Effiziente Reparatur von Repliken in Distributed Hash Tables

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Maik Lange
geboren am: 17.11.1984
in: Strausberg

Gutachter(innen): Prof. Dr. Alexander Reinefeld
Prof. Dr. Miroslaw Malek

eingereicht am: 27.09.2012 verteidigt am: .....

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Diplomarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 27.09.2012 .....

Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Berlin, 27.09.2012 .....

Zusammenfassung

Eine *distributed hash Table* (DHT) stellt eine über multiple Rechner strukturiert verteilte Tabelle von Schlüssel-Wert-Paaren dar. Ein verteiltes System unterliegt zumeist einer Teilnehmerdynamik in Form von ausfallenden oder beitretenden Knoten, wodurch Daten verloren gehen können. Zur Sicherstellung der Verfügbarkeit aller Schlüssel-Wert-Paare ist Redundanz notwendig.

Die Einführung einer Redundanzschicht erhöht die Verfügbarkeit der Daten erheblich auf Kosten einer verringerten Leistungsfähigkeit. Der Fokus dieser Arbeit liegt im Erhalt der Redundanz, was die Regeneration verlorener Kopien sowie eine Aktualisierung veralteter Kopien umfasst. Die Regeneration verlorener Kopien ist notwendig um die Verfügbarkeit langfristig zu sichern. Die Aktualisierung ist optional und verbessert die Leselatenz in geografisch weit verteilten Systemen.

Die vorgestellte Lösung besteht aus drei Komponenten. Der *Set Reconciliation* Dienst ermöglicht den effizienten Vergleich zweier Knoten zur Identifikation verlorener und veralteter Kopien. Der Aktualisierungsdienst synchronisiert die verglichenen Knoten durch die Übertragung der Differenzen. Die dritte Komponente ist ein *Anti-Entropie* Protokoll, dass die Auswahl eines jeden Knotens als Synchronisationspartner sicherstellt. Das Ziel dieser Arbeit liegt in dem Vergleich dreier *Set Reconciliation* Algorithmen, deren Eignung und Leistungsfähigkeit für die Erkennung verlorener und veralteter Kopien evaluiert werden soll. Ausgewählt wurden der *Bloom Filter*, der *Merkle-Tree* sowie deren Kombination der *Approximate Reconciliation Tree*.

Die Evaluation zeigte, dass der *Merkle-Tree* die höchste Genauigkeit besitzt und spätestens nach einer initiierten Synchronisation eines jeden Knotens zur vollständigen Synchronität in der DHT führt. Der *Bloom Filter* besitzt bei der Erkennung verlorener Repliken nur die halbe Genauigkeit gegenüber veralteten Repliken, wodurch die Konvergenzgeschwindigkeit dementsprechend geringer ist gegenüber dem *Merkle-Tree*. Dafür sind die Übertragungskosten des *Bloom Filter* um bis zu 80% geringer als beim *Merkle-Tree* und weisen kaum nennenswerte Spitzen auf. Die Übertragungskosten des *Merkle-Tree* sind abhängig von der Anzahl der Differenzen, sodass bei einer Differenz von Null nur eine Signatur mit weniger als einem kByte übertragen werden muss. Ist die Differenz größer, steigen die Kosten je nach *Bucket*-Größe an.

Der *Approximate Reconciliation Tree* ist ein als *Bloom Filter* codierter *Merkle-Tree*, der je nach Parameterkonstellation die Eigenschaften einer der beiden Strukturen aufweist. Die *Bloom Filter* ähnlichen Konfigurationen bieten keinen Vorteil, wohingegen die *Merkle-Tree* ähnlichen nur zwei Mitteilungsrunden benötigen bei einer leichten Genauigkeitsverringerung gegenüber dem *Merkle-Tree*.

Inhaltsverzeichnis

1 Einführung

Die Verwaltung großer Datenmengen unter stetig steigenden Nutzerzahlen hat den Bedarf an skalierbaren Computersystemen forciert und so zur Entwicklung von verteilten Systemen beigetragen. Ein verteiltes System aus gleichberechtigten Rechnern (Knoten) wird als *Peer-to-Peer*-System (P2P) bezeichnet und bietet vor allem Hochverfügbarkeit und Skalierbarkeit. Hochverfügbarkeit bezeichnet die Fähigkeit einen ununterbrochenen Betrieb zu gewährleisten, trotz steigender Last oder Knotenausfällen. Die Skalierbarkeit steht für die Anpassungsfähigkeit eines Systems indem die Leistungsfähigkeit proportional von den Ressourcen abhängt.

Eine effiziente Umsetzung eines P2P-basierten Datenspeichers stellt die *distributed hash table* (DHT) dar. Eine DHT stellt eine verteilte Tabelle von Schlüssel-Wert-Paaren dar, in der jeder eindeutige Schlüssel genau einem Knoten zugewiesen ist.

Im Allgemeinen wird von der konkreten Anwendung abstrahiert und von einem verteilten Datenspeicher gesprochen, der entweder Änderungen zulassen oder verwehren kann. DHTs die Änderungen bereits gespeicherter Einträge zulassen, werden *mutable* DHTs genannt. Jene die keine Änderungen zulassen, werden *immutable* genannt. Beispiele für *mutable* DHTs sind Scalaris [83], Amazons Dynamo [28] oder Oceanstore [51]. Eine *immutable* DHT ist PAST [32], in welcher der Schlüssel über die Signatur des Dateiinhaltes erstellt wird und damit eineindeutig ist.

Redundanz beschreibt die Vervielfältigung von Schlüssel-Wert-Paaren, sodass bei einem Redundanzfaktor von beispielsweise vier, vier identische Objekte eines jeden Schlüssel-Wert-Paares im System existieren. Mit einem Redundanzfaktor von eins ist jedes Schlüssel-Wert-Paar nur einmal vorhanden. Damit ein verteiltes System Verfügbarkeit und Leistungsfähigkeit gewährleisten kann, ist Redundanz zwingend erforderlich. Die Einführung von Redundanz in ein verteiltes System führt zu einer Reihe von Herausforderungen, die die Eigenschaften des Systems stark beeinflussen. Es muss entschieden werden, wo Kopien im System plaziert werden, wie auf diese zugegriffen werden soll, *et cetera*. Diese und weitere Herausforderungen stellen damit Designentscheidungen dar, deren Lösungen in Ihrer Gesamtheit als Redundanzschicht bezeichnet werden.

Verteilte Systeme sind sehr dynamisch und unterliegen einem ständigen Wechsel ihrer partizipierenden Knoten, was als *churn* bezeichnet wird. *Churn* kann zu Redundanzverlust führen, weshalb jedes verteilte System Mechanismen zur Redundanzerhaltung benötigt.

1.1 Problembeschreibung

Die langfristige Sicherstellung der Verfügbarkeit von Schlüssel-Wert-Paaren in *mutable* und *immutable* DHTs erfordert einen Regenerationsmechanismus, der verloren gegangene Daten wiederherstellt. Daten gehen in verteilten Systemen zumeist durch den Ausfall von Knoten verloren. Der Prozess der Regeneration beinhaltet die Erkennung verlorener Kopien sowie die Erstellung neuer Kopien im System. Ein Regenerationsmechanismus ist essenziell, da sonst aufgrund von *churn* die Anzahl der Kopien langfristig gegen Null geht.

Ein weiteres Problem, das nur bei *mutable* DHTs auftreten kann, sind veraltete Kopien. Diese kommen nur bei bestimmten Redundanzzugriffsformen vor, die in dem Grundlagenkapitel 2.2.4 detailliert erläutert werden. Im Allgemeinen existieren zwei Klassen von Zugriffsformen: In der ersten gilt ein neuer Wert als abgespeichert, wenn alle Kopien diesen übernommen haben. In der zweiten muss nur eine Anzahl n kleiner als der Redundanzgrad r den neuen Wert übernehmen, damit dieser als gespeichert gilt. Dies kann zur Entstehung von veralteten Kopien, in der Höhe der Differenz $r - n$, führen.

Veraltete Kopien senken die Datensicherheit und folglich auch die Verfügbarkeit. In Systemen mit großen geographischen Distanzen zwischen den einzelnen Knoten können veraltete Kopien die Zugriffslatenz beim Lesen beeinflussen. Eine geographisch nahe, jedoch veraltete Kopie kann den Wert nicht an den Nutzer ausliefern, wodurch ein weiter entfernter Knoten die Daten senden muss. Die Aktualisierung veralteter Kopien ist daher gerade in geographisch weit verteilten Systemen vorteilhaft.

Algorithmen für die Probleme der Regeneration sowie der Replikaktualisierung bestehen aus zwei Phasen. In Phase I müssen jene Knotem im System identifiziert werden, denen potenziell Kopien fehlen bzw. die veraltete Versionen besitzen. In der Phase II werden die Differenzen in den Datenbeständen identifiziert und anschließend aufgelößt, indem neue Kopien erstellt werden bzw. veraltete aktualisiet werden.

Algorithmen für die Regenerierung und Aktualisierung von Repliken, die auf der Schlüsselebene den Redundanzgrad bzw. die Versionskonsistenz überwachen und entsprechend korrigieren, sind flexibel, skalieren jedoch schlecht bei großen Datenmengen. Daraus ergibt sich vor allem für die Phase II der Bedarf an einem Algorithmus, der effizient die Datenbestände zweier Knoten vergleicht und dessen Differenzen aufzeigt. Dieser Abgleich zweier Mengen wird in der Literatur [66, 87, 19, 18, 34] als *Set Reconciliation* bezeichnet und stellt den Fokus dieser Arbeit dar.

Dabei wird sich auf kontextlose Lösungen beschränkt. Eine übliche kontextabhängige Art, zwei Mengen abzugleichen, ist die Änderungen der eigenen Datenbank zu protokollieren und den Zeitpunkt der letzten Kommunikation mit einem anderen Knoten abzuspeichern. Das Protokoll der Datenbankänderungen wird im weiteren Log genannt. Bei der Log-basierten Synchronisation werden mit dem Partnerknoten nur jene Protokolleinträge abgeglichen, die seit der letzten Kommunikation hinzugefügt bzw. geändert wurden [88].

Eine solche kontextabhängige Lösung für das *Set Reconciliation* Problem verursacht eine Vielzahl von Problemen, die die Leistungsfähigkeit des Systems beeinträchtigen. Es muss auf jedem Knoten ein Kontext-Objekt für jeden Synchronisationspartner angelegt werden, in dem das Log gespeichert wird sowie der Zeitpunkt der letzten Synchronisation. Die Anzahl der Kontext-Objekte wächst mindestens linear mit der Knotenanzahl und der verbrauchte Speicherplatz je Kontext-Objekt hängt linear von der Änderungshäufigkeit von Schlüssel-Wert-Paaren ab. Die Kosten von Schreib-Operationen werden erhöht, weil bei jeder Datenänderung ein Log-Eintrag pro replikhaltenden Knoten angelegt werden muss. Die Speicherung des letzten Synchronisationszeitpunkts erfordert eine lose synchronisierte Systemzeit, die eine Toleranz von d garantiert, sodass die Zeit keines Knotens stärker als d von der eines anderem Abweicht. Eine solche Systemzeit kann über das *Network Time Protocol* [63] oder in Form einer logischen Uhr (Lamport Uhr) [53] umgesetzt werden, wodurch weitere Kosten entstehen. Ein weiterer Nachteil ist die Ansammlung nicht mehr benötigter Kontext-Objekte, die durch ausfallende Knoten entstehen. Diese können durch einen *garbage collector* entfernt werden, welcher lokale Kosten in Form von Speicherzugriffen und Prozessorlast verursacht. Letztlich verursacht die Log-Lösung dauerhafte Mehrkosten auch wenn keine *Set Reconciliation* durchgeführt wird.

1.2 Ziele

Ziel dieser Arbeit ist die Implementation und Evaluation eines Algorithmus zur Regeneration sowie Aktualisierung von Repliken. Für die Phase I wird ein epidemischer Algorithmus verwendet, der die Selektion des Synchronisationspartners vornimmt. Die Identifikation von Differenzen zweier Datenbestände wird in Phase II mit Hilfe von verschiedenen *Set Reconciliation* Algorithmen durchgeführt. Drei der im Kapitel 2.4 vorgestellten *Set Reconciliation* Algorithmen, Bloom Filter, Merkle Tree und ART werden implementiert und in der Evaluation 4 analysiert und verglichen.

1.3 Methoden

Die Implementation aller Algorithmen erfolgt in der Programmiersprache Erlang[1] und wird in das Scalaris-Projekt[2] integriert. Die Evaluation besteht aus zwei Teilen. Im ersten Teil werden die *Set Reconciliation* Algorithmen in den Eigenschaften Genauigkeit, Übertragungskosten und Latenz in ihren jeweiligen Parametern optimiert und in verschiedenen Szenarien gegenübergestellt. In den Szenarien wird insbesondere die Auswirkung unterschiedlicher Fehler- und Datenverteilungen auf die Algorithmen betrachtet.

Im zweiten Teil der Evaluation wird die Konvergenz sowie die Konvergenzgeschwindigkeit der

1 `erlang.org`
2 `code.google.com/p/scalaris`

Lösung analysiert. Dies wird in Scalaris mit unterschiedlichen Verteilungen der Daten, Fehler und Knoten durchgeführt.

1.4 Aufbau

Das Kapitel 2 beginnt mit einer näheren Beschreibung des Aufbaus einer DHT. Darauf folgt eine detaillierte Übersicht über die Architekturmöglichkeiten einer DHT-Redundanzschicht in 2.2. Nach der Beschreibung von Scalaris in 2.3 wird eine Auswahl der am häufigsten verwendeten *Set Reconciliation* Algorithmen vorgestellt, die die Grundlage für einen effizienten Vergleich großer Datenmengen bilden. Anschließend wird in 2.5 das Regenerationsproblem und in 2.6 das Aktualisierungsproblem näher betrachtet und jeweils existierende Lösungen vorgestellt.

Im Kapitel 3 wird das Systemmodell und die Architektur des Repliken-Reparatur-Service beschrieben, mit einer genauen Erläuterung der Implementation der ausgewählten *Set Reconciliation* Algorithmen. Darauf folgt die Evaluation in Kapitel 4, in der nach der Vorstellung der Szenarien und Metriken die Ergebnisse dargestellt werden. Die Arbeit endet im Kapitel 5 mit einer Zusammenfassung und einem Ausblick auf mögliche Verbesserungen und Erweiterungen.

2 Grundlagen

2.1 Distributed Hash Tables (DHT)

Eine DHT ist ein verteilter Datenspeicher, deren Grundlage ein P2P-Netzwerk darstellt. Rechner in einem solchen Netzwerk werden als *Peer* oder Knoten bezeichnet und sind Server und Client zugleich. Der Zusammenschluss gleichartiger Ressourcen ermöglicht die Erstellung einer größeren dezentralen Ressource, die nicht von einzelnen Peers abhängt. P2P-Systeme zeichnen sich vor allem durch Skalierbarkeit und Verfügbarkeit aus.

Ein zentrales Problem von P2P-Systemen ist das effiziente dezentrale Auffinden von Daten, welches als *lookup problem* bezeichnet wird [5]. Dies ergibt sich bei einem verteilten System S mit n Knoten $K = k_1, \ldots, k_n$ wie folgt. Ein Nutzer speichert eine Datei f in S ab, indem f auf einem Knoten k_i abgelegt wird. Ein weiterer Nutzer möchte die Datei f aus S laden, wozu der Knoten k_i bekannt sein muss. Das effiziente Auffinden von k_i unter n Knoten stellt das *lookup problem* dar.

Es existieren verschiedene Lösungsansätze für das *lookup problem*. Napster[1] verwendet eine zentrale Datenbank, die Dateinamen auf Knotenadressen abbildet [72]. Eine solche zentrale Datenbank stellt ein sogenanntes *bottleneck* dar und wird in aktuellen P2P-Systemen nur noch selten verwendet. *Bottlenecks* verursachen eine Konzentration der Systemlast auf wenigen Knoten, wodurch die Leistungsfähigkeit des Gesamtsystems beeinträchtigt wird. Gnutella [49] und Freenet [26] verwenden *broadcasts* zur Suche, sodass alle n Knoten nach f befragt werden. Die bisher effizienteste Lösung des Problems stellen *distributed hash tables* (DHTs) dar.

Eine DHT ist eine Tabelle von Schlüssel-Wert-Paaren, die über die Knoten des Systems verteilt vorliegt. Mit Hilfe eines *structured overlay network* (SON) erfolgt eine strukturierte Zuweisung von Schlüsselbereichen der DHT auf die Knoten, wodurch das *lookup problem* effizient gelöst werden kann. Beispiele für SONs sind Chord [89], Chord$^{\#}$[82], Pastry [80] oder DKS [39]. Chord ermöglicht das Auffinden eines Wertes zu einem Schlüssel in $O(\log N)$ Kommunikationsschritten unabhängig von der Knotenanzahl, wobei N die maximale Schüsselanzahl der DHT ist.

Nach Brewer's CAP-Theorem [15, 60] ist es für einen *web service* sowie für eine DHT [39] nicht möglich, in einem asynchronen Netzwerk, die drei Eigenschaften strenge Konsistenz (C), Verfügbarkeit (A) und *partition-tolerance* (P) zu besitzen. Die strenge Konsistenz garantiert, dass immer der zuletzt geschriebene Wert gelesen wird. Die Verfügbarkeit garantiert, dass zu jedem

1 `http://napster.com`

Zeitpunkt der Zugriff auf ein Datum möglich ist. Es kann bewiesen werden, dass in einem verteilten System aus zwei Knoten, bei Abbruch der Netzwerkverbindung, entweder die Konsistenz verloren geht, falls einer der beiden eine Anfrage beantwortet oder die Verfügbarkeit verloren geht falls keiner die Anfrage beantwortet [92]. Dieser Zusammenhang verdeutlicht die *partition-tolerance*, die demnach eine Toleranz gegenüber einem permanenten Nachrichtenverlust zwischen einer Partition zur anderen darstellt [60].

2.2 Redundanz in DHTs

Dauerhaftigkeit (*durability*) und Verfügbarkeit (*availability*) von Daten können in verteilten Systemen nur mit Hilfe von Redundanz sichergestellt werden. Der Begriff Redundanz stammt vom lateinischen *redundantia*[1] ab und beschreibt, dass etwas im Überfluss vorhanden ist. In der Technik bezeichnet Redundanz das mehrfache Vorhandensein gleichwertiger Ressourcen sowie in der Informationstheorie der Anteil einer Nachricht, der ohne Informationsverlust weggelassen werden könnte. DHTs können Informationen nur durch Redundanz dauerhaft speichern, weil durch *churn* jederzeit Knoten aus dem System ausscheiden können. Redundante Daten werden im Folgenden entweder als Kopie oder als Replik bezeichnet.

Komplexe Softwaresysteme werden zumeist in einer Multi-Schicht-Architektur beschrieben, in der jede Schicht eine Funktion kapselt. Daher wird im Folgenden von einer Redundanzschicht gesprochen, die die Verwaltung und das Design der Redundanz in einer DHT beschreibt. Die Architektur einer Redundanzschicht erfordert verschiedene Design-Entscheidungen, die die Eigenschaften der Redundanzschicht prägen und diese auf unterschiedliche Umgebungen optimiert.

