzugriff. Das Simulationsereignis Datenzugriff kann einen Schreib- oder Lesezugriff betreffen. Für diese Ereignisse wird einerseits das Schreib-/Leseverhältnis benötigt, andererseits die durchschnittliche Zugriffszeit eines solchen Datenzugriffs. Weiterhin wird zwischen globalen und lokalen Schreib- bzw. Lesezugriffen unterschieden. Lokale Zugriffe erfolgen auf die initiierende Komponente mit Replikat. Globale Zugriffe erfolgen auf andere Komponenten mit Replikat, je nach Replikationsstrategie im Allgemeinen auf eine Menge von Replikaten. Bei der Ausführung eines Zugriffs wird geprüft, ob die Komponente mit Replikat verfügbar ist und ob keine Sperre im Zuge einer Transaktionsverarbeitung eines anderen Zugriffs vorliegt. Für Details sei auf [Nie09] verwiesen.

5 Fazit

In heterogenen, autonomen Informationssystemen stellt die Integration von Daten eine große Anforderung dar, weil für ein Anwendungsszenario ein optimaler Kompromiss hinsichtlich der konkurrierenden Ziele Konsistenz, Verfügbarkeit und Performance erreicht werden muss. Wenn die Daten redundant in den lokalen Informationssystemen gespeichert werden, dann wird eine Replikationsstrategie benötigt, mit der der Trade-Off der Replikationsziele gelöst werden kann. Die in dieser Arbeit vorgestellte regelbasierte Replikationsstrategie RegRess bietet hierfür Lösungsmöglichkeiten, weil RegRess durch Vorgabe von Replikationsregeln konfiguriert wird und sich zur Laufzeit dem Systemverhalten anpasst, weil bei der Inferenz der Regeln Systemzustände berücksichtigt werden. Durch die Regeln können je nach Anwendungsszenario die Ziele gewichtet werden.

In dieser Arbeit wurde die Koordination von Schreib- und Lesezugriffen bei RegRess dadurch erläutert, dass die Reaktion auf die verschiedenen Inferenzarten beschrieben wurde. Eine Inferenz erfolgt auf eine vorgegebene Menge von Replikationsregeln, die in der hier vorgestellten Regelsprache RRML formuliert werden. Der in dieser Arbeit definierte Konsistenzgrad dient einerseits als Korrektheitskriterium für RegRess und andererseits als Metrik für die Konsistenz bei der Ermittlung des Trade-Offs zwischen Konsistenz, Verfügbarkeit und Performance. Des Weiteren wurde kurz die grobe Software-Architektur des Replikationsmanagers KARMA gezeigt, der RegRess implementiert. Die Evaluation erfolgte mittels des Simulators F4SR, mit dem Replikationsstrategien analysiert werden können.

RegRess bietet durch die Verwendung der RRML verschiedene Konfigurationsmöglichkeiten. Es lässt sich leicht zeigen, dass durch Regeln die meisten der so genannten asynchronen Replikationsstrategien nachgebildet werden können. Darüber hinaus können durch die RRML bei Schreib- und Lesezugriffen sowohl fachliche als auch technische Konsistenzbedingungen berücksichtigt. Weil Schreib-/Schreibkonflikte durch das Koordinationsprotokoll verhindert werden, ergibt sich im Falle der Verteilung des KARMA der Nachteil der Abhängigkeit von zentralen bzw. konsistent zu replizierenden Daten wie z.B. der Replica Queue. Daher ist eine mögliche Erweiterung von RegRess die Tolerierung von Schreib-/Schreibkonflikten, wobei dann eine geeignete Konfliktbehandlung benötigt wird.

Literatur

[ABGMA88] Rafael Alonso, Daniel Barbará, Hector Garcia-Molina und Soraya Abad. Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems. In *Conference on Extending Database Technology EDBT*, Jgg. 303 of *Lecture Notes in Computer Science*, Seiten 443–468. Springer, 1988.

[AL80] Michel E. Adiba und Bruce G. Lindsay. Database Snapshots. In *Sixth International Conference on Very Large Data Bases, VLDB*, Seiten 86–91. IEEE Computer Society, 1980.