Im Folgenden werden die wichtigsten Design-Elemente erläutert. Es wird mit den Formen der Redundanz (2.2.1) begonnen, darauf folgt die Platzierung (2.2.2) und der Zugriff (2.2.4) auf die redundanten Daten. Der Zugriff auf redundant gespeicherte Daten erfordert die Wahl eines Konsistenzmodells (2.2.3), das die Leistungsfähigkeit stark beeinflussen kann. Im Abschnitt 2.2.5 werden Aufrechterhaltungsstrategien erläutert, die dem Redundanzverlust entgegenwirken.

2.2.1 Formen

Die Redundanzform definiert die Art der replizierten Daten. Im Allgemeinen wird zwischen Dateireplikation (*mirroring*) und Blockreplikation (*block-level replication*) unterschieden.

Mirroring beschreibt die Erstellung exakter Kopien der Originaldaten. Der Redundanzgrad r beschreibt die Anzahl der Kopien des Originals. Ist ein Zielverfügbarkeitsgrad A sowie die mittlere Knotenverfügbarkeit μ gegeben, kann die benötigte Zahl an Kopien r, die zur Erreichung von A nötig ist, nach Formel 2.1 berechnet werden. Ist die mittlere Knotenverfügbarkeit μ über r

1 lt. `de.wiktionary.org/wiki/Redundanz`, besucht 10.05.2012

unabhängige Knoten gegeben, kann der Verfügbarkeitsgrad einer Datei f nach der Formel 2.2 aus [56] berechnet werden.

$$r = \frac{\log(1-A)}{\log(1-\mu)} \tag{2.1}$$

$$A_f^{MIR}(r) = \sum_{i=1}^{r} \binom{r}{i} \mu^i (1-\mu)^{r-1} \tag{2.2}$$

Der Verfügbarkeitsgrad eines replizierten Objekts wird im Allgemeinen als „Anzahl von Neunen“ angegeben, da $A_f^{MIR}(r) \in [0{,}1[$. Ein Objekt mit 4 Kopien $r = 4$ und einer durchschnittlichen Knotenverfügbarkeit von $\mu = 0{,}8$ am Tag besitzt einen Verfügbarkeitsgrad von $A = 0{,}9984$ bzw. zwei Neunen, was bedeutet, dass das Objekt weniger als zwei Minuten am Tag nicht verfügbar ist.

Die BLOCK-LEVEL REPLICATION steigert die Verfügbarkeit durch die Partitionierung eines zu Datums f in b Blöcke, die r-fach repliziert werden [10]. Die damit erzeugten $r * b$ Blöcke werden auf separaten Knoten verteilt abgelegt. Die Rekonstruktion von f erfordert mindestens eine Kopie eines jeden Blockes. Mit dieser Methode kann der Verlust von $r - 1$ Kopien jedes Blockes b toleriert werden.

Vergleicht man die Verfügbarkeit von *block-level replication* mit dem *mirroring* unter Verwendung des doppelten Speicherplatzes, so existieren beim *mirroring* zwei Kopien der Originaldaten f auf zwei Knoten. Das *mirroring* kann in diesem Fall den Ausfall eines Knotens tolerieren. Bei der *block-level replication* wird f in $b = 3$ Blöcke geteilt die mit $r = 2$ zweimal im System abgelegt werden. Dies entspricht dem doppelten Speicherverbrauch von f, wie beim *mirroring*. Im besten Fall können so drei Knotenausfälle toleriert werden, wenn jeweils eine Kopie jedes Blockes b verloren geht. Dies zeigt eine effizientere Speichernutzung der *block-level replication*. Das *mirroring* kann als Sonderfall der *block-level replication* mit $b = 1$ betrachtet werden.

Die *block-level replication* wird fast ausschließlich mit *erasure codes* verwendet. ERASURE CODES gehören zu den *error-correcting codes* und generieren $r * b$ voneinander abhängige Blöcke, im weiteren Fragmente genannt. Durch *erasure codes* ist es möglich, eine in b Blöcke geteilte Datei aus beliebigen b der $r * b$ Fragmente zu rekonstruieren. Musste beispielsweise bei der *block-level replication* ohne *erasure codes* mindestens ein Knoten je Block bei $r = 2$ verfügbar sein, so reicht es bei Verwendung der *erasure codes*, wenn beliebige b der rb Fragmente verfügbar sind. Die Verfügbarkeit eines mit *erasure codes* replizierten Objektes f ergibt sich aus Formel 2.3 [56].

$$A_f^{EC}(r,b) = \sum_{i=b}^{rb} \binom{rb}{i} \mu^i (1-\mu)^{rb-1} \tag{2.3}$$

Bei dem vierfachen Speicherplatzverbrauch einer Datei f kann mit *erasure codes* eine Verfügbarkeit von $A_f^{EC}(4{,}2) = 0{,}999915$ erreicht werden, mit $\mu = 0{,}8$. Damit erreichen *erasure codes* vier Neunen

im Gegensatz zum *mirroring*, das mit vier Kopien nur zwei Neunen erreichte.

Erasure codes sind eine seit langem bekannte Technik, die wesentlich bessere Verfügbarkeit bei geringerem Speicherplatzverbrauch bietet als *mirroring* [73]. Bei *erasure codes* müssen b Knoten nach Fragmenten befragt werden um die Originaldaten rekonstruieren zu können, wodurch solche Systeme ein höheres Nachrichtenaufkommen besitzen als *mirroring*-basierte Systeme [97]. Beim *mirroring* kann ein Knoten eine Leseanfrage vollständig beantworten wodurch die Latenz häufig geringer ist als bei *erasure codes*. Ein weiterer Nachteil von *erasure codes* ist deren Berechnungsaufwand, welcher bei Rekonstruktion und Kodierung anfällt.

Die Art des verwendeten *erasure codes* beeinflusst die Rekonstruktionslatenz maßgeblich, die bei größeren Datenvolumen beträchtlich sein kann. Luby zeigt in einem Benchmark [57] das "neuere" Ansätze wie *Tornado codes* [59, 58] bei kleineren Nachrichten bis zu 100 mal und bei größeren bis zu 10000 mal schneller sind als *Cauchy-based Reed-Solomon codes* [12]. Weitere Vergleiche und detailliertere Informationen zu den beiden Redundanzformen können in [97, 79, 56] gefunden werden.

2.2.2 Platzierung von Repliken

Platzierungsstrategien geben den Ort im System an, an denen die Repliken eines Datensatzes abgelegt werden. Im Allgemeinen wird zwischen *random placement*, dem zufälligen Platzieren und dem *selective placement*, dem selektiven Platzieren unterschieden.

Das RANDOM PLACEMENT bezeichnet das zufällige Auswählen eines Knotens für ein zu sicherndes Objekt. Die Adressen der Kopien werden normalerweise in Meta-Daten gesichert, um diese effektiv wiederzufinden, wie in [9, 25, 38, 6, 27]. Durch *churn* müssen die Meta-Daten aktualisiert werden, wodurch Verwaltungskosten entstehen, die einen Nachteil dieser Platzierungsstrategie darstellen. Dafür sind Verschiebungen von Kopien bei Knotenbeitritten nicht notwendig, da die Kopieposition knotengebunden gesichert ist, wohingegen selektive Platzierungsstrategien meist schlüsselgebunden sind.

Das SELECTIVE PLACEMENT versucht die Objekte mit Hilfe einer Strategie so zu platzieren, dass die Orte der Objekte zu einem Datum aufgrund der Strategie immer bekannt sind, ohne zusätzliche Informationen speichern zu müssen. Geläufige selektive Platzierungsstrategien sind, das *successor placement*, in dem die Nachfolger, eines für ein Datum zuständigen Knotens, Kopien besitzen, das *predecessor placement*, in dem die Vorgänger des zuständigen Knotens Kopien des Datums besitzen oder das *symmetric placement* [40], in dem f Schlüssel durch eine Funktion symmetrisch miteinander assoziiert werden und jeweils die gleiche Kopie besitzen.

Der wesentliche Vorteil von selektiven Strategien ist, dass die Speicherorte aus der Strategie ableitbar sind und nicht gespeichert werden müssen. Dies wird größtenteils auf Basis von Schlüsseln durchgeführt, wodurch bei Knotenbeitritten Kopien verschoben werden müssen.

2.2.3 Konsistenz

Die Konsistenz beschreibt die Korrektheit und Widerspruchsfreiheit von Daten in einer Datenbank, die bei einer Sequenz von Operationen erhalten bleiben soll. Die Sicherstellung der Konsistenz bei parallelem Zugriff ist schwierig und erfordert spezielle Zugriffsmethoden, die im Abschnitt 2.2.4 erläutert werden. Es existiert eine Vielzahl unterschiedlicher Konsistenzarten bzw. Konsistenzmodelle, wobei in der Praxis nur die strenge sowie die schwache Konsistenz relevant sind. Ein Konsistenzmodell schränkt dabei die Werte ein, die bei einer Lese-Operation zurückgegeben werden dürfen. Dabei gilt je beschränkter die Rückgabe, desto strenger die Konsistenz und desto aufwendiger ist die Sicherstellung dieser bei parallelen Zugriffen.

Die strenge Konsistenz (*strong consistency*) ist die stärkste und aufwendigste Art der Konsistenz. Diese garantiert, dass jede Leseoperation für ein Datum x den zuletzt geschrieben Wert wiedergibt. Damit existiert eine vollständige sequenzielle Abfolge des Zugriffs auf x und die verschiedenen Kopien von x agieren vollständig transparent, wie eine einzelne Kopie. Diese Art der Konsistenz wird auch *one-copy equivalence* genannt und ist essenziell für transaktionale Datenbanken, ohne die keine Buchungssysteme realisierbar wären.

Die schwache Konsistenz (*weak consistency*) stellt sicher, dass alle Prozesse, die auf ein Datum x zugreifen, die gleiche Abfolge von "Änderungen" sehen. Finden demnach zwei Schreibvorgänge statt $w_1(x) = a, w_2(x) = b$, sehen alle anderen Prozesse die gleiche Abfolge von Werten, jedoch nicht zwingend den letzten. So kann zeitgleich Prozess P_1 Wert a von der Speicherzelle x lesen und ein Prozess P_2 den Wert b. Bei nachfolgenden Leseoperationen von P_1 wird eventuell einmal b gelesen, P_2 ließt jedoch nie den Wert a, da er schon b gelesen hat. Diese Form der Konsistenz garantiert, dass irgendwann alle Kopien den gleichen, zuletzt geschriebenen, Wert besitzen. Leseoperationen in diesem Modell sind dafür nicht deterministisch, da bei jedem Lesen entweder ein alter oder ein neuer Wert zurückgegeben werden kann. In verteilten Systemen kann diese Konsistenzart einfacher und leistungsfähiger implementiert werden als die strenge Konsistenz. Dafür ist die schwache Konsistenz nicht für alle Anwendungsfälle geeignet, jedoch für sehr viele, wie dem Web-Caching oder *dynamic name system*.

2.2.4 Zugriff auf replizierte Daten

Die Zugriffsart bzw. Zugriffsstrategie auf redundant gespeicherte Daten entscheidet über die Art der Datenkonsistenz sowie über die allgemeine Lese-/Schreibgeschwindigkeit. Als Zugriffsstrategien existieren die synchrone, asynchrone und hybride Replikation.

Bei der ASYNCHRONEN REPLIKATION wird immer auf eine Primärkopie zugegriffen, deren Antwort eine Anfrage vollständig beantwortet. Eine Schreibanfrage gilt als beendet, wenn die Primärkopie den neuen Wert übernommen hat. Alle anderen Kopien werden durch die Primärkopie mit Hilfe eines *replica-update*-Protokolls verzögert aktualisiert. Bekannte asynchrone Replikationsschemen sind *primary-backup replication* und *multi-primary-backup replication*. Der Hauptvorteil der asynchronen

Replikation ist die geringe Antwortlatenz, weil Anfragen nur durch die Primärkopien beantwortet werden müssen. Primärkopien stellen dafür ein *bottleneck* dar, das die Skalierbarkeit einschränkt sowie zur Dateninkonsistenz oder zu Änderungsverlust führen kann.

Bei der SYNCHRONEN REPLIKATION muss auf alle Kopien zugegriffen werden um eine Anfrage zu beantworten. Demnach ist eine Schreib-Anfrage erst beendet, wenn alle Kopien den neuen Wert übernommen haben. Diese Strategie ist auch als *active replication* bzw. *state machine approach* in der Literatur bekannt [54]. Die synchrone Replikation ermöglicht einen streng konsistenten Zugriff auf replizierte Daten sowie eine gleichmäßige Lastverteilung und eine bessere Skalierbarkeit gegebnüber asynchronen Zugriffsschemen. Einen Nachteil stellt die erhöhte Zugriffslatenz dar, die durch den Kontakt aller Kopien je Zugriff verursacht wird.

Quorumbasierte Zugriffsprotokolle ermöglichen die Umsetzung der synchronen Replikation ohne auf alle Kopien bei einer Anfrage zuzugreifen. Die Art des Quorums entscheidet, wie viele Repliken beteiligt sein müssen, um eine Anfrage zu beantworten. Es wird zwischen strikten Quoren (*strict quorum*) und schwachen Quoren (*weak quorum*) unterschieden. Erstere bieten eine strenge Datenkonsistenz, da sie eine Überlappung aufeinanderfolgender Quoren garantieren, während letztere dies nur mit hoher Wahrscheinlichkeit garantieren und somit eine Wahrscheinlichkeit der Konsistenzverletzung beinhalten.

In verteilten Systemen schien es sicher, dass nur der Quorum-basierte Zugriff strenge Konsistenz sicherstellen kann. Es existieren jedoch auch zwei Besondere Algorithmen, die strenge Konsistenz ohne Quorum sicherstellen können. Dies sind zum einen der ABD-Algorithmus [4] und zum anderen *Weak snapshots* [3].

Strikte Quorumprotokolle unterscheiden sich vor allem in der Größe der Lese- bzw. Schreibquoren. Durch Verschiebungen der Lese- bzw. Schreibquorumgröße kann ein System auf verschiedene Zugrifssmuster optimiert werden. Im Allgemeinen bieten kleine Quoren schnellere Zugriffszeiten als größere. Das ROWA-Protokoll (*Read-One-Write-All*) [7] besitzt ein Lese-Quorum von eins und ein Schreibquorum von allen Repliken. Damit ist dieses Protokoll auf Leseoperationen optimiert, die wesentlich schneller verarbeitet werden können als Schreibvorgänge.

Die *chain replication* [75] gehört auch zu den strikten Quoren und organisiert die Kopien in einer Kette, deren Kopf die Schreibzugriffe verarbeitet und deren Ende die Leseanfragen beantwortet. Der Neue Wert wird über den Kopf entlang der Kette bis ans Ende propagiert. Durch diese Art des Zugriffs wird die Organisation der Kopien vereinfacht und für die Tolerierung von f Ausfällen werden nur $f + 1$ Kopien benötigt. Erst wenn alle Kettenmitglieder eine vom Kopf propagierte Aktualisierung übernommen haben, ist ein Schreibvorgang beendet, wodurch das Protokoll strenge Konsistenz bietet. In diesem Fall verhält sich die *chain replication* wie das ROWA System. Es ist jedoch auch möglich Schreib- und Lesequoren der Größe eins festzulegen, wodurch das Protokoll nur noch *weak consistency* garantiert.

Bei einem Redundanzgrad von r Kopien je Datum sind folgende weitere strikte Quorum Protokolle umsetzbar. Das *Tree Quorum* (TQP) [2] und das *Hierarchical Quorum* (HQP) [52] verwenden

Protokoll	Minimale Quorumgröße	Maximale Quorumgröße	Konsistenzart streng, schwach
ROWA [7]	1	r	streng
ROWAA [8]	1	r	schwach
MQP [93]	$\frac{r+1}{2}$	$\frac{r+1}{2}$	streng
TQP [2]	$\log(r)$	$\frac{r+1}{2}$	streng
HQP [52]	$r^{0,63}$	$r^{0,63}$	streng
CQP [46]	1	r	streng
GQP [23]	$\sqrt{r}$	$\sqrt{r}$	streng

Tabelle 2.1: Übersicht über die Quorumgrößengrenzen der einzelnen Quorum Protokolle bei einem Redundanzgrad r, modifiziert nach [46]

Baumstrukturen um die Quorengrößen klein zu halten. Das *Majority Quorum* (MQP) [93] verwendet die Mehrheit von $\frac{r+1}{2}$ Kopien als Quorum und unterscheidet nicht in Lese- und Schreibquoren. Das *Column Quorum* (CQP) [46] basiert auf einer tabellarischen Anordnung der Quoren, wobei die Zeilenanzahl l je Spalte c unterschiedlich sein darf. Das Lesequorum besteht aus einer vollständigen Spalte, das Schreibquorum aus mindestens einem Eintrag je Spalte *vice versa*. Ein Spezialfall des CQP ist das *Grid Quorum* (GQP) [23], welches eine quadratische Tabelle darstellt. Das ROWA-Protokoll kann auch mit dem CQP gebildet werden, wenn $c = r$ und jede Spalte nur eine Zeile besitzt. Aufgrund der frei wählbaren Spalten sowie Zeilenanzahl ist eine sehr fein granulare Lastoptimierung möglich.

Schwache Quorumprotokolle werden auch *probabilistic quorum systems* (PQS) genannt, da sie nur mit einer hohen Wahrscheinlichkeit eine Überschneidung der Quoren garantieren. Der Vorteil dieser Lockerung ist eine erhebliche Aufwandsverringerung für die Bildung eines Quorums. Beispiele für PQS-Protokolle sind das *Witness Quorum* (WQP) [99] und *Signed Quorum System* (SQS) [100].

Es existieren auch HYBRIDE ZUGRIFFSSCHEMEN, die versuchen die Performanz der asynchronen Replikation mit der strengen Konsistenz der synchronen Replikation zu verbinden. Ein hybrides Protokoll ist das ROWAA-Protokoll (*Read-One-Write-All-Available*) [8] in dem alle verfügbaren Kopien synchron aktualisiert werden und die verbleibenden asynchron, sobald sie wieder verfügbar werden. In diesem Fall gilt die strenge Datenkonsistenz nur, wenn keine Netzwerkpartitionierungen vorliegen. Eine Übersicht über die verschiedenen Quorengrößen der einzelnen Quorumprotokolle ist in der Tabelle 2.1 dargestellt.

2.2.5 Aufrechterhaltung

Churn führt in verteilten Systemen zu Datenverlust. Damit die Verfügbarkeit redundant gespeicherter Daten nicht verloren geht, muss langfristig verlorene Redundanz wiederhergestellt werden. Aufrechterhaltungsstrategien (*maintenance strategies*) entscheiden wie und wann neue Kopien erstellt werden. Es existieren zwei Aufrechterhaltungsansätze. Beim reaktiven Ansatz wird nach einem

Knotenausfall mit der Wiederherstellung begonnen, beim proaktiven wird vor einem Knotenausfall neue Redundanz generiert [98].

Eines der größten Probleme bei der Umsetzung einer effizienten Aufrechterhaltungsstrategie ist die Unterscheidung zwischen transienten und permanenten Knotenausfällen. Bei Netzwerkschwankungen kann ein Knoten k zeitweise unerreichbar sein und von einem Ausfalldetektor fälschlicherweise als ausgefallen gemeldet werden. Dies kann zu einer unnötigen Aufrechterhaltungsoperation führen, da k nur kurzzeitig nicht erreichbar war und keine Redundanz verloren gegangen ist. Das Problem würde nicht existieren, wenn perfekte Ausfalldetektoren existieren würden. Ein perfekter Ausfalldetektor ist jedoch in asynchronen Systemen nicht implementierbar [36]. Der Grund hierfür liegt in der Asynchronität des Netzwerkes, das keine zeitliche Grenze für die Zustellung einer Nachricht kennt [84]. Daraus folgt, dass eine perfekte Unterscheidung zwischen sehr langsamen Knoten, deren Antwort noch nicht empfangen wurde und ausgefallenen Knoten, die nicht mehr Antworten, nicht möglich ist. Somit können fehlerhafte Ausfallmeldungen in asynchronen DHTs auftreten und zu unnötigen Aufrufen der Aufrechterhaltung führen.

Die Reaktive Aufrechterhaltung erstellt neue Kopien in Reaktion auf Ausfälle. Eine Umsetzungsmöglichkeit ist die periodische Verfügbarkeitsprüfung der verwalteten Objekte, die bei Unterschreitung eines Schwellenwertes die Regeneration auslöst. Es existieren zwei Arten der reaktiven Aufrechterhaltung, *eager repair* und *lazy repair*.
Eager repair ist die sofortige Wiederherstellung eines verlorenen Objektes infolge eines Ausfalles, angewandt in [20, 51, 70, 6, 102]. Diese Methode ist sehr ineffizient, da nicht zwischen transienten und permanenten Ausfällen unterschieden wird und häufig unnötige Wiederherstellungen durchgeführt werden [9].
Lazy repair beschreibt eine verzögerte Wiederherstellung von verlorenen Objekten, verwendet in [9, 47, 102]. Hierbei wird versucht abzuschätzen, ob ein Objekt durch einen transienten oder permanenten Fehler verloren gegangen ist. Die Qualität dieser Schätzung ist proportional zur Effizienz der Regenerationslösung. Im schlechtesten Fall werden so viele Wiederherstellungen angestoßen wie von einer *Eager repair*-Methode und im besten Fall so viele wie reale permanente Fehler aufgetreten sind.

Die Unterscheidung zwischen transienten und permanenten Fehlern erfolgt meist durch *timeouts*. Ist ein Objekt länger als ein bestimmter Schwellenwert nicht erreichbar, gilt es als permanent ausgefallen. Je mehr transiente Fehler in ihrer temporären Dauer unter dem *timeout* liegen, desto weniger Wiederherstellungen werden durchgeführt. Die Verzögerung der Wiederherstellung um einen *timeout* verlängert die Reaktionszeit bei permanenten Fehlern und vergrößert damit das „*window of vulnerability*“. Das *window of vulnerability* beschreibt ein Zeitintervall, indem jeder weitere Ausfall zum Verlust der Verfügbarkeit führt. Fällt eine von zwei Kopien aus, entspricht das *window of vulnerability* der Dauer der Erstellung einer neuen Kopie.

Ein Problem bei reaktiven Mechanismen sind *positive feedback loops*, die zu Netzwerküberlastungen führen können [76]. Eine *positive feedback loop* entsteht, wenn ein Ausfalldetektor eine falsche

Ausfallmeldung generiert, die zu einer Regeneration führt. Durch die erhöhte Netzwerklast generiert ein weiterer Ausfalldetektor eine weitere falsche Ausfallmeldung, die wiederum eine Regeneration auslöst. Auf diese Art kann eine Schleife entstehen, die das System zum kollabieren bringt.

Reaktive Aufrechterhaltungsstrategien verursachen eine ungleichmäßige und höhere Bandbreitennutzung als proaktive Strategien. So entstehen minimale Kosten, wenn keine Ausfälle stattfinden und hohe Kosten, falls Ausfälle auftreten [86]. Dies ist zum einen ein Vorteil, da keine Bandbreite verwendet wird, wenn kein Fehler auftritt, zum anderen kann es zu Spitzen bei der Bandbreitennutzung und zu Netzüberlastungen (*congestion*) führen.

Proaktive Aufrechterhaltung beschreibt die Erstellung neuer Objekte bevor Knoten ausfallen. Diese Technik der präventiven Erstellung neuer Redundanz mit konstanter bzw. variabler Geschwindigkeit wurde in [44, 86] verwendet. Dies hat den Vorteil, dass kurzzeitige Spitzenwerte in der Netzwerkauslastung, wie sie bei der reaktiven Aufrechterhaltung nach Fehlern vorkommen, vermieden werden [86]. Des Weiteren sind die Aufrechterhaltungskosten für einen beliebigen Redundanzgrad vorhersehbar. In Tempo [86] wurde durch Simulation gezeigt, dass mit proaktiver Aufrechterhaltung die gleichen Dauerhaftigkeitsgrade sichergestellt werden können, wie mit reaktiven Methoden. Die Erstellung nicht benötigter Kopien steht dem Ziel der Minimierung der Bandbreitennutzung entgegen, auch wenn über einen längeren Zeitraum der Gesamtbandbreitenverbrauch bei proaktiver und reaktiver Redundanzaufrechterhaltung identisch ist [86]. Eine ständige Erstellung von neuen Kopien verursacht zum einen erhöhte Verwaltungskosten zum anderen kann es zu Zugriffsproblemen kommen. Die Verwaltungskosten entstehen durch das Abspeichern der Position der neuen Kopie. So muss diese entweder in einem zentralen oder dezentralen Speicher eingefügt werden oder der *Routing*-Algorithmus muss angepasst werden. Zugriffsprobleme können vor allem in Quorumsystemen entstehen, wenn die Anzahl der Kopien, eine vom Zugriffsschema abhängige Grenze, überschreitet (siehe 2.5.3).

2.3 Scalaris

Scalaris[1] [83] ist eine *mutable* DHT, die die Schlüssel-Wert-Paare in einer *In-Memory*-Datenbank ablegt und Chord als ringbasiertes SON für logarithmisches *Routing* verwendet.

Scalaris verwendet *Mirroring* als Redundanzform sowie das *symmetric placement*, das r Kopien eines jeden Datums symmetrisch im Ring abgelegt. Der streng konsistente Zugriff auf Schlüssel-Wert-Paare erfolgt über das *Majority Quorum*. Weiterhin wird das *crash-stop*-Fehlermodell verwendet, sodass einmal ausgefallene Knoten dem System nicht wieder beitreten.

Scalaris unterstützt die verteilte Abarbeitung von Transaktionen mit Hilfe der Algorithmen Paxos Consensus [55] und Paxos Commit [43]. Damit wird ein ACID-konformer Zugriff auf multiple Schlüssel-Wert-Paare ermöglicht. Ein adaptierter Paxos Commit [81] stellt einen nicht

1 Scalaris Projekt `http://code.google.com/p/scalaris/`

blockierenden *atomic commit* zur Verfügung, um Transaktionen abzuschließen. Dieser verwendet den Paxos Consensus, um für jedes in der Transaktion verwendete Schlüssel-Wert-Paar über *prepared* (Änderung kann übernommen werden) oder *abort* (Änderung kann nicht übernommen werden) zu entscheiden. Nur wenn alle Paxos Consensus Instanzen *prepared* entschieden haben, kann die Transaktion erfolgreich durch Paxos Commit abgeschlossen werden. Kann eine Transaktion nicht erfolgreich abgeschlossen werden, verbleibt das System in dem vorherigen Zustand. Detailliertere Erläuterungen zum *atomic commit* von Scalaris sind in [68, 81] zu finden.

Scalaris unterliegt dem CAP-Theorem und kann damit nicht zugleich über Verfügbarkeit, Konsistenz und *partition tolerance* verfügen. Scalaris verletzt unter bestimmten Umständen die Verfügbarkeit, um die Datenkonsistenz sicherzustellen und besitzt damit die Eigenschaften C und P. Die Verfügbarkeit wird verletzt, wenn der Chord Ring in zwei Teile zerfällt und keine Mehrheit mehr in einem oder beiden Ringen entstehen kann.

2.4 Erkennung von Mengendifferenzen (*Set Reconciliation*)

Replizierte Datenbanken müssen langfristig synchronisiert werden, um die Vorteile der Redundanz zu bewahren und der Divergierung entgegenzuwirken. Im Allgemeinen ist die Anzahl der asynchronen Elemente zweier Datenbanken geringer als die Anzahl der gespeicherten Elemente. Hierdurch ist es effizienter bei einer Synchronisation nur die asynchronen Einträge zu übertragen anstatt alle. Die effiziente Erkennung der asynchronen Elemente wird *Set Reconciliation* genannt [66]. Das Problem der Erkennung von Differenzen zwischen zwei Datenbanken kann im Kontext von DHTs mit Replikation wie folgt formalisiert werden.

Seien S_A, S_B die Mengen der belegten Schlüssel zweier Knoten A, B. Die Differenzmenge $S_{\Delta A} = S_A \setminus S_B$ stellt die Menge der Schlüssel von A dar, die B fehlen respektive für $S_{\Delta B} = S_B \setminus S_A$. Die Vereinigung der beiden Differenzmengen entspricht der symmetrischen Differenz $S_A \triangle S_B = S_{\Delta A} \bigcup S_{\Delta B}$, nachfolgend Δ. Ziel der *Set Reconciliation* ist die Berechnung von $S_{\Delta A}, S_{\Delta B}$ mit dem kleinst möglichen Kommunikations- und Bandbreitenaufwand. Damit nach der Übertragung von $S_{\Delta A}$ an B und $S_{\Delta B}$ an A die Mengen identisch sind $S_A \equiv S_B$.

Die informationstheoretische untere Grenze der zu übertragenden Bits für die kontextlose Bestimmung von Δ ist nach [64]:

$$\log_{10}\left[\binom{2^b - |S_A|}{|S_{\Delta B}|} + \binom{2^b - |S_B|}{|S_{\Delta A}|}\right] \tag{2.4}$$

Nachfolgend wird eine Auswahl von *Set Reconciliation* Algorithmen im Detail vorgestellt.

2.4.1 Bloom Filter

Der *Bloom Filter* [13], benannt nach Burton H. Bloom, ist ein m-Bit-Array, dessen Bits initial null sind. Eine Menge $S = \{s_1, \ldots, s_n\}$ kann komprimiert als *Bloom Filter* $B(S)$ dargestellt werden, indem jedes $s \in S$ mit Hilfe von K Hashfunktionen auf K Bits des Bit-Array abgebildet wird. Das Verhältnis $\frac{m}{n}$ wird Kompressionsrate genannt. Die K Hashfunktionen $h : U \to \mathbb{N} \mod m$ bilden das Universium U auf das Bit-Array im Idealfall gleichmäßig und unabhängig ab. Ein Element $s \in U$ wird dem *Bloom Filter* hinzugefügt, indem die Bits $\{h_1(s), \ldots, h_K(s)\}$ auf 1 gesetzt werden. Ein Element $x \in U$ ist in $B(S)$ enthalten, wenn alle $h_k(x)$ Bits eins sind. Ist eines dieser Bits Null, so ist mit hundertprozentiger Wahrscheinlichkeit $x \notin S$, sonst ist $x \in S$ mit einer gewissen Fehlerwahrscheinlichkeit FP. Im Fall $x \in B(S) \wedge x \notin S$ wird von einem *false positive* gesprochen, dessen Wahrscheinlichkeit FP wächst proportional zur Datenmenge die der *Bloom Filter* repräsentiert, da die Kollisionswahrscheinlichkeit mit der Anzahl der Elemente bei konstantem m steigt.

Die Fehlerwahrscheinlichkeit $FP = (1-p)^K$ resultiert aus der Wahrscheinlichkeit $p = (1 - \frac{1}{m})^{Kn}$, dass ein Bit nach n hinzugefügten Elementen Null ist [67]. Diese kann auf $FP = (\frac{1}{2})^K \approx (0{,}6185)^{\frac{m}{n}}$ minimiert werden, wenn K nach Formel 2.5 optimal gewählt wird [67].

$$K = \ln(2)(\frac{m}{n}) \tag{2.5}$$

In der Praxis ist FP und S häufig gegeben, sodass die optimale Bit-Array Größe mit der Formel 2.6 aus [16] berechnet werden kann.

$$m \geq n * \log_2 e * \log_2 \frac{1}{FP} \tag{2.6}$$

Ein Problem bei der praktischen Umsetzung des *Bloom Filter* ist das Auffinden von K unabhängigen zufälligen Hash-Funktionen, die für die optimale Abbildung von U auf $\mathbb{N} \mod m$ nötig sind. Zum einen ist das Auffinden von K voneinander unabhängigen Hashfunktionen schwierig, zum anderen sind $K * |S|$ Hashoperationen aufwendig zu berechnen. In [50] wurde gezeigt, dass zwei unabhängige Hashfunktionen $h_1(x), h_2(x)$ ausreichen um eine Gruppe von K unabhängigen Hashfunktionen der Form

$$g_i(x) = h_1(x) + ih_2(x) + i^2 \mod m, \text{ für } i = 0 \ldots k-1 \tag{2.7}$$

zu bilden. Diese erreichen in der Praxis die gleiche Fehlerwahrscheinlichkeit wie K unabhängige zufällige Hashfunktionen, die in der Theorie nötig sind um die oben angegebene Fehlerwahrscheinlichkeit zu erreichen [50].

Bloom Filter stellen eine der effizientesten Darstellungen einer Menge S in $O(|S|)$ Bits dar. Die Prüfung, ob ein Element in S ist, kann mit $O(1)$ Schritten erfolgen. Weiterhin kann die Vereinigung zweier Mengen A,B, die als *Bloom Filter* kodiert sind, mit der Bit-weisen *AND* Verknüpfung

berechnet werden.

Bloom Filter sind jedoch ineffektiv beim Lösen des *Set Reconciliation* Problems, da sie linear mit den repräsentierten Daten wachsen und nicht in der Höhe von Δ [66]. Weiterhin können fehlende Elemente nur durch vollständige Aufzählung gefunden werden.

Bloom Filter werden seit langem für eine Vielzahl von Anwendungen verwendet, sodass der aktuelle Survey [90] allein 22 Varianten aufzählt. Diese fügen dem *Bloom Filter* verschiedenste Eigenschaften hinzu oder modifizieren diese. Der Aufbau eines Bloom Filters ist teuer und verlangt eine vollständige Aufzählung der zu kodierenden Daten. Um diese Kosten zu amortisieren wird häufig neben der Datenbank S der *Bloom Filter* $B(S)$ gespeichert und Operationen auf S werden auch auf $B(S)$ durchgeführt. Damit existiert $B(S)$ bei Anfragen bereits und muss nicht neu erstellt werden. Um eine solche Bauweise zu unterstützen wird eine Löschfunktion benötigt, die auf $B(S)$ operiert, was zur Entstehung von *Countable Bloom Filter* [35] führte. Diese ersetzen die m-Bits zu Zählern, die beim Einfügen inkrementiert und beim Löschen dekrementiert werden. Eine implizite Löschfunktion wird in [14] vorgeschlagen, bei der der *Bloom Filter* nach einer bestimmten Zeit Elemente vergisst und diese dadurch löscht.

Bei den *Retouched Bloom Filter* werden *false positives* durch *false negatives* eingetauscht, in denen mit hundertprozentiger Sicherheit ein Element $x \in S$ erkannt werden kann aber nur mit einer Fehlerwahrscheinlichkeit die Aussage getroffen werden das $x \notin S$ [31]. Ein für die Übertragung optimierter *Bloom Filter* ist der *compressed bloom filter* [67].

Eine weitere Variante ist der Bloomier Filter [22], welcher sich grundsätzlich von allen anderen Varianten unterscheidet. Dieser kodiert eine Multi-Menge bzw. eine Funktion und kann damit Werte zu jedem Element der dargestellten Menge speichern. Diese Variante hat die Nachteile einer verlängerten Abfragezeit, einer erhöhten Konstruktionszeit sowie einem erhöhten Speicherplatzverbrauch. In [21] wird eine verbesserte Konstruktion vorgeschlagen die drei bis fünf mal schneller ist als die Originale aus [22]. Eine Verbesserung im Speicherverbrauch konnte jedoch nicht erzielt werden.

Invertible Bloom Lookup Tables (IBLT) [42] stellen eine Tabelle dar, die dem *Bloom Filter* ähnlich ist und können die eingefügten Paare Auflisten. Die Tabelle besteht aus t Zellen mit jeweils drei Zählern. Einer für die Anzahl der in die Zelle eingefügten Paare sowie einer für die Schlüssel und für die Wertsumme. Die Tabelle besitzt K Spalten für die jeweils eine Hashfunktion benötigt wird. Ein Paar (k,v) wird dem IBLT hinzugefügt in dem die Zelle jeder Spalte durch die Hashfunktion bestimmt wird und der Paar-Zähler um 1, die anderen beiden Zähler jeweils um den Schlüssel und den Wert erhöht werden. Beim Löschen werden die Zähler jeweils um Schlüssel und Wert verringert.

Die Differenz Δ von S_A und S_B wird bestimmt durch Kodierung von S_A in einen IBLT und anschließender Subtraktion aller Elemente von S_B. Die Zellen des IBLT mit Paar-Zähler 1 sind ein Element aus $S_{\Delta A}$. Ein größer Vorteil des IBLT ist, dass die Zellenanzahl nicht von der Menge der zu kodierenden Elemente abhängt sondern von $|\Delta|$. Sodass $t \geq |\Delta|$ gewählt werden sollte. Diese

Methode ist jedoch nicht für die Lösung des Aktualisierungsproblems geeignet, da die Subtraktion unterschiedlicher Paar-Versionen nicht unterstüzt wird.

2.4.2 Characteristic Polynom Interpolation

Minsky Y. et al. stellen in [66] drei Protokolle vor, die auf der Interpolation von charakteristischen Polynomen (CPI) basieren. Die belegten Schlüssel $S_A = \{x_1,x_2,\ldots,x_n\}$ eines Knoten A werden als charakteristisches Polynom $\theta_{S_A}(Z) = (Z - x_1)(Z - x_2)\cdots(Z - x_n)$ dargestellt, dessen Nullstellen die belegten Schlüssel repräsentieren. Die Kernidee des Ansatzes ist, dass sich durch die Division zweier Polynome die gemeinsamen Elemente aufheben und nur die Differenzmengen $S_{\Delta A}$ und $S_{\Delta B}$ übrig bleiben. Die charakteristischen Polynome $\theta_{S_{\Delta A}}(Z)$ sowie $\theta_{S_{\Delta B}}(Z)$ entsprechen den Differenzmengen $S_{\Delta A}$ bzw. $S_{\Delta B}$ in Polynomform. Die folgende Gleichung aus [66] verdeutlicht die Eliminierung gemeinsamer Elemente durch Division:

$$\frac{\theta_{S_A}(Z)}{\theta_{S_B}(Z)} = \frac{\theta_{S_A \cap S_B}(Z) * \theta_{S_{\Delta A}}(Z)}{\theta_{S_A \cap S_B}(Z) * \theta_{S_{\Delta B}}(Z)} = \underbrace{\frac{\theta_{S_{\Delta A}}(Z)}{\theta_{S_{\Delta B}}(Z)}}_{*}$$

Die Berechnung der rationalen Funktion $*$ würde eine Übertragung der Polynome voraussetzen und wäre damit genauso teuer wie die vollständige Übertragung von S_A bzw. S_B. Wird die Funktion $*$ jedoch interpoliert, muss nur eine Anzahl von Auswertungspunkten übertragen werden. Die Anzahl der erforderlichen Auswertungspunkte $\bar{\Delta} \geq |S_A \triangle S_B| = |\Delta|$ muss mindestens der Anzahl der Elemente der symmetrischen Differenz entsprechen damit genügend Nullstellen vorhanden sind. Der vollständige Algorithmus mit bekanntem $\bar{\Delta}$ ist in Algorithmus 1 dargestellt. Die Interpolation wird

Algorithmus 1 CPISync

1) Die Knoten A,B werten $\theta_{S_A}(Z)$ bzw. $\theta_{S_B}(Z)$ an den gleichen $\bar{\Delta}$ Auswertungspunkten aus.
2) A sendet B die Ergebnisse und B berechnet für jeden Auswertungspunkt $\theta_{S_A}(Z)/\theta_{S_B}(Z)$. Anschließend wird $*$ durch die Interpolation der eben berechneten Punkte erhalten.
3) Durch Faktorisierung von $\theta_{S_{\Delta A}}$ und $\theta_{S_{\Delta B}}$ können die Mengen $S_{\Delta A}$ bzw. $S_{\Delta B}$ rekonstruiert werden.

mit Hilfe einiger Auswertungspunkte vorgenommen. Diese werden gewonnen indem $\chi_A(Z)$ und $\chi_B(Z)$ an einer Auswahl von Punkten ($Z = \{1,2,3,4,5,6\}$) ausgewertet werden und die $|Z|$ Werte der rationalen Funktion interpoliert werden. Der Grad des interpolierten Polynoms entspricht der Anzahl der Auswertungspunkte, wodurch kleinere Differenzmengen weniger Berechnungsaufwand erfordern. Nach dem Erhalt der rationalen Funktion können Zähler und Divisor fakturiert werden, wodurch die jeweiligen Differenzmengen erhalten werden können.

Damit die Auswertungspunkte an den Polynomen $\theta_A(Z)$ und $\theta_B(Z)$ nicht bei jedem Abgleich

aufwendig errechnet werden müssen, können die Auswertungsergebnisse aufgehoben und modifiziert werden. Wird ein Element dem Knoten A hinzugefügt, so ist der neue Auswertungspunkt $\theta_A(Z)_{neu} = \theta_A(Z)_{alt} * (Z - x)$, wird ein Element gelöscht ist $\theta_A(Z)_{neu} = \frac{\theta_A(Z)_{alt}}{(Z-x)}$. Mit Hilfe dieser iterativen Berechnung ist der Berechnungsaufwand für die Auswertungspunkte der charakteristischen Polynome zeitlich verteilt und amortisiert sich.

Der Berechnungsaufwand für die Interpolation beträgt $O(\bar{\Delta}^3)$ und die Kosten für die Auswertung der lokalen Polynome an den $\bar{\Delta}$ Auswertungsstellen betragen $O(\bar{\Delta})$. Die Kommunikationskosten von $(\bar{\Delta}+1)(b+1)-1$ zu übertragenden Bits ist fast optimal, wobei b die Bitlänge eines Elementes x ist. Die Kommunikationskosten sind jedoch auf die Abschätzung von $\bar{\Delta}$ angewiesen, deren Qualität über den Berechnungsaufwand und Kommunikationsaufwand entscheidet. Im optimalen Fall mit $\bar{\Delta} = |\Delta| = |S_A \triangle S_B|$ beträgt der Kommunikationsaufwand optimale $|\Delta|b$ Bits, sodass nur die fehlenden Elemente versendet werden. Weiterhin beeinflusst die Höhe von $\bar{\Delta}$ den Grad des interpolierten Polynoms $*$, dass den Berechnungsaufwand proportional bestimmt. Aufgrund dessen ist diese Methode für große Mengendifferenzen (Δ) langsam, da der Berechnungsaufwand sehr groß ist.

Dieser kubisch von $\bar{\Delta}$ abhängige Berechnungsaufwand kann auf eine lineare Abhängigkeit verringert werden. Dies wird mit Hilfe eines *divide-and-conquer* Ansatzes erreicht, der in [65] beschrieben wird. Dort wird durch Partitionierung von S_A,S_B die Berechnungskomplexität verringert, sodass für jede Partition $\bar{\Delta}$ klein ist.

Die Abschätzung von $\bar{\Delta}$ ist schwierig und stellt ein Problem für die praktische Umsetzung dar. *Minsky Y. et al.* stellen eine Protokollvariante vor, in der $|\Delta|$ nicht bekannt sein muss. Dabei wird $\bar{\Delta}$ sukzessiv erhöht bis $\bar{\Delta} \geq |\Delta|$. Dies ist genau dann der Fall, wenn die durch $\bar{\Delta}$ gewonnene rationale Funktion mit der eigenen übereinstimmt. Die Äquivalenz wird durch Vergleich der Bildwerte der beiden Funktionen an zufälligen Punkten überprüft. Die Kommunikationskosten dieser Variante sind um die Übertragung der zusätzlichen Auswertungspunkte erhöht. Die Methode erhöht den Rechenaufwand, da jeder Test $\bar{\Delta} \geq |\Delta|$ die Berechnung der interpolierten Funktion $*$ erfordert.

Der CPI Algorithmus gehört zu den *single-round*-Algorithmen und löst damit das *Set Reconciliation* Problem in einer Kommunikationsrunde. Hierdurch ist er für eine Vielzahl von Anwendungsfällen geeignet in denen häufige Kommunikationsrunden nicht praktikabel sind, wie *deep space communication*. Die Kommunikationskosten von $(\bar{\Delta}+1)(b+1)-1$ Bits liegen sehr nahe am Optimum von $|\Delta|b$ Bits. Nachteile dieser Methode stellen der starke Rechenaufwand sowie die Abschätzung von $\bar{\Delta}$ dar.

2.4.3 Merkle Tree

Ein *Merkle-Tree* [62], auch *Hash Tree* genannt, ist ein n-ärer Baum dessen Blätter den Hash-Wert über eine Menge von Objekten enthalten und dessen innere Knoten den Hash-Wert über ihre n Kindknoten enthalten. Sind bei dem Vergleich zweier innerer Knoten die Hash-Werte identisch, so sind deren Kindknoten identisch. Diese Eigenschaft ermöglicht einen effizienten Vergleich großer

Datenmengen, da identische Teilmengen übersprungen werden können.

J. Cates verwendete in *DHash* [20] erstmals *Merkle-Trees* zur Synchronisation und entwickelte ein *multi-round* Synchronisationsprotokoll. Dieses Protokoll vergleicht die Knoten A,B von der Wurzel an. Stimmen die Hash-Werte der Wurzeln nicht überein, werden alle Kindknoten verglichen. Gleichen sich zwei Knoten so gelten alle Kindknoten als verglichen und identisch. Dieser Abgleich erfolgt solange bis alle Knoten verglichen wurden. Ist der Algorithmus an einem Blatt angekommen dessen Hash-Wert zwischen A und B nicht übereinstimmt, werden alle n Schlüssel übertragen, die dieses Blatt bildeten. Nur auf diese Weise können hinzugefügte bzw. veränderte Elemente erkannt werden.

Das *DHash*-Übertragungsprotokoll für *Merkle-Trees* kann auch bei Optimierung nicht verhindern, dass beim Hash-Vergleich die fehlenden Elemente nicht genau identifiziert werden können. Daher ist es immer nötig alle Blatt bildenden Elemente zu übertragen, wodurch Zusatzkosten entstehen.

Merkle-Trees sind teuer in ihrer Erhaltung, da jedes Hinzufügen und Entfernen von Schlüssel-Wert-Paaren das erneute Erstellen eines Hash-Wertes („*rehash*“) von $O(\log_n k)$ inneren Knoten erfordert, wobei k die Gesamtzahl der Knoten darstellt. Das Auffinden eines neuen Elements in einem *Merkle-Tree* erfordert den Vergleich aller Knoten von der Wurzel bis zum jeweiligen Blatt, sowie die Übertragung aller Blatt bildenden Schlüssel. Diese Methode ist offensichtlich nur bei geringen Differenzen zwischen A und B effizient, da im *worst-case* in jedem Blatt ein neuer Wert existiert, werden alle Elementschlüssel sowie alle inneren Knoten übertragen. Der *Merkle-Tree* ist ab einer Differenz kleiner gleich 5% effizienter als die vollständige Übertragung der Mengen [20].

Mit Hilfe eines *Nye's Trie* [94], der ein modifizierter *Burst Trie* [45] ist, können die Kosten für das Hinzufügen und Entfernen von Elementen von $O(\log_n k)$ *rehash* Operationen auf $O(\log_n k)$ XOR-Operationen verringert werden. Der *Nye's Trie* kann zur schnellen Erstellung von *Merkle-Trees* verwendet werden und ist in der Erhaltung günstiger. Er besteht aus sogenannten *Container*, die eine Sammlung von Werten darstellen, die geteilt werden (*burst*), wenn sie zu groß werden. *Container* sind üblicherweise die Blätter eines Präfixbaumes. Der Präfixbaum existiert jedoch erst ab dem ersten „*burst*“ des initialen *Containers*. Der *Nye's Trie* ermöglicht eine wesentlich schnellere und kostengünstigere Erstellung eines *Merkle-Trees*.

Merkle-Trees können auch Versionsunterschiede zwischen Objekten erkennen, wenn wie in Dynamo [28] und OpenDHT [77] der Hash über die Werte der Schlüssel-Wert-Paare erstellt wird.

2.4.4 Approximate Reconciliation Tree

Der *Approximate Reconciliation Tree* (ART) [18] ist ein hybrides *Set Reconciliation* Protokoll, das einen *Merkle-Tree* als *Bloom Filter* codiert. ART besteht in seiner Standardform aus zwei *Bloom Filtern*. Der erste B_{Tree} enthält alle inneren Knoten des *Merkle-Tree* und ein weiterer B_{Data} enthält alle Datenelemente. Die Kollisionsfreiheit von B_{Tree} ist besonders wichtig und bei $\Theta(\log(S_B))$ Bits mit hoher Wahrscheinlichkeit gegeben [18]. Für die Kodierung der Daten S_A in B_{Data} genügen $\Omega(\log\log(|S_B|))$ Bits sowie $\Theta(\log\log(|S_B|))$ Hash-Funktionen, analog für S_B. Bei der Verwendung

einer konstanten Anzahl von c Hash-Funktionen werden $\Omega(\sqrt[c+1]{\log(|S_B|)})$ Bits je Element in B_{Data} benötigt.

Die größte Schwachstelle von ART ergibt sich aus der hohen Fehlerrate der *Bloom Filter*. Tritt eine Kollision beim Vergleich zweier innerer Knoten in B_{Tree} auf, wird ein ganzer Bereich nicht abgeglichen. Um die Wahrscheinlichkeit solcher Kollisionen zu verringern und die Effizienz zu erhöhen, wird ein hoher Verzweigungsgrad sowie die Einführung eines *correction level* empfohlen.

Die Erhöhung des Verzweigungsgrades des *Merkle-Trees* verringert die Baumhöhe und verkleinert den durch eine Hashkollision übersprungenen Bereich. Das *correction level* erhöht die Genauigkeit der Erkennung gemeinsamer Teilbäume. Dabei wird die Suchfunktion verändert, indem die Expansion eines Knotens nicht gestoppt wird, wenn zwei Knoten gleich sind, sondern erst wenn die letzten c Knoten als gleich erkannt wurden. Das *correction level* c erhöht dadurch den Vergleichsaufwand bei einem Baum mit Verzweigungsgrad b um einen Faktor von b^c.

Die Kodierung der inneren Knoten eines *Merkle-Tree* in einem *Bloom Filter* verringert die Kommunikationskosten von $O(\log(|S_B|)$ auf 1 und fügt dem *Bloom Filter* die Vorteile eines suchbasierten Ansatzes hinzu. Weiterhin können Hash-Kollisionen innerhalb des *Merkle-Tree* stark reduziert werden, da die Größe der Hash-Werte innerer Knoten nicht mit der Nachrichtengröße korrelieren, da diese vom *Bloom Filter* abhängt [18]. Somit können die Hash-Werte innerer Knoten problemlos genügend groß gestaltet werden.

ART gehört zu den probabilistischen *Set Reconciliation* Algorithmen und ist nicht genauer als ein einzelner *Bloom Filter* der selben Größe. Durch die Baumstruktur können Schlüsselbereiche übersprungen werden und müssen nicht mit B_{Data} abgeglichen werden wodurch ein Nachteil der reinen *Bloom Filter* Lösung ausgeglichen wird.

Die Evaluation von ART in [18] zeigt eine stark erhöhte Geschwindigkeit bei der Erkennung der Mengenunterschiede im Gegensatz zur Verwendung eines einzelnen Bloom-Filters. Dies zeigt sich besonders deutlich, wenn S_A,S_B sich nur in 0,001% unterscheiden und ART *correction level* von $c = 2$ nur $\frac{1}{20}$ der Erkennungszeit des Bloom Filters benötigt. Es ist auch ersichtlich, dass das *correction level* signifikanten Einfluss auf die Erkennungsgenauigkeit besitzt, sodass bei $c = 0$ nur bis zu 20% der Differenzen erkannt werden, wohingegen bei $c = 4$ dieser Wert auf über 80% steigt.

2.4.5 Klassifikation

Die Lösungen für das *Set Reconciliation* Problem können im Allgemein an drei Merkmalen, Mitteilungsrunden, Genauigkeit und Art der Differenzbestimmung unterschieden werden. Die Tabelle 2.2 zeigt zusätzlich eine Übersicht der vorgestellten *Set Reconciliation* Methoden.

Das erste Klassifikationsmerkmal sind die Mitteilungsrunden, wobei zwischen *single-round* und *multi-round* Algorithmen unterschieden wird. *Single-round* Algorithmen benötigen nur eine Mitteilung zum Abgleich. Diese Mitteilung ist jedoch üblicherweiser größer sowie rechenintensiver in Ihrer Erstellung als jene Mitteilungen, die bei *multi-round* Algorithmen versendet werden. *Multi-round* Algorithmen verwenden eher kleine Nachrichten, die in mehreren Runden versendet

Methode	Nachrichtengröße	Komm. Runden	Query Zeit	Exakt	Aufbauzeit für $S_X = S_A \vee S_B$
Aufzählung	$O(\|S_A\|)$	1	$O(\|S_A\| + \|S_B\|)$	Ja	$O(\|S_X\| \log(u))$
CPI [66]	$O(\|\Delta\| \log(u))$	1	$O(\|\Delta\|^3)$	Ja	$O(\|\Delta\| \log(u)$
Bloom Filter [13]	$O(\|S_A\|)$	1	$O(\|S_B\|)$	Nein	$O(\|S_X\|)$
Merkle Tree [62]	$O(\log(h) \log(\|S_B\|)$	$O(\log(\|S_B\|)$	$O(\|\Delta\| \log(\|S_B\|))$	m.h.W.	$O(\|S_X\| \log(\|S_X\|))$
ART [18]	$O(\|S_A\|)$	1	$O(\|\Delta\| \log(\|S_B\|)$	Nein	$O(\|S_X\| \log(\|S_X\|))$

Tabelle 2.2: Vergleich der *Set Reconciliation* Methoden für zwei Mengen S_A, S_B (modifiziert, Quelle: [18])

werden.

Bei der Klassifizierung nach der Erkennungsgenauigkeit wird zwischen *exakten* und *approximativen* Algorithmen unterschieden. Algorithmen die *Hashing* verwenden, wie *Bloom Filter* (siehe 2.4.1), zählen zu den approximativen und können keine hundertprozentige Erkennung aller Differenzen garantieren, da durch Kollisionen Erkennungsfehler entstehen können, deren Wahrscheinlichkeit minimiert wird. Exakte Algorithmen sind selten und außer dem trivial Algorithmus, der S_A vollständig an B überträgt, ist in der Literatur nur CPI (vorgestellt in 2.4.2) bekannt.

Das dritte Klassifizierungsmerkmal richtet sich nach der Art der Differenzbestimmung, die such- oder aufzählungsbasiert sein kann. Bei aufzählungsbasierten Algorithmen wird S_A vollständig übertragen und alle Elemente von S_B werden in S_A einzeln gesucht. Um die Übertragungskosten zu minimieren wird eine komprimierte Darstellung von S_A erstellt, beispielsweise ein *Bloom Filter* (siehe 2.4.1). Aufzählungsbasierte Algorithmen sind bei der Erkennung großer Mengendifferenzen effizient. Suchbasierte Algorithmen sind durch die Verwendung von Baumstrukturen effizienter in der Erkennung von kleineren Mengendifferenzen. Diese Algorithmen basieren hauptsächlich auf *Merkle-Trees* (siehe 2.4.3), in denen strukturiert über Zusammenfassungen von Teilmengen gesucht werden kann. Mischformen aus beiden Ansätzen, wie ART (siehe 2.4.4) sind bei großen und kleinen Mengendifferenzen effizient [18].

2.5 Replik-Regeneration

Jedes Computersystem unterliegt einer Fehlerrate $\lambda > 0$, die dessen Ausfallhäufigkeit je Zeiteinheit angibt. Verteilte Systeme unterliegen einem Teilnehmerschwund, weil keine Rechnersysteme mit $\lambda = 0$ existieren. Daraus ergibt sich die Notwendigkeit von neu beitretenden Knoten, damit ein Systemausfall sowie Datenverlust verhindert wird.

Sei eine DHT bestehend aus K Knoten, einem Schlüsselbereich von I und einem Redundanzgrad r gegeben. Die Menge I_k enthält alle einem Knoten $k \in K$ zugewiesenen Schlüssel, sodass $I = \bigcup_{k \in K} I_k$ gilt. Äquivalent sei I_k^* die Menge der belegten Schlüssel von k, wobei $I^* = \bigcup_{k \in K} I_k^*$ der Menge aller belegten Schlüssel der DHT entspricht. Die Funktion $r^* : I \to \{0, \ldots, r\}$ gibt für jeden Schlüssel $i \in I^*$ die Anzahl der in der DHT existierenden Kopien an. Fällt in der DHT der Knoten k aus, so

sinkt der Redundanzgrad $r^*(i) = r - 1, \forall i \in I_k^*$ um eins. Aufgrund von $\lambda > 0$ fällt jeder Knoten k einmal aus und die Anzahl der existierenden Kopien strebt nach einer Systemlaufzeit t gegen Null $r^*(i) \xrightarrow[t\to\infty]{} 0, \forall i \in I^*$. Das Ziel der Regeneration ist $r^*(i) \xrightarrow[t\to\infty]{} r, \forall i \in I^*$ und damit die Bewahrung der Redundanz.

Blake und *Rodrigues* [11] zeigen, dass dieses Ziel nur unter Einschränkungen erreichbar ist. Demzufolge kann ein System aufgrund einer limitierten Bandbreite nur beschränkt viele Daten bei einer gegebenen *churn*-Rate verwalten. Sei MTBF die mittlere Zeit zwischen Knotenausfällen und MTTR die mittlere Zeit zur Regeneration verlorener Daten. Der MTBF wird durch die *Churn*-Rate bestimmt und die Höhe des MTTR richtet sich nach der Bandbreite und dass der durchschnittlichen Datenmenge je Knoten. Eine Regenerationsfunktion muss Redundanz schneller regenerieren als diese verloren geht, was das Verhältnis $MTTR < MTBF$ erfordert [11]. Da die Bandbreite in der Praxis fest ist, existiert bei jedem System eine kritische Speichermenge bzw. *Churn*-Rate in der ein verteiltes Speichersystem trotz Regeneration keine Datensicherheit mehr sicherstellen kann.

Eine besondere Herausforderung für die Umsetzung einer Regenerationsfunktion stellt die Entscheidungsfindung dar, wann eine neue Kopie erstellt werden muss. Soll in Reaktion auf den Ausfall eines Knotens k eine Regeneration durchgeführt werden, ist dem regenerierenden Knoten nur der Zuständigkeitsbereich von k bekannt, nicht die Menge der belegten Schlüssel I_k^*. Eine andere Variante ist die Beobachtung des aktuellen Redundanzgrads $r^*(i)$ für einen Schlüssel i, um bei Schwankungen eine Regeneration einzuleiten. Die dafür notwendige Zählung der vorhandenen Repliken ist jedoch schwierig in Systemen, die keine einheitliche Sicht auf die Gruppe der Repliken bieten. *Routing inconsistencies* sind die Hauptursache für eine fehlerhafte Bestimmung des Redundanzgrades eines Schlüssels $i \in I^*$. *Routing inconsistencies* führen zu unterschiedlichen Sichten auf die DHT, sodass zwei Knoten A, B unterschiedliche Knoten für den gleichen Schlüssel i für zuständig halten. Dies kann dazu führen, dass bei der Bestimmung des Redundanzgrades Knoten A ein anderes Ergebnis erzielt als B für einen Schlüssel i.

Abhängig vom Design des Regenerationsalgorithmus können neue Kopien *push*- sowie *pull*-basiert erstellt werden. Im weiteren wird der Knoten mit der fehlenden Kopie als Klient bezeichnet und der regenerierende Knoten als Server.

Die *pull* Methode erzeugt neue Kopien indem der Klient diese vom Server abruft. Hierbei stellt sich das Akquisitionsproblem, da der Klient keine Kenntnis besitzt, welche Schlüssel seines Zuständigkeitsbereichs regeneriert werden müssen [47].

Bei der *push*-basierten Methode sendet der Server seine Kopie an den Klient und erstellt somit eine neue Kopie. Diese Methode passt zu der ROWA Zugriffsstrategie oder dem häufig verwendeten *primary-backup-scheme*, in dem die Primärkopie den Zugriff serialisiert und somit auch leicht eine neue konsistente Kopie erstellen kann. Ein weiterer Vorteil ist die höhere Robustheit von *push*-basierten Ansätzen im Gegensatz zu *pull*-basierten [20]. Systeme die diese Methode verwenden [101, 9, 20, 6, 70] sind häufig nicht vollständig dezentralisiert und besitzen durch die Primärkopie einen Flaschenhals.

Eine Regenerationslösung wird als effizient bezeichnet, wenn die verwendete Bandbreite minimal ist. Daher regeneriert ein perfekter Regenerationsalgorithmus nur bei permanenten Knotenfehlern und verursacht keine Verwaltungskosten. Da eine eindeutige Unterscheidung zwischen transienten und permanenten Knotenfehlern nicht möglich ist, existiert kein perfekter Regenerationsalgorithmus. Die Menge der zu regenerierenden Daten kann von einem Algorithmus nicht beeinflusst werden, wodurch die Effizienz einer Regenerationslösung von den Verwaltungskosten und dem Umgang mit transienten Fehlern abhängt.

2.5.1 Regeneration der neusten Version

Systeme mit strikter Datenkonsistenz müssen neue Kopien entweder in der aktuellsten Version oder in der verlorenen Version wiederherstellen, um die Konsistenz zu bewahren. Die Wiederherstellung der verlorenen Version ist mit einem hohen Verwaltungsaufwand verbunden und komplex.

Ist ein r-fach repliziertes Datum in einem *Majority-Quorum*-System $\lceil \frac{r+1}{2} \rceil$ mal mit dem neusten Wert vorhanden, kann es beim Ausfall von einer (r gerade) bzw. zwei (r ungerade) aktuellen Kopie(n) zur Dateninkonsistenz kommen. Genau dann wenn Kopien regeneriert werden, die nicht die neuste Version vorweisen. In einem solchen Fall ist die Bildung einer Mehrheit aus veralteten Kopien möglich.

Die Regeneration der neusten Version verlangt ein Quorum, um den aktuellsten Wert zu einem Schlüssel i zu lesen. Dies erhöht die Kosten einer Regenerationslösung erheblich. Ist der aktuelle Wert gelesen, muss dieser auf dem Zielknoten abgelegt werden. Die *push*-Methode, in der ein Knoten die aktuelle Kopie auf den Zielknoten überträgt, ist nicht geeignet, da während der Übertragung der Wert veralten kann. Die *pull*-Methode ist effizient umsetzbar, da dem Zielknoten nur der Schlüssel mitgeteilt werden muss, dessen Kopie dieser laden und speichern soll. Eine weitere Möglichkeit der Regeneration wäre die Durchführung eines *quorum-read* gefolgt von einem *quorum-write* des gelesenen Wertes, wobei das Schreibquorum den Zielknoten zwingend enthalten muss.

2.5.2 Transiente Fehler

Ein transienter Fehler führt zu einer temporären Unerreichbarkeit eines nicht ausgefallenen Knotens. Sie entstehen durch Netzwerkschwankungen, *lookup inconsistencies* oder durch Knotenüberlastung, die fehlerhafte Ausfallmeldungen des Ausfalldetektors verursachen [91]. Die Anzahl transienter Fehler hängt nach [91] stark von der Umgebung der systembildenden Knoten ab, kommen jedoch häufig in *wide-area* Systemen vor, wie die Studie [24] zeigt. Weil transiente Fehler wie Knotenausfälle erscheinen, kann eine Regeneration durch diese ausgelößt werden, die zu unnötiger Bandbreitenauslastung sowie zu einer Steigerung des Redundanzgrades führt. Letzteres kann unter Umständen zu Problemen führen, was in 2.5.3 beschrieben wird. Die Auswirkungen transienter Knotenfehler werden derzeit mit *timeouts* und *masking* verringert.

Timeouts verzögern die Regeneration und tauschen Datensicherheit gegen Bandbreite ein. Durch

Einfügen eines *timeouts* t werden Regenerationen erst nach einer Wartezeit t ausgelößt, wodurch die Wahrscheinlichkeit sinkt, bei transienten Fehlern eine Regeneration auszulösen. Die Datensicherheit wird dafür verringert, weil der MTTR um t erhöht wird.

Das Verstecken von temporären Ausfällen mit Hilfe von zusätzlicher Redundanz wird *masking* genannt [91]. Benötigt ein System r Kopien um einen Verfügbarkeitsgrad zu erreichen, können ε zusätzliche Kopien erstellt werden, sodass bei temporären Ausfällen keine Regeneration eingeleitet werden muss. Das *masking* kommt bei den Strategien der Extra-Repliken [96, 9] sowie bei der Reintegration [25] zur Anwendung. Die Reintegration erstellt keine neuen Kopien sondern versucht Kopien, die durch transiente Fehler wiederkehren, zu reintegrieren. Daher ist es nötig die Positionen jeder Kopie zu kennen und deren Status zu verfolgen. In Kombination mit einer Aufrechterhaltungsstrategie kann die Anzahl der Kopien den Redundanzgrad überschreiten. Die Technik der Extra-Repliken erstellt ε zusätzliche Kopien, die transient verloren gegangene ersetzen können. *Masking* tauscht Speicherplatz gegen Bandbreite ein, da mehr Kopien erstellt werden als der Zielredundanzgrad vorgibt, damit bei transienten Fehlern keine neuen Kopien erstellt werden. Die Erstellung von Kopien über den Redundanzgrad hinaus führt zu zusätzlichen Kosten wie bei der proaktiven Aufrechterhaltung, siehe 2.2.5, sowie zu den Problemen die nachfolgend in 2.5.3 beschrieben werden.

2.5.3 Beschränkung der Replikenanzahl

In DHTs können neue Kopien bei der Regeneration sowie bei der Migration von Kopien entstehen. Letzteres ist bei strikten Plazierungsstrategien, wie dem *symmetric placement*, bei Knotenbeitritten nötig. Bei Quorum-basierten Redundanzzugriffsstrategien ist es erforderlich, dass zu keinem Zeitpunkt mehr als r_U Kopien eines Datums existieren. Beispielsweise geht in einem *Majority-Quorum*-System die strikte Datenkonsistenz verloren, wenn mehr als

$$r_U = \begin{cases} r+1 & \text{r gerade} \\ r & \text{r ungerade} \end{cases} \tag{2.8}$$

Kopien an einer Abstimmung teilnehmen. Nicht Quorum-basierte Zugriffsstategien wie ROWA oder ROWAA besitzen keine solche obere Schranke. Nehmen $r_U + 1$ Kopien an einer Änderungsabstimmung teil, können multiple Majoritäten entstehen, die zu einer Dateninkonsistenz führen [84].

In jeder DHT existiert auch eine untere Schranke für die Anzahl der Repliken r_L, bei deren Unterschreitung die Verfügbarkeit und die Datenkonsistenz verloren gehen. Die Schranke r_L hängt wie r_U von der Zugriffsstrategie ab und ist bei ROWA $r_L = 1$ sowie $r_L = \lceil \frac{r+1}{2} \rceil$ bei *Majority-Quorum*.

Für die Einhaltung der Schranke r_L muss $MTTR < MTTF$ gelten, sodass neue Kopien schneller erstellt werden als diese ausfallen. Die Einhaltung von r_U erfordert eine Abstimmung der Kopien in

Form eines Quorums oder einer konsistenten Sicht auf die Gruppe der Kopien. Denn nur wenn die korrekte Anzahl der Kopien bestimmt werden kann, kann entschieden werden, ob eine neue Kopie erstellt werden muss. Dies ist mit der lokalen Sicht eines Knotens auf die DHT nicht entscheidbar. *Lookup inconsistencies* führen zu einer fehlerhaften Bestimmung der Replikenanzahl und sind die Ursache für die Generierung von neuen Kopien durch die r_U überschritten werden kann [84]. Existieren mehr als r_U Kopien können *lookup inconsistencies* und *partions* zur Bildung fehlerhafter Quoren führen, die zu einem Verlust der Konsistenz führen können [84]. So können bei $r = 4$ mit $r_U + 1 = r + 2$ Kopien zwei Majoritäten gebildet werden $r_U + 1 = 6 = \lceil \frac{4+1}{2} \rceil + \lceil \frac{4+1}{2} \rceil$. Die Wahrscheinlichkeit der Bildung multipler Majoritäten ist bei geradem Redundanzgrad geringer als bei ungeradem [84, 85]. Dies ergibt sich draus, dass bei $r = 4$ zwei Kopien $r_U + 1 = r + 2$ benötigt werden, um multiple Majoritäten zu ermöglichen, im Gegensatz zu $r = 3$ mit nur einer $r_U + 1 = r + 1$.

Eine konsistente Sicht kann entweder über die Implementation eines konsistenten *lookup*-System in der DHT gelößt werden oder über das Erstellen einer konsistenten Gruppe. Die Implementation eines konsistenten *lookup*-Systems erscheint effektiver, da eine Gruppenverwaltung je Replik oder Replikengruppe schlechter skaliert als ein konsistentes *lookup*-System, das mit der Knotenanzahl skaliert.

2.5.4 Existierende Lösungen

Tempo [86] und Glacier [44] sind *immutable* DHTs mit proaktiver pull-basierter Regeneration, in denen ein fester konfigurierbarer Teil der Knotenbandbreite für die Regeneration verwendet wird. Jeder Knoten wählt periodisch eine Kopie seiner Datenbank aus und repliziert dieses. Die Auswahl der Kopie kann beliebigen Kriterien unterliegen, so wählt Tempo immer jene mit dem geringsten Redundanzgrad. Tempo und Glacier sind sich sehr ähnlich und unterscheiden sich hauptsächlich darin, dass in Glacier ab einem bestimmten Redundanzgrad keine neuen Kopien mehr erstellt werden. In Tempo hingegen können unbegrenzt viele Kopien entstehen.

In Murex [47] wird ein *pull*-basiertes, reaktives Regenerationsprotokoll vorgestellt. Das Protokoll wird auch *On-Demand*-Regeneration genannt, da nur nachgefragte Kopien auf den Knoten regeneriert werden, auf denen diese fehlen. Erhält ein Knoten K eine Anfrage nach einem Datum i, dass er nicht besitzt, jedoch in seinem Zuständigkeitsbereich liegt, so wird angenommen das er dieses besitzen sollte. Mit der Kenntnis von i kann K das Datum von i *pull*-basiert regenerieren. Dies löst das Akquisitionsproblem, erfordert jedoch eine häufige periodische Nachfrage nach allen Daten auf jedem Knoten, da sonst der Redundanzgrad für wenig nachgefragte Daten langfristig sinkt. Das periodische Nachfragen wird von *Jiang, J. R. et al.* über eine *Dummy*-Anfrage umgesetzt, die als modifizierte Leseoperation nicht beantwortet wird.

Einen hybriden Ansatz stellen *Jing Z. et al.* in ConDHT [102] vor, der zwei reaktive Regenerationsmodi unterstützt. Zum einen den *eager*-Modus, bei dem sofort nach einem Knotenausfall eine neue Kopie erstellt wird und zum anderen den *lazy*-Modus, bei dem die Regeneration verzögert

wird. In ConDHT obliegt die Regeneration eines Datums i dem Knoten K der für i zuständig ist. Er fungiert als Primärknoten und überwacht alle r Kopien, die über das *Successor placement* verteilt sind. Die Regeneration erfolgt, wie auch die Migration, mit einem *Push/Pull*-Algorithmus. Der Ausfall eines Knoten K führt zur Übernahme des Zuständigkeitsbereichs und der Regenerationstätigkeit durch den nachfolgenden Knoten N. Bemerkt K den Ausfall eines Knotens A, der eine Kopie von i besitzt, so wird im *lazy*-Modus über den *Push*-Schritt der $r-1$ Nachfolger B von K benachrichtigt, dass er eine Kopie von i besitzen soll. Im darauffolgenden *Pull*-Schritt fordert B die Kopie für i an und speichert diese ab. Im *eager*-Modus wird hingegen der *Pull*-Schritt eingespart, indem im *Push*-Schritt die Kopie mitgesendet wird. Der *lazy*-Modus des *Push/Pull* Protokolls ermöglicht eine Verzögerung der Regeneration, wodurch Knoten völlige Freiheit bei der Regeneration besitzen. So können diese die Regeneration bei starker Auslastung verschieben oder Daten deren Redundanzgrad gering ist beim Bezug bevorzugen. Der *eager*-Modus ist für Systeme mit Primärkopie geeignet, die mit einer Nachricht eine neue Kopie erstellen können.

Einer der wenigen Algorithmen, der nicht auf Basis von Schlüsseln sondern auf *Set Reconciliation* operiert, ist DHash [20]. Darin werden die Fragmente eines mit *erasure codes* replizierten Datums i mit dem *Successor-Placement* im Ring verteilt. Die reaktive Regeneration erfolgt mit Hilfe des *local maintenance* Protokolls, dass für jedes $i \in I$ vom für i zuständigen Knoten K ausgeführt wird. Dieses Protokoll synchronisiert die Datenbank von K mit seinen r *Successor*-Knoten, unter Verwendung eines *Merkle-Trees* als *Set Reconciliation* Algorithmus.

Eine große Gruppe von Regenerationsalgorithmen verwenden Konfigurationen als Hilfsmittel für die Verwaltung und Regeneration von Repliken [101, 70, 6, 61, 41, 9, 25, 71].

Rambo [61, 41] und IrisLog [71] besitzen ein Lese- sowie ein Schreib-Quorum mit flexibel konfigurierbaren Mitgliedern, die in einer Konfiguration je Schlüssel abgelegt sind. Die Konfiguration wird geändert, wenn Kopien bzw. Knoten entfernt oder hinzugefügt werden. Neu in die Konfiguration aufgenommene Knoten besitzen jedoch noch keine Kopie und bekommen diese erst beim nächsten Schreibvorgang. IrisLog verwendet einen TTL zum Löschen veralteter Konfigurationen, wohingegen Rambo einen *Garbage Collector* implementiert.

Carbonite [25] verwendet Konfigurationen für die Reintegration transient verloren gegangener Kopien, um diese über die Konfiguration wiederzufinden. Da *Carbonite* zu den *immutable DHTs* gehört, ist dies ohne Probleme für die Datenkonsistenz möglich.

Etna [70] und DhtFlex [6] verbinden die Rekonfiguration mit der Regeneration, indem neue Konfigurationen den aktuellsten Wert des Datums i enthalten. Ein Knoten, der eine neue Konfiguration vorschlägt, führt ein *majority quorum-read* durch und fügt den so erhaltenen aktuellen Wert der vorgeschlagenen Konfiguration an. Da die neue Konfiguration über r Knoten hergestellt wird, entspricht der Redundanzgrad von i wieder dem Soll-Redundanzgrad r. DhtFlex ist eine Weiterentwicklung von Etna, in der der Redundanzgrad für jedes Datum separat eingestellt werden kann und die Rekonfigurationskosten von sieben Kommunikationsrunden bei Etna auf fünf bis drei reduziert wurden.

In Om [101] existieren zwei Rekonfigurationsmechanismen. Zum einen die *failure-induced* Rekonfiguration, die ausgefallene Knoten aus einer Konfiguration entfernt und von einem beliebigen Knoten eingeleitet werden kann. Zum anderen die *failure-free* Rekonfiguration, die von einem Primärknoten im Regenerationsfall gestartet wird und der Konfiguration ein Mitglied hinzufügt. Dabei wird die *failure-free* Rekonfiguration erst ausgeführt, wenn die neue Kopie erstellt wurde und in den Betrieb aufgenommen werden soll. Die Erstellung erfolgt *Push*-basiert von der Primärkopie, die nach der ersten Übertragung auch eine Delta-Liste sendet, die die Änderungen enthält die während der Übertragung getätigt wurden. Dadurch ist die neu erstellte Kopie aktuell und kann über die *failure-free* Rekonfiguration in die Konfiguration integriert werden. Durch die Verwendung der Delta-Liste verringert sich die Dauer der Verfügbarkeitslücke auf die Dauer der *failure-free* Rekonfiguration. Om besitzt eine bessere Verfügbarkeit als Rambo, da durch den Einsatz des *Witness Quorum* eine Regeneration von einer einzelnen Kopie möglich ist. Dafür ist es durch diese Quorum Art möglich, dass die Konsistenz verletzt wird, wobei diese Wahrscheinlichkeit mit $\approx 10^{-6}$ je Regeneration sehr gering ist [101]. Om setzt das ROWA Quorum für den Datenzugriff ein. Die damit verbundene Optimierung auf Lesezugriffe kann für andere Arbeitslasten, durch Einsatz eines anderen Datenzugriffquorums geändert werden.

In TotalRecall [9] werden Konfigurationen *inodes* genannt und sind separat von den Daten, auf die sie verweisen, gespeichert. Mit Hilfe der Konfigurationen kann für jedes Datum eine eigene Redundanzform (Mirroring oder Erasure Codes) gewählt werden, genauso wie die Reaktionsgeschwindigkeit der Regeneration (*eager* oder *lazy*) und der Zielverfügbarkeitsgrad. Zugriffe auf Daten werden über eine Primärkopie serialisiert, die auch die Regeneration durchführt, wenn der Zielverfügbarkeitsgrad unterschritten wird. Die Regeneration besitzt die Semantik einer Lese-Operation gefolgt von einer Schreib-Operation, auf einer vergrößerten Knotenmenge. Dies wird durch die ROWA-Zugriffsstrategie von TotalRecall ermöglicht. Um bei transienten Fehlern Regenerationen zu vermeiden, verwendet TotalRecall *extra replicas*. Fällt der aktuelle Redundanzgrad r_A unter einem Schwellenwert r_S werden nicht nur $x = r_S - r_A$ Kopien regeneriert sondern noch ε zusätzliche, wodurch der Aufwand von Schreibzugriffen erhöht wird.

2.5.5 Diskursion

Die Regeneration von Redundanz wird durch eine ganze Reihe von Problemen eingeschränkt und erschwert. Das Kernproblem stellt die Identifikation fehlender Kopien dar. Dieses kann zum einen durch eine aufwendige Verfolgung des Redundanzgrades je Schlüssel erfolgen oder zum anderen durch die Synchronisation von Knotengruppen, die gemeinsame Kopien besitzen. Die Verfolgung des Redundanzgrades wird durch transiente Fehler und *routing inconsistencies* erschwert und verursacht wesentlich mehr Netzwerklast innerhalb des verteilten Systems als die Synchronisation einzelner Knoten. Die Synchronisation zweier Knoten erfordert den effizienten Vergleich der Knotendatenbanken mit Hilfe eines *Set Reconciliation* Algorithmus. Grundsätzlich können hierfür alle *Set Reconciliation* Algorithmen verwendet werden, wobei die Eignung der

Algorithmen vom Design des Zielsystems sowie dessen Umgebung abhängt. *Multi-round Set Reconciliation* Algorithmen sind in verteilten Systemen die über ein lokales Netzwerk verbunden sind weniger problematisch als in geographisch weit verteilten.

Systeme die strikte Quoren verwenden, müssen immer die aktuellste Version regenerieren sowie die Anzahl der Kopien nach oben beschränken, um ihre strenge Konsistenz zu erhalten. Aufgrund der hohen Kosten von Regenerationen sollte diese Funktion selten eingesetzt werden. Daraus ergibt sich die Notwendigkeit eines speziellen Umgangs mit transienten Knotenausfällen durch *timeouts* oder *masking*.

2.6 Replik-Aktualisierung

Zur Aufrechterhaltung der Redundanzvorteile gehört auch die Aktualisierung von veralteten Kopien. Diese können in allen Quorum-basierten Systemen entstehen, deren Schreibquorum kleiner als der Replikationsgrad ist, wie beispielsweise beim *Majority-Quorum*. Solche Quoren tauschen Schreiblatenz gegen Datensicherheit ein und sind die Ursache von veralteten Kopien.

Die Änderung eines r-fach replizierten Datums mit dem Schlüssel i erfordert ein Änderungsquorum von mindestens w Kopien um erfolgreich abgeschlossen zu werden. Ist $w < r$ sind $r - w$ Kopien nach einer Änderung veraltet.

Veraltete Kopien senken die Lesezugriffslatenz für i, da die Kopie, die geographisch der Anfragequelle am nächsten ist, veraltet sein kann und somit eine weiter entfernte Kopie den Wert für i ausliefern muss. Eine möglichst schnelle Aktualisierung veralteter Kopien ist daher in weit verteilten Systemen sinnvoll. Eine Aktualisierung muss jedoch nicht zwingend stattfinden, da keine Kernfunktionen durch veraltete Kopien beeinträchtigt werden und diese bei erneutem Schreibzugriff eventuell erneuert werden.

Die effiziente Umsetzung einer Replik-Aktualisierungsfunktion erfordert einen *Set Reconciliation* Algorithmus sowie eine Kommunikationsstrategie. Mit Hilfe des *Set Reconciliation* Algorithmus (siehe 2.4) werden die Versionsdifferenzen zweier Datenbanken bestimmt, die anschließend aufgelößt werden können. Die Kommunikationsstrategie wählt die zu synchronisierenden Knotenpaare mit minimalem Nachrichtenaufwand aus und stellt sicher, dass jeder Knoten ausgewählt wird.

Die Herausforderungen für eine effiziente Aktualisierungsfunktion ist daher die Vermeidung eines hohen Nachrichtenaufwandes bei der Wahl des Synchronisationspartners, sowie die Minimierung der Übertragungskosten bei der Bestimmung der Versionsdifferenzen.

2.6.1 Anpassung der *Set Reconcilication* Algorithmen

Die versionsabhängige Synchronisation der Datenbestände zweier Knoten S_A, S_B entspricht nicht genau dem in 2.4 beschriebenen eindimensionalen *Set Reconciliation* Problem. Denn die Datenbestände S_A, S_B besitzen die gleichen Schlüssel, jedoch verschiedene Versionen. Die Funktion

$v_X : S_X \to \mathbb{N}$ bildet jedes Element des Datenbestandes eines Knoten X auf eine Versionsnummer ab. Das zweidimensionale *Set Reconciliation* Problem besteht aus der Bestimmung von $S^*_{\Delta A} = \{i \mid i \in S_A \cap S_B \wedge v_A(i) > v_B(i)\}$ und $S^*_{\Delta B} = \{i \mid i \in S_A \cap S_B \wedge v_A(i) < v_B(i)\}$. Die Mengen $S^*_{\Delta A}$ und $S^*_{\Delta B}$ spiegeln die jeweils auszutauschenden Daten wieder. Die Überführung des zweidimensionalen Problems auf das eindimensionale ist sinnvoll aufgrund einer besseren Verfügbarkeit von eindimensionalen *Set Reconciliation* Algorithmen.

Das zweidimensionale Problem kann in ein eindimensionales überführt werden, indem die Schlüssel i mit ihrer Version $v_X(i)$ zu $S^*_X = \{i \circ v_X(i) \mid i \in S_X\}$ konkateniert werden. Die dadurch gewonnenen eindimensionalen Mengen können wie in 2.4 beschriebenen für die *Set Reconciliation* verwendet werden.

Die Überführung in die eindimensionale Form durch Konkatenation ist mit einem Fehler behaftet. Die Menge $S^*_{\Delta X} \neq S_{\Delta X}$, weil die größer kleiner Relation in der konkatenierten Version nicht möglich ist. Demzufolge entspricht $S^*_{\Delta A} = \{i \mid i \in S_A \cap S_B \wedge v_A(i) \neq v_B(i)\} = S^*_{\Delta B} = S_A \Delta S_B = S_{\Delta A} \cup S_{\Delta B}$, wodurch die allgemeine Differenz bekannt ist, jedoch nicht welche Elemente der Differenz von A nach B übertragen werden müssen *et vice versa*.

2.6.2 Kommunikationsstrategien

Eine Kommunikationsstrategie stellt sicher, dass mit minimalem Nachrichtenaufwand alle Knoten miteinander synchronisiert werden, d.h. als Partner für die *Set Reconciliation* ausgewählt werden. Der Replikationsgrad wird mit $r \in \mathbb{N} \setminus \{0\}$ bezeichnet und ohne Beschränkung der Allgemeinheit wird davon ausgegangen, dass sich jede der r Kopien auf einem separaten Knoten befindet.

In einem System mit N Knoten und einem Replikationsgrad r existieren mit einer selektiven Platzierungsstrategie $(N - r) + 1$ Gruppen von Knoten, die gemeinsame Kopien besitzen. Die Knoten innerhalb dieser Gruppen müssen alle periodisch miteinander synchronisiert werden, sodass alle veralteten Kopien im System gefunden werden können.

Die Tabelle 2.3 zeigt geläufige Kommunikationsstrategien, die bei der Replikaktualisierung verwendet wurden. Hierbei ist die Anzahl der Nachrichten äquivalent zu den durchgeführten Synchronisationsvorgängen.

Die Broadcast Methode ist die einfachste, in der jeder der r Knoten einer Gruppe G sich mit den restlichen $r - 1$ Knoten seiner Gruppe synchronisiert. Diese Methode verursacht die Größten Kosten von $r * (r - 1)$ Nachrichten und Synchronisationsvorgängen.

Methode	#Nachrichten	Verwendet in
Broadcast	r(r-1)	-
Master-Slave	r-1	OpenDHT [77]
Chain	r	Irislog [71]
Epidemisch	flexibel	Scuttlebutt [95]

Tabelle 2.3: Kommunikationsstrategien und ihre Kosten

Die Master-Slave Strategie basiert auf der Deklarierung eines Primärknoten p. Ist ein Knoten von G als Primärknoten deklariert, synchronisiert dieser sich mit den verbleibenden $r-1$ Knoten. Auf diese Weise werden nur $r-1$ Nachrichten bzw. Synchronisationen durchgeführt. Diese Methode verteilt die Synchronisationslast dafür nicht gleichmäßig in der Gruppe.

Die Chain Replication bildet eine Kette aus r Knoten über die Replikengruppe $G = \{r_0, r_1, r_2, r_3\}$. Jede Kopie r_i, $i = 0, \ldots, 3$ synchronisiert sich ausschließlich mit seinem Nachfolger $r_{(i+1) \mod r}$ wodurch die Synchronisationslast gleichmäßig verteilt wird und nur r-fach durchgeführt wird.

Epidemische Algorithmen stellen eine effiziente Lösung für derartige Kommunikationsprobleme dar. Das epidemische *Anti-Entropie* Protokoll wird auf jedem Knoten periodisch ausgeführt, bei dem ein zufälliger Partnerknoten kontaktiert wird. Mit dieser Methode werden nur $\log_2(r) + \ln(r) + O(1)$ Perioden benötigt, damit alle r Knoten mit hoher Wahrscheinlichkeit einmal kontaktiert wurden [74].

2.6.3 Existierende Lösungen

OpenDHT [77] von *Rhea S. C.* verwendet ein *replica synchronization* Algorithmus der fast identisch ist zu *Cates J. local maintenance algorithm* aus DHash [20]. Beide Algorithmen basieren auf *Merkle-Trees* zur Synchronisation der Knotendatenbanken. Weiterhin verwenden beide DHTs das *successor placement* sowie die *Master-Slave* Kommunikationsstrategie zur Synchronisation. Der *local maintenance algorithm* von *Cates J.* wurde zur Repliken-Regeneration verwendet, indem die Knoten des *Merkle-Tree* einen Hash über die Schüssel bilden. In *OpenDHT* hingegen wird der Hash über das Schlüssel-Wert-Paar gebildet damit Versionsdifferenzen identifiziert werden können. Eine weitere Anpassung zur Effektivitätssteigerung ist die Sortierung des *Merkle-Tree* nach Zeitstempel. Dadurch liegen in *OpenDHT* die asynchronen Einträge immer im rechten Teilbaum, wodurch der linke Teilbaum möglichst übersprungen werden kann.

IrisLog [71] verwendet eine periodische *data refresh* Funktion auf Schlüsselebene. Die Kopien eines r-fach repliziertes Datum bilden eine kettenförmige Gruppe (*Chain*), in der jede Kopie in einem 30 Sekunden Intervall die Version seiner Vorgängerkopie prüft. Die Kopie mit dem kleinsten Schlüssel prüft die Version aller anderen Kopien. Werden bei einer solchen Prüfung Differenzen festgestellt, wird die ältere Version aktualisiert. Durch die Verwendung einer kettenförmigen Anordnung der Kopien wird vermieden, dass jede Kopie mit jedem anderen kommunizieren muss. Dennoch entstehen $2(r-1)$ Nachrichten je Replikengruppe je Intervall. Wird die *data refresh* Methode je Replikengruppe verwendet, skaliert der Ansatz sehr schlecht und ist ineffizient. Die Effizienz ließe sich jedoch einfach steigern, durch die Verwendung von *Set Reconciliation* Algorithmen.

Scuttlebutt [95] verwendet ein *Anti-Entropie* Protokoll für die Aktualisierung veralteter Kopien. Da die Netzwerkbandbreite beschränkt ist und die *gossiping* Paketgröße nach oben beschränkt werden sollte, können nicht immer alle Differenzen in einer Periode aufgelöst werden. Um dennoch so effektiv wie Möglich die Entropie zu verringern, ist ein Auswahlkriterium nötig,

um die am weitesten zurückliegenden Kopien und Knoten auswählen zu können. Grundlage hierfür ist eine systemweite, Schlüssel unabhängige strenge totale Ordnung über die Versionsnummern, sodass nach jeder Änderung die betroffene replizierte Kopie die höchste Versionsnummer trägt. *Scuttlebutt* besitzt zwei Sortierungsoptionen. In *scuttle-breadth* werden Gossipping-Knoten fair ausgewählt, wodurch mit allen Knoten ähnlich häufig Synchronisationen durchgeführt werden. In *scuttle-depth* werden jene Knoten bevorzugt die die meisten Versionsdifferenzen besitzen. Bei der Synchronisation werden die zu synchronisierenden Daten nach der Höhe ihrer Versionsunterschiede sortiert, sodass am weitesten zurückliegende Kopien bevorzugt werden. Die Evaluation zeigt, das ein Bias bei der Auswahl des Gossippingpartners die Versions-Entropie wesentlich stärker verringert als eine faire Auswahl. Daher ist *scuttle-depth* der *scuttle-breadth* Methode überlegen.

In Dynamo [28] existieren zwei Mechanismen um die Anzahl asynchroner bzw. veralteter Kopien zu verringern. Der erste Mechanismus basiert auf *Merkle-Trees* wobei die Kommunikationsstrategie nicht angegeben ist. Aufgrund der Verwendung des *successor-placement* und der Benennung eines *coordinator* Knotens kann von der *Master-Slave* Kommunikationsstrategie ausgegangen werden. Der zweite Mechanismus wird *read repair* genannt und entspricht einer *On-Demand* Replikaktualisierung. So wird bei jedem Lese-Vorgang auf ein Schlüssel-Wert-Paar eine *state machine* erstellt, die über die Anfrageantwort hinaus existiert. So stoppt die *state machine* nicht nach der Rückgabe des Wertes, sondern wartet auf verspätet antwortende Kopien und aktualisiert diese falls nötig. Dadurch werden langsame Kopien im Betrieb aktualisiert, die dem *Merkle-Tree* basierenden Protokoll Aufwand ersparen.

2.6.4 Diskursion

Die Aktualisierung von Repliken ist in geographisch weit verteilten Systemen sinnvoll, um „nahe“ Kopien aktuell zu halten und die Leselatenz niedrig zu halten. Es wurde aufgezeigt, dass eine Lösung aus zwei Schichten besteht. Die untere Schicht dient der Bestimmung differierender Daten zwischen zwei Knoten durch Verwendung von *Set Reconciliation* Algorithmen. Diese können durch die im Abschnitt 2.6.1 gezeigten Anpassungen für die Erkennung von Versionsdifferenzen eingesetzt werden. Die obere Schicht implementiert eine Kommunikationsstrategie, die die Knotenpaare bestimmt, die eine Synchronisation mit Hilfe der unteren Schicht durchführen. Die Hauptaufgabe der Kommunikationsstrategie ist die Sicherstellung einer gleichmäßigen Verwendung aller Knoten als Synchronisationspaare, sowie eine Minimierung der Synchronisationen.

3 Repliken-Reparatur

In diesem Kapitel wird der Repliken-Reparatur-Service (RR-Service) vorgestellt. Dieser identifiziert und korrigiert veraltete und verlorene Repliken. Im Folgenden wird das Systemmodell sowie der detailierte Aufbau des RR-Serivce beschrieben.

3.1 System Modell

Eine DHT mit Chord-basiertem Routing und symmetrischer Replikation kann wie folgt formal beschrieben werden. Die Grundlage der DHT bildet ein Schlüsselraum, dessen Größe eine Konstante $N \in \mathbb{N}$ definiert. Der ringförmige Schlüsselraum ist definiert als $I := \mathbb{N} \bmod N = \{0, \cdots, N-1\}$. Jeder partizipierende Knoten k erhält einen Schlüssel $i \in I$, der seine Position im Ring definiert. Die Schlüsselmenge $K \subset I$ enthält die Schlüssel aller verfügbaren Knoten. Aufgrund der Verwendung der DHT als Speichersystem und einem frei wählbaren N wird $|K| < N$ vorausgesetzt. Die *Successor*-Funktion $succ : I \to K$ bildet jeden Schlüssel $i \in I$ auf genau einen Knoten $k \in K$ ab, der i im Uhrzeigersinn folgt.

$$succ(i) := i + \min(\{(k-i) + N \bmod N \mid k \in K\}) \bmod N \tag{3.1}$$

Die Menge der Schlüssel, die einem Knoten k durch die *Successor*-Funktion zugewiesen wird, bildet den Zuständigkeitsbereich $Z_k := \{i \mid i \in I \wedge succ(i) = k\}$.

Die symmetrische Replikation wird für einen Replikationsgrad $r \in \mathbb{N}^+, r < N$ mit Hilfe der Assoziation eines jeden Schlüssels mit r anderen umgesetzt. Dadurch wird der Schlüsselraum I in $\frac{N}{r}$ Äquivalenzklassen partioniert. Diese Aufteilung des Schlüsselraums in Äquivalenzklassen hat den Vorteil, dass jede Kopie einen eindeutigen Platz über die *succ*-Funktion besitzt und $I = \bigcup_{\forall k \in K} Z_k$. In anderen Platzierungsstrategien wird der Zuständigkeitsbereich Z_k jedes Knotens vergrößert, sodass $I \subset \bigcup_{\forall k \in K} Z_k$.

Die Assoziation jedes Schlüssels i mit $\{1, \cdots, r\} = R$ anderen wird durch die Funktion $a : I \times R \to I$ definiert:

$$a(i, x) := i + \left((x-1) * \frac{N}{r}\right) \bmod N \tag{3.2}$$

Die Äquivalenzklasse $A_i := \{a(i,x) \mid x \in R\}$ enthält alle mit i assoziierten Schlüssel des Rings und

wird Replikengruppe genannt. Die Assoziationsfunktion $a(i,x)$ teilt den Ring I in r ringförmige Restklassen $\Gamma(x) = \bigcup_{\forall i \in I} a(\min(A_i),x), x \in R$ ein, die im folgenden Replikenquadrant genannt werden. Zwei beliebige Knoten k_1, k_2 die mindestens einen Schlüssel der gleichen Äquivalenzklasse besitzen, stehen in der Relation $\Theta \subseteq K \times K$.

$$k_1 \Theta k_2 :\Leftrightarrow \exists i \in Z_{k_1} : ((A_i \setminus \{i\}) \cap Z_{k_2} \neq \emptyset) \tag{3.3}$$

DHTs verwalten Schlüssel-Wert-Paare (i, val), deren Schlüssel $i = h(y)$ durch eine Hashfunktion $h(y) \rightarrow I$ bestimmt wird, wobei y eine beliebige Zeichenfolge ist. Eine DHT ist beispielsweise *immutable*, wenn $i = h(val)$, weil jedem Wert genau ein Schlüssel zugeordnet wird und eine Wertänderung einen anderen Schlüssel generiert. Im Folgenden werden ohne Beschränkung der Allgemeinheit *mutable* DHTs betrachtet. *Mutable* DHTs verwenden eine andere Art der Schlüsselgenerierung. Beispielsweise kann ein benutzerdefinierter Name verwendet werden, dessen Hash i generiert. Die Verwendung eines Quorum-basierten Zugriffs auf die replizierten Schlüssel-Wert-Paare verlangt weiterhin nach einer Versionsnummer, damit neuere Kopien erkannt werden können. Quorum-basierte *mutable* DHTs verwalten dadurch geordnete Tripel der Form $(i, val, version), i \in I, version \in \mathbb{N}$, val ist ein beliebiger Wert. Im Folgenden werden die Schlüssel-Wert-Version Tripel (KVV) genannt und $version(i)$ wird als Notation für die dritte Komponente des Tripels mit Schlüssel i verwendet.

Die Belegung $\beta(I^*)$ eines Intervals $I^* \subseteq I$ enthält alle Schlüssel aus I^* die mit einem Wert belegt sind $\beta(I^*) := \{i \mid i \in I^* \wedge i \texttt{ ist belegt}\}$. Die Belegung eines Knotens k ist damit $\beta(Z_k) \subseteq Z_k$.

Ein *mutable Key-Value-Store* kann auf diesem DHT Modell mit Hilfe der Operationen $write(i, val)$ und $read(i)$ erstellt werden. Der Zugriff auf ein Datum mit dem Schlüssel i erfolgt über ein Quorum, in dem alle für $i^* \in A_i$ zuständigen Knoten abstimmungsberechtigt sind. Ohne Beschränkung der Allgemeinheit befindet sich jede Kopie auf einem separaten Knoten $|A_i| = |\dot{\bigcup}_{i^* \in A_i} \{succ(i^*)\}|$. Der Replikationsgrad r gibt die Anzahl der abstimmungsberechtigten Knoten an, sowie die notwendige Mehrheit von $\lceil \frac{r+1}{2} \rceil$ zustimmenden Knoten für ein *Majority Quorum*.

Die Operation $write(i, val)$ schreibt nach der Bildung des Quorums $Q \subseteq \{succ(i^*) \mid i^* \in A_i\}$ den Wert (i_a, val) auf allen Quorumteilnehmern $k_q \in Q$ unter den jeweiligen Schlüssel $i_a \in A_i \wedge i_a \in Z_{k_q}$. Neu hinzugefügte Werte besitzen die Version 0 und geänderte Werte erhalten die inkrementierte höchste Versionsnummer der Mehrheit $\text{version}(i) = \max(\cup_{i^* \in A_i \cap (\cup_{k \in Q} \beta(Z_k))} \{\text{version}(i^*)\}) + 1$. Die Operation $read(i)$ gibt nach der erfolgreichen Quorumbildung das KVV-Tripel mit der höchsten Version zurück.

Die Funktion $Upd(A, B) : (K \times K) \rightarrow I$ gibt die Menge der Schlüssel des Knotens A an, deren Wert in einer älteren Version vorliegt als der mit ihm assoziierte aus B.

$$Upd(A,B) := \{i \in \beta(Z_A) \mid \text{version}(i) < \max(\text{version}(A_i \subset \beta(Z_B)))\} \tag{3.4}$$

Die Funktion $Reg(A, B) : (K \times K) \rightarrow I$ gibt die Menge der nicht belegten Schlüssel von A wieder,

deren assoziierter Schlüssel in B belegt ist.

$$Reg(A, B) := \{i \in Z_A \mid i \notin \beta(Z_A) \wedge A_i \subset \beta(Z_B) \neq \emptyset\} \tag{3.5}$$

Die Menge $\Delta_A = Upd(A,B) \cup Reg(A,B)$ enthält die Schlüssel aller KVV-Tripel die A fehlen oder veraltet sind. Das Ziel des RR-Service ist die systemweite Minimierung von $Upd(A,B)$ und $Reg(A,B)$ für alle $A,B \in K \wedge A\Theta B$.

3.2 Architektur

Der RR-Service ist als knoteninternes *Singleton* [37] konzipiert. Die Gesamtheit der RR-Services bilden die Repliken-Reparatur-Schicht der DHT. Kommuniziert der RR-Service von Knoten A mit dem von Knoten B wird A Initiator und B Partner genannt. Der RR-Service besteht aus drei Komponenten:

Anti-Entropie Protokoll: Jeder Knoten wählt periodisch ein in Θ Relation stehender Knoten zufällig zur Synchronisation aus.

Set Reconciliation Dienst (RC): Bestimmt die Menge Δ.

Aktualisierungs-Dienst (RS): Überträgt für alle $i \in \Delta$ die KVV-Tripel von A nach B sowie Δ_A von B nach A.

Erhält der RR-Service eine *sync* Anfrage, so erstellt dieser eine neue Instanz des RC-Dienstes und übergibt diesem die Details der Anfrage. Der RC-Dienst beginnt dann mit der Bestimmung der differierenden Daten mit dem übergebenen Knoten oder mit einem Zufälligen. Nachdem die Differenz mit dem Partner bestimmt wurde, wird diese dem RR-Service übergeben mit der Anfrage nach Aktualisierung. Die Aktualisierungsanfrage wird dann an eine neue Instanz des RS-Dienstes gegeben, der die übergebenen KVV-Tripel versucht zu aktualisieren bzw. regenerieren.

Die *sync*-Anfragen können entweder manuell dem RR-Service übermittelt werden oder werden periodisch vom Anti-Entropie Protokoll generiert. Das Anti-Entropie Protokoll stellt eine systemweite Regeneration und Aktualisierung sicher.

Im Folgenden werden der *Set Reconciliation* Dienst, der Aktualisierungsdienst sowie das Anti-Entropie Protokoll detailliert beschrieben.

3.3 *Set Reconciliation* Dienst

Der *Set Reconciliation* Dienst ermittelt Δ mit Hilfe eines *Set Reconciliation* Algorithmus. Dabei wird jede Anfrage in einem separaten Prozess bearbeitet, sodass multiple Anfragen parallel verarbeitet

werden können. Um veraltete und verlorene Kopien identifizieren zu können, werden die KVV-Tripel, wie in Abschnitt 2.6.1 beschrieben, als Konkatenation „$i\#\text{version}(i)$“ der *Set Reconciliation* Datenstruktur hinzugefügt.

Die *Set Reconciliation* Datenstrukturen werden jeweils über ein kontinuierliches Interval $I^* \subseteq Z_k$ eines Knoten k erstellt und repräsentieren in komprimierter Form dessen Belegung $\beta(I^*)$. Damit ein Vergleich der Daten zwischen zwei Knoten A,B möglich ist, muss $A\Theta B$ gelten.

Durch die symmetrische Replikation besitzt jedes Element einer Replikengruppe einen eindeutigen Schlüssel, $a(i,1) \neq a(i,2)$. Dementsprechend können die Tripel unterschiedlicher Knoten $(a(i,1),val,\text{version}(a(i,1))) \neq (a(i,2),val,\text{version}(a(i,2)))$ nicht einfach verglichen werden. Eine triviale Lösung ist die Abbildung aller KVV-Tripel auf einen gemeinsamen Schlüssel in $\Gamma(1)$ mittels $(\min(A_i), val, \text{version}(i)), \forall i \in Z_A$ äquivalent für B. Diese Abbildung kann bei aufzählungsbasierten *Set Reconciliation* Algorithmen, wie dem *Bloom Filter*, problemlos verwendet werden. Suchbasierte Algorihtmen, wie der *Merkle-Tree*, vergleichen Signaturen von Intervallen. Idealerweise existiert dafür ein auf beiden Knoten identisches kontinuierliches Intervall. Bei der trivialen Lösung $Z_A \cap^{R_t} Z_B = \{\min(A_i) \mid i \in G\} = S$ wird das gemeinsame Interval in $\Gamma(1)$ definiert. Dieses Intervall S ist kontinuierlich im Restklassen Ring $\Gamma(1)$, jedoch nicht immer kontinuierlich in I. In Scalaris existiert kein Modul zur Verwendung benutzerdefinierter Ringe, wodurch das gemeinsame Interval im DHT-Schlüsselraum I kontinuierlich sein muss.

Ein in I kontinuierliches gemeinsames Intervall $Z_A \cap^R Z_B$ wird daher mit Hilfe der Relation $\cap^R \subseteq Z_A \times Z_B$ berechnet. Sei $G = \{i \mid i \in Z_A \wedge A_i \cap Z_B \neq \emptyset\}$ die Menge der Schlüssel von A die mit Schlüsseln aus B assoziiert werden.

$$Z_{AB} = Z_A \cap^R Z_B := \{\min(\{i^* \mid i^* \in A_i \wedge i^* \geq a(\min(G),1)\}) \mid i \in G\} \tag{3.6}$$

Das gemeinsame Intervall beginnt in der ersten Restklasse $\Gamma(1)$ und endet in einer nachfolgenden Restklasse $\Gamma(z), z \in R \setminus \{1\}$.

Nach der Berechnung des Intervalls Z_{AB} werden die KVV-Tripel $(i, val, version), i \in Z_A \vee i \in Z_B$ als $(i_a,val,version), i_a \in Z_{AB} \wedge i_a \in A_i$ in die *Set Reconciliation* Datenstruktur eingefügt. Diese Vereinheitlichung der in Assoziation stehenden Schlüssel von A und B ermöglicht die Vergleichbarkeit der Datenstrukturen zur Identifikation veralteter und fehlender Tripel.

Die Erstellung der *Set Reconciliation* Datenstruktur über Z_{AB} anstatt Z_A ist beim *Merkle-Tree* notwendig, beim *Bloom Filter* sinnvoll. Ist $|Z_{AB}| \leq |Z_A|$ müssen weniger Elemente in den *Bloom Filter* eingefügt werden. Dies verringert die Anzahl an Hashoperationen bei der Erstellung und minimiert die Bitgröße. Die Effizienz des *Bloom Filters* wird gesteigert, da B für die Elemente in $Z_{AA} \setminus Z_{AB}$ nicht zuständig ist und diese dadurch im *Bloom Filter* überflüssig sind.

Nachfolgend werden die drei umgesetzten *Set Reconciliation* Algorithmen *Bloom Filter*, *Merkle-Tree* und *ART*, welche bereits in 2.4 vorgestellt wurden, in ihrer Umsetzung beschrieben.

3.3.1 *Bloom Filter*

Der *Bloom Filter* ist als $\lceil \frac{m}{8} \rceil$ Byte großes Erlang BINARY implementiert. Die in der Konfiguration fest angegebene Zielfehlerwahrscheinlichkeit dient als Basis für die Bestimmung von m nach Formel 2.6 sowie die Anzahl der benötigten Hashfunktionen K nach Formel 2.5. Damit wird für einen beliebigen Knoten x abhängig von $|\beta(Z_x)|$ der *Bloom Filter* in Größe und Berechnungsaufwand angepasst.

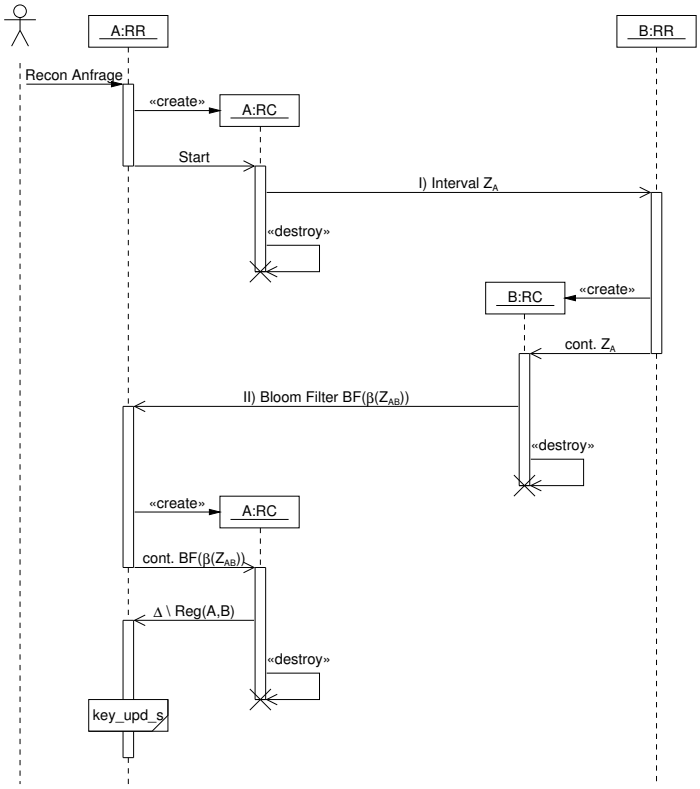


Abbildung 3.1: *Bloom Filter* Protokoll für die Synchronisation der Knoten A und B

Die K unabhängigen Hashfunktionen werden wie in Formel 2.7 angegeben durch die Kombination zweier unabhängiger Hashfunktionen h_1,h_2 generiert. Für h_1 wurde ADLER32 [30] verwendet sowie für h_2 MD5 [78].

Die Implementationsvariante als partionierten *Bloom Filter*, welcher das m Bit-Array in K Partitionen der Größe $\frac{m}{K}$ teilt und jede Hashfunktion in genau eine Partition abbildet, wurde nicht verwendet. Diese Variante bietet Parallelisierungsmöglichkeiten zur Verringerung der Konstruktionszeit. Es wurde jedoch in [50] gezeigt, dass dies zu einer erhöhten Fehlerwahrscheinlichkeit führt, wodurch die Standardvariante der Implementation verwendet wurde.

Die Abbildung 3.1 zeigt als UML Sequenzdiagramm [1] den Ablauf der Differenzbestimmung zwischen dem Initiator A und dem Partner B mit einem *Bloom Filter*. Wie in der Abbildung

dargestellt beginnt der Algorithmus mit einer „Recon“-Anfrage an den RR-Service von A. Dieser erstellt daraufhin eine Instanz des RC-Dienstes und leitet diesem die Anfrage weiter, wodurch der RR-Service neue Anfragen entgegennehmen kann. Der RC-Dienst beginnt bei dem Empfang der Anfrage mit der Bestimmung von Δ indem er sein Intervall Z_A an B sendet und sich beendet. Der RR-Dienst von B nimmt die „continue“-Anfrage entgegen und leitet diese an eine neu erstellte RC-Dienst-Instanz B:RC weiter. Falls nicht $A\Theta B$ gilt, ist $Z_{AB} = \emptyset$ und B:RC beendet sich, andernfalls wird der *Bloom Filter* $BF(\beta(Z_{AB})) = BF_1$ über das gemeinsame Intervall erstellt und an A gesendet. Anschließend beendet sich B:RC. Der RR-Service von A leitet BF_1 an eine neue Instanz des RC-Dienst weiter, der die Differenz $\Delta = \{i \mid i \in \beta(Z_{AB}) \wedge i \in BF_1\}$ berechnet. Das Δ wird an den RR-Serivce von A übergeben, der dies dem Aktualisierungsdienst zur Übertragung übermittelt.

Das durch den *Bloom Filter* ermittelte $\Delta = \Delta_A \cup \Delta_B \setminus Reg(A,B)$. Die Menge der Tripel welche A fehlen $Reg(A,B)$, kann nicht erkannt werden. Dies folgt aus der Art der Differenzberechnung bei der A für alle $i \in \beta(Z_{AB})$ prüft, ob diese in BF_1 enthalten sind. Die fehlenden Tripel von A sind nicht in $\beta(Z_{AB}) \subset \beta(Z_A)$ enthalten und können somit nicht geprüft werden. Wie in 2.6.1 beschrieben, ist die Unterscheidung zwischen $Upd(A,B)$ und $Upd(B,A)$ aus Δ nicht möglich. Die Auflösung des ermittelten Δ in $Upd(A,B)$ und $Upd(B,A)$ ist die Aufgabe des Aktualisierungsdienstes und wird in 3.4 beschrieben.

Aufgrund der Terminierung jedes RC-Prozesses nach einer rein lokalen Abarbeitung einer Abfrage ist das *Bloom Filter* Protokoll zustandslos. Das Protokoll gehört zu den *single-round* Protokollen, weil nur die numerierten Nachrichten I, II aus Abbildung 3.1 knotenübergreifend sind und eine Mitteilungsrunde darstellen. Die Kommunikation zwischen A:RR und A:RC sowie B:RR und B:RC ist knotenintern und beansprucht keine Netzwerkresourcen.

3.3.2 *Merkle-Tree*

Der *Merkle-Tree* wurde als n-ärer unbalanzierter Baum mit Hilfe verketteter Listen realisiert. Der Baum repräsentiert das Intervall $I^* \subseteq Z_A$ des DHT-Knotens A und jeder Knoten k des Baumes repräsentiert ein Teilintervall $I_k \subseteq I^*$. Innere Knoten k_i enthalten eine Liste ihrer n-Kindknoten, die das Intervall I_{k_i} möglichst gleichmäßig disjunkt partitionieren. Blattknoten k_b besitzen eine *Bucket* genannte Liste die die Schlüssel von $\beta(I_{k_b})$ enthält.

Der Verzweigungsgrad n des *Merkle-Trees* sowie die *Bucket*-Größe b können flexibel angepasst werden, um die Knotenanzahl k sowie die Baumhöhe zu beeinflussen. Bei gleichmäßiger Datenverteilung ist die Baumhöhe $h \geq \log_n(k+1)$. Die Datendichte eines Intervalls beeinflusst die Konstruktion des *Merkle-Trees*. Demnach wird ein stark belegtes Intervall stärker im Baum verzweigt als eines mit geringer Belegung. Der Vergleich zweier *Merkle-Trees* benötigt je Baumebene eine Nachricht, wodurch der Nachrichtenaufwand proportional zur Baumhöhe ist und über diese optimiert werden kann. Der Vergleich zweier Blätter führt zur vollständigen Übertragung des *Bucket*, wodurch die *Bucket*-Größe zur Übertragung unnötiger Daten führt.

Die Erstellung des *Merkle-Trees* wird in Algorithmus 2 dargestellt. Demzufolge wird der *Merkle-Tree* durch den Aufruf MerkleTree(Z_{AB}) über das gemeinsame Intervall der Knoten A,B erstellt. Der Algorithmus erstellt ein Blatt, wenn die Belegung des Teilintervalls kleiner ist als die *Bucket*-Größe. Andernfalls wird ein innerer Knoten erstellt, deren Kindknoten durch die Rekursion der Funktion erstellt werden.

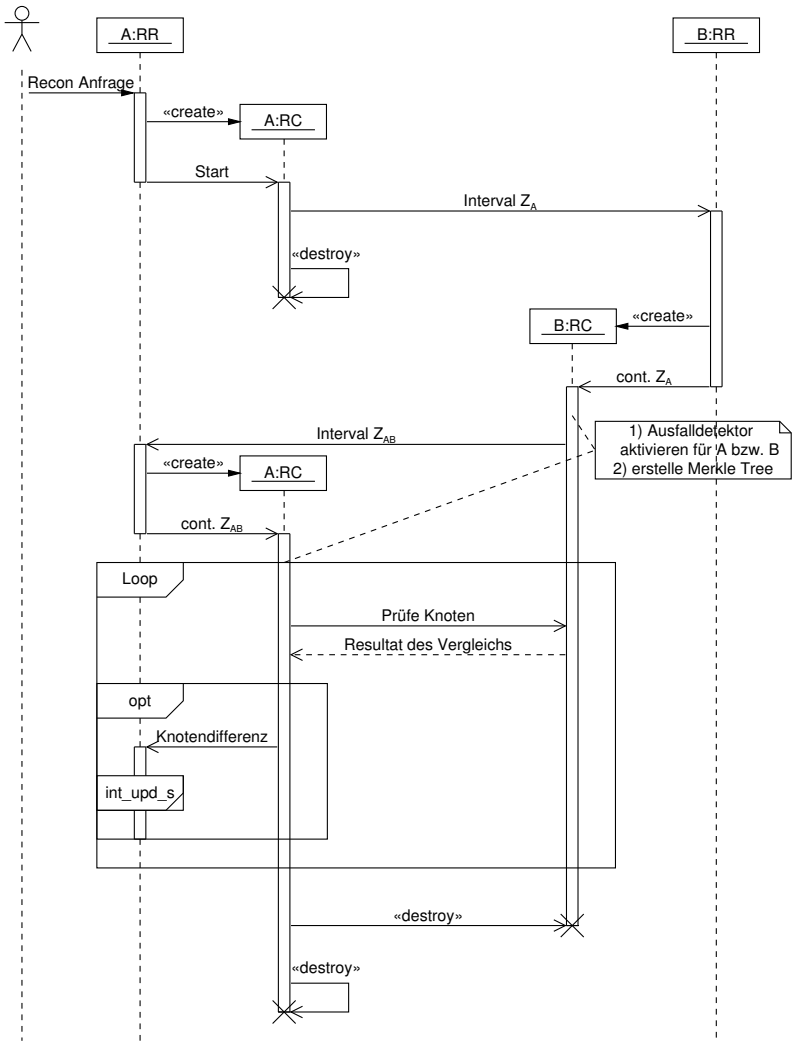


Abbildung 3.2: *Merkle-Tree* Protokoll zur Differenzbestimmung zwischen Knoten A und B

Nach der Konstruktion des Baumes werden die Signaturen für jeden Knoten mit Hilfe zweier Hashfunktionen berechnet. Die Signatur der Blätter entspricht der 160 Bit SHA-1 [33] Prüfsumme, die über alle „i#version“ Paare des *Buckets* berechnet wird. Die Signatur der inneren Knoten wird, wie in [94] vorgeschlagen, mit Hilfe der XOR-Funktion über die Signaturen der Kindknoten erstellt.

Die Bestimmung von Δ_A und Δ_B der Knoten A, B erfolgt nach der in Abbildung 3.2 angegebe-

Algorithmus 2 *Merkle-Tree* Erstellung

```
function MerkleTree(I*)
    if |β(I*)| ≤ b then
        return (Blatt, I*, Bucket(β(I*)))
    else
        [a,b] ← I*
        for t = 1 to n do
            I*_t ← [a + (t − 1)(b−a)/n, b − (n − t)(b−a)/n]
        end for
        return (innen, I*, [MerkleTree(I*_1), ..., MerkleTree(I*_n)])
    end if
end function
```

nen Interaktionsabfolge. Das Protokoll beginnt äquivalent zum *Bloom Filter* Protokoll, indem ein Austausch der Intervalle stattfindet, um das gemeinsame Intervall Z_{AB} zu berechnen. Anschließend erstellen beide Knoten einen *Merkle-Tree* über Z_{AB}. Die *Merkle-Tree* spezifische Differenzbestimmung beginnt im *loop* Fragment und entspricht dem in [20] angegebenen Ablauf. Zuerst wird die Signatur der Wurzel von A an B gesendet. Ist die Signatur mit der Wurzel von B identisch, ist der Abgleich abgeschlossen. Anderenfalls werden die Signaturen aller Kindknoten an B übermittelt, der daraufhin die Signaturen mit den eigenen vergleicht und das Ergebnis als geordnete Liste an A zurücksendet. Das Ergebnis eines Vergleichs ist *ok* falls die Signaturen übereinstimmen, andernfalls *fail*. Im *fail*-Fall wird zusätzlich übermittelt ob es ein innerer oder Blattknoten war. Dies ist wichtig, damit der Initiator bestimmen kann, ab wann ein Intervall dem Aktualisierungsdienst übergeben wird. Stimmen die Signaturen für ein Interval I^* nicht überein und ist I^* in A ein innerer Knoten, in B jedoch ein Blatt, so braucht A nicht die Signaturen der Kindknoten abgleichen und kann I^* dem Aktualisierungsdienst übergeben.

Das *Merkle-Tree* Protokoll ist zustandsbehaftet, weil sich der RC-Dienst auf B nicht nach dem Erhalt des Intervalls Z_A beendet. Dieser wartet nach der Übertragung an A bis B Knotensignaturen von A empfängt. Während der Übertragung der Signaturen einer Baumebene kann einer der beiden Knoten ausfallen, wodurch der verbleibende Knoten dauerhaft auf eine Antwort warten würde. Ein solcher verwaister Prozess kann mit Hilfe eines Ausfalldetektors vermieden werden. Der Prozess B:RC überwacht mit seinem Ausfalldetektor den Knoten A, der Prozess A:RC den Knoten B. Falls einer der beiden Knoten ausfällt, wird die RC-Instanz des anderen durch den Ausfalldetektor benachrichtigt und beendet sich. Im Falle eines fehlerfreien Abgleichs sendet der Initiator eine *destroy* Nachricht an B:RC, damit dieser über das Ende des Abgleichs informiert wird und sich beenden kann.

3.3.3 Approximate Reconcilication Tree (ART)

ART stellt, wie in 2.4.4 beschrieben, einen als *Bloom Filter* codierten *Merkle-Tree* dar. Die Erstellung von ART erfolgt in zwei Schritten. Im Ersten wird ein *Merkle-Tree* wie in 3.3.2 beschrieben über ein Intervall I^* erstellt. Im zweiten Schritt werden die Signaturen aller inneren Knoten einem *Bloom Filter* BF_I hinzugefügt sowie separat die Signaturen der Blätter einem weiteren BF_L. ART ist als Erlang Tupel (I^*, BF_I, BF_L) implementiert.

Das Protokoll für den Austausch der ART-Datenstruktur entspricht exakt dem des *Bloom Filter* und kann in Abbildung 3.1 betrachtet werden. Die Ermittlung der Differenz unterscheidet sich jedoch. Nach dem Empfang von ART erstellt A einen *Merkle-Tree* über das Intervall der empfangenen ART-Struktur $I^* = Z_{AB}$, falls noch immer $Z_{AB} \subseteq Z_A$ gilt. Andernfalls wird der Abgleich aufgrund einer Änderung des Zuständigkeitsbereichs abgebrochen. Nach der Erstellung des *Merkle-Tree* wird das in 3.3.2 beschriebene *Merkle-Tree* Vergleichsprotokoll verwendet, wobei die Signaturen nicht mehr per Nachricht an B gesendet werden, sondern ein *Bloom Filter* den Vergleich lokal durchführt.

Das in 2.4.4 beschriebene *correction level* wurde zusätzlich implementiert, um die Genauigkeit des Vergleichs eines *Merkle-Tree* Knotens mit dem jeweiligen *Bloom Filter* zu erhöhen. Das ermittelte Δ entspricht dem des *Bloom Filters* und ist $\Delta = \Delta_A \cup \Delta_B \setminus Reg(A,B)$, wodurch auch ART die fehlenden Repliken $Reg(A,B)$ auf A nicht identifizieren kann. Die Evaluation wird jedoch zeigen, dass in einigen Fällen ART das vollständige Δ bestimmen kann.

3.4 Aktualisierungsdienst

Der Aktualisierungsdienst (RS) regeneriert und aktualisiert alle ihm übermittelten Tripel. Anfragen an den RS-Dienst werden an den RR-Service gesendet, der eine neue Instanz des RS-Dienstes erstellt und diesem die Anfrage weiterleitet. Der RS-Dienst unterscheidet nur zu statistischen Zwecken zwischen Regeneration und Aktualisierung, da alle ihm übergebenen KVV-Tripel der Datenbank übermittelt werden. Die Datenbank aktualisiert bzw. regeneriert lokale Tripel nur dann, wenn die vom RS-Dienst übermittelten neuer sind oder kein lokales Tripel existiert.

Der RS-Dienst unterstützt vier Anfragearten von denen zwei in der Abbildung 3.3 als UML-Sequenzdiagramm abgebildet sind. Die Anfrage *key_upd* enthält eine KVV-Liste, die auf dem Empfänger aktualisiert werden soll. Bei dieser Anfrage übermittelt der RS-Dienst die KVV-Liste an die Knoten-Datenbank zur Aktualisierung und beendet sich. Die Anfrage *key_upd_s* enthält nur eine Liste von Schlüsseln sowie die Adresse eines Zielknotens. Der RS-Dienst lädt bei dieser Anfrage die Werte der übermittelten Schlüssel-Liste und bildet eine KVV-Liste die anschließend an den Zielknoten als *key_upd* Anfrage übermittelt wird. Die *key_upd* Anfrage verursacht nur knoteninterne Nachrichten wohingegen *key_upd_s* eine knotenübergreifende Nachricht produziert. Die dritte Anfrage *int_upd* stellt eine erweiterte Form von *key_upd* dar, und verarbeitet die übertragene KVV-Liste äquivalent. Zusätzlich zu der KVV-Liste wird jedoch ein Intervall übertragen, aus dem

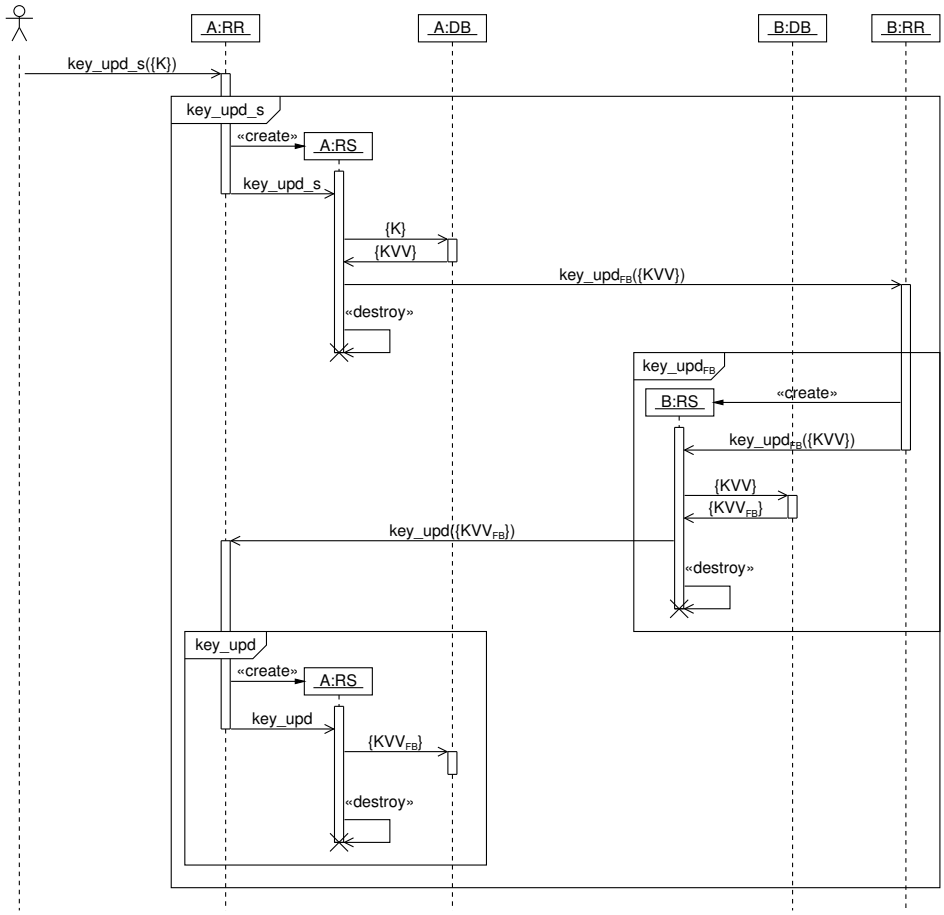


Abbildung 3.3: Aktualisierungsprotokoll für die Anfragen *key_upd_s* und *key_upd*

die KVV-Werte stammen. Dies wird benötigt um in einem zweiten Schritt die lokalen KVV-Werte aus dem Intervall mit der übertragenen Liste abzugleichen. Auf diese Art können fehlende Repliken des Anfrageabsenders identifiziert werden. Dies ist speziell für den *Merkle-Tree* notwendig um verlorene Repliken auf dem Initiator zu identifizieren. Die vierte Anfrage *int_upd_s* entspricht der *key_upd_s* Anfrage und sendet zusätzlich den Intervall der Werte mit.

Die Trennung des ermittelten Δ in $Upd(A,B)$ sowie $Upd(B,A)$ erfolgt mit Hilfe des *Feedback*-Mechanismus. Empfängt der Knoten B eine KVV-Liste als *key_upd* Anfrage, werden die Paare $Upd(B,A)$ sowie $Reg(B,A)$ aktualisiert. Die von der Datenbank als nicht aktualisiert zurückgegebenen Elemente FB entsprechen $\Delta \setminus (Upd(B,A) \cup Reg(B,A)) = Upd(A,B) = FB$ und werden als *Feedback* bezeichnet. Ist der Mechanismus aktiviert wird die FB-Liste in einer *key_upd* Anfrage an A zurück übermittelt. Dadurch erhält A Δ_A und kann diese Kopien aktualisieren bzw. regenerieren.

Die *Set Reconciliation* Protokolle verwenden, wie in Abbildung 3.1 und 3.2 dargestellt, die *key_upd_s* Anfrage mit B als Zielknoten und aktiviertem *Feedback*.

3.5 *Anti-Entropie* Protokoll

Das *Anti-Entropie* Protokoll [29] synchronisiert mit Hilfe des RC-Dienstes Schritt für Schritt alle Knoten des Systems. Dieses Verteilungsprotokoll ist skalierbarer, robuster und effizienter als alternative Protokolle, wie z.B *direct mailing* [29]. Das Ziel des Protokolls ist eine rundenbasierte Konvergenz aller fehlerhaften Repliken gegen 0 $\lim_{r\to\infty} Upd(A,B) \to 0 \wedge Reg(A,B) \to 0, \forall A,B \in K, A\theta B$.

Das *Anti-Entropie* Protokoll basiert auf dem epidemischen SI-Modell [69], dass die Ausbreitung ansteckender Krankheiten beschreibt. Dem Modell zufolge wird eine Population N in die zwei Gruppen, S für die Gesunden sowie I für die Kranken eingeteilt. Es gilt $N = S + I$, sodass jedes Individuum genau einer Gruppe angehört. In epidemischen Modellen werden gesunde Individuen durch die Kranken angesteckt. Die Anzahl der Ansteckungen hängt von der Anzahl der Kranken sowie der Anzahl der Kontakte ρ je Individuum je Runde ab. Das SI-Modell wurde für verteilte Systeme angepasst, sodass eine Aktualisierung eine Krankheit darstellt, die auf die nicht aktualisierten übertragen werden soll [29]. Im Folgenden entsprechen die Kranken I den aktuellen Kopien und die Gesunden S den veralteten Kopien.

Demers et. al [29] definieren drei Implementationsvarianten des Protokolls. Ohne Beschränkung der Allgemeinheit seien A und B zwei auf unterschiedlichen Knoten befindliche Tripel. Die drei Protokollvarianten sind wie folgt definiert, wenn A mit B Kontakt aufnimmt:

push: Falls $version(A) > version(B)$ wird A an B übertragen.

pull: Falls $version(A) < version(B)$ fordert A den Wert von B an.

push-pull: Bei der Kombination beider wird bei $version(A) > version(B)$ der Wert von A an B gesendet und bei $version(A) < version(B)$ der Wert von B angefordert.

Die *push*-Variante benötigt $\log_2(n) + \ln(n) + o(1)$ Runden, um die gesamte Population anzustecken [74]. Die *pull* Variante unterscheidet sich stark von der *push* Variante, wenn $|S| \leq |I|$. So wächst die Anzahl von I bei der *push*-Variante exponentiell bis $I = \frac{N}{2}$, danach fällt die Verbreitungsgeschwindigkeit auf $1 - \frac{1}{e}$ [74]. Dies folgt aus der sinkenden Wahrscheinlichkeit, dass eine aktuelle Kopie eine veraltete Kopie für den Kontakt auswählt. Die *pull*-Variante benötigt

Algorithmus 3 *Anti-Entropie* Protokoll für jeden Knoten a mit festem ρ

```
function ANTI-ENTROPIE(I_a)
    z ← zufälliger Wert aus ]0,1] ⊂ ℝ
    if z < ρ then
        b ← zufälliger Schlüssel aus I_a
        Δ = RR-Service.Reconcile(a,b)
        sende Δ zu b
    end if
end function
```

$\log_2(n) + O(\log(\log(n)))$ Runden bis alle aktualisiert wurden [29]. Nach [48] verringert sich dies nochmals bei der *push-pull* Variante auf $\log_3(n) + O(\log(\log(n)))$ Runden, wodurch diese Variante die höchste Verbreitungsgeschwindigkeit aufweist.

Das *Anti-Entropie* Protokoll ist in Algorithmus 3 dargestellt und wird von jedem Knoten der DHT periodisch ausgeführt. Es wurde mit Hilfes eines *Timer* umgesetzt, der auf jedem Knoten $k \in K$ aktiv ist und periodisch mit der Wahrscheinlichkeit $\rho \in]0,1] \subset \mathbb{R}$ RC-Anfragen an den lokalen RR-Service sendet. Die Kontaktwahrscheinlichkeit ρ verringert die Anzahl der Synchronisationen. Der Zielknoten für die generierten RC-Anfragen wird durch die Wahl eines zufälligen Schlüssel $i \in Z_k$ ermittelt. Sei $z \in A_i \setminus \{i\}$ ein zufälliger mit i assoziierter Schlüssel, so ist $succ(z) = B$ der Zielknoten. Nach der Differenzbestimmung durch den RC-Dienst ist Δ dem Knoten A bekannt, der dieses über die *key_upd_s* Anfrage an B versendet. Dieser Schritt entspricht der *push* Protokollvariante. Der RS-Dienst auf B aktualiert seine Kopien und bildet die *Feedback* Menge, jener KVV-Tripel, die auf A veraltet sind. Ist der *Feedback* Mechanismus aktiviert, wird die *Feedback* Menge an A zurückgesendet. Dieses Verfahren entspricht der effizienten *push-pull* Variante.

3.6 Diskussion

Der Repliken-Reparatur-Service (RR-Service) besteht aus zwei Diensten. Der *Set Reconciliation* Dienst (RC-Dienst) dient der Bestimmung der veralteten und fehlenden Repliken zweier Datenbanken. Der Aktualisierungsdienst (RS-Dienst) dient der Regeneration und Aktualisierung differierender Repliken, welche durch den RC-Dienst bestimmt wurden. Das Design des RR-Service erlaubt die parallele Verarbeitung von RC- sowie RS-Anfragen. Um eine systemweite Konvergenz der Anzahl veralteter und fehlender Repliken gegen Null sicherzustellen, generiert ein Anti-Entropie Protokoll periodisch RC-Anfragen.

Der RC-Dienst enthält drei verschiedene Algorithmen zur Differenzbestimmung. Der *Bloom Filter* benötigt nur eine Mitteilungsrunde zur Differenzbestimmung und tauscht lokalen Berechnungsaufwand gegen geringe Netzwerkkommunikation ein. Der *Merkle-Tree* benötigt bei gleichmäßiger Datenverteilung maximal h Mitteilungsrunden mit kleinen Nachrichten. Der dritte implementierte Algorithmus ist ART und kombiniert die Differenzsuche des *Merkle-Trees* mit der Übertragungsform des *Bloom Filter*. Dieser