[ASC85] Amr El Abbadi, Dale Skeen und Flaviu Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. In *Symposium on Principles of Database Systems PODS*, Seiten 215–229. ACM, 1985.

[CGM00] Junghoo Cho und Hector Garcia-Molina. Synchronizing a database to Improve Freshness. In Weidong Chen, Jeffrey F. Naughton und Philip A. Bernstein, Hrsg., *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Seiten 117–128, Dallas, Texas, USA, 2000.

[DGMD85] Susan Davidson, Hector Garcia-Molina und Skeen Dale. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[ES83] Derek L. Eager und Kenneth C. Sevcik. Achieving Robustness in Distributed Database Systems. *ACM Trans. Database Syst.*, 8(3):354–381, 1983.

[Fro06] Markus Fromme. *Framework zur Simulation und Evaluation von Replikationsstrategien*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fakultät II, Department für Informatik, Abteilung Software Engineering, 2006.

[GGL03] Sanjay Ghemawat, Howard Gobioff und Shun-Tak Leung. The Google file system. In Michael L. Scott und Larry L. Peterson, Hrsg., *SOSP*, Seiten 29–43. ACM, 2003.

[Has97] Wilhelm Hasselbring. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, 1(3):192–208, 1997.

[HR85] Frederick Hayes-Roth. Rule-Based Systems. *Commun. ACM*, 28(9):921–932, 1985.

[Len97] Richard Lenz. *Adaptive Datenreplikation in verteilten Systemen*. Teubner, Leipzig, 1997.

[Mos85] J. Eliot B. Moss. *Nested Transactions : An Approach to Reliable Distributed Computing (Digital Communication)*. The MIT Press, 1985.

[MTJ+08] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou und Ryan Peterson. Niobe: A practical replication protocol. *TOS*, 3(4), 2008.

[NHW+02] Heiko Niemann, Wilhelm Hasselbring, Thomas Wendt, Alfred Winter und Matthias Meierhofer. Kopplungsstrategien für Anwendungssysteme im Krankenhaus. *Wirtschaftsinformatik*, 44(5):425–434, 2002.

[Nie09] Heiko Niemann. *Regelbasierte Replikationsstrategie für heterogene, autonome Informationssysteme*. Software Engineering Research. Vieweg+Teubner Verlag, 2009.

[PG06] Cécile Le Pape und Stéphane Gançarski. Replica Refresh Strategies in a Database Cluster. In Michel J. Daydé, José M. L. M. Palma, Alvaro L. G. A. Coutinho, Esther Pacitti und João Correia Lopes, Hrsg., *VECPAR*, Jgg. 4395 of *Lecture Notes in Computer Science*, Seiten 679–691. Springer, 2006.

[PGV04] Cécile Le Pape, Stéphane Gançarski und Patrick Valduriez. Refresco: Improving Query Performance Through Freshness Control in a Database Cluster. In *CoopIS/DOA/ODBASE (1)*, Jgg. 3290 of *Lecture Notes in Computer Science*, Seiten 174–193. Springer, 2004.

[RBSS02] U. Röhm, K. Böhm, H.-J. Schek und H. Schuldt. FAS - A Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Very Large Data Bases Conference (VLDB)*, Seiten 754–765, Hong Kong, 2002.

[SS05] Yasushi Saito und Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[Sto79] Michael Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[TGGL82] I. L. Traiger, J. Gray, C. A. Galthier und B. G. Lindsay. Transactions and consistency in distributed database systems. *ACM Transactions on Database Systems*, 7(3):323–342, 1982.

[TPST98] Douglas B. Terry, Karin Petersen, Mike Spreitzer und Marvin Theimer. The Case for Non-transparent Replication: Examples from Bayou. *IEEE Data Eng. Bull.*, 21(4):12–20, 1998.

[vRS04] Robbert van Renesse und Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, Seiten 91–104, 2004.

[WC95] Jennifer Widom und Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc,US, 1995.

[YV02] Haifeng Yu und Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

[ZZ03] Chi Zhang und Zheng Zhang. Trading Replication Consistency for Performance and Availability: an Adaptive Approach. In *23rd International Conference on Distributed Computing Systems (ICDCS)*, Seiten 687–695. IEEE Computer Society, 2003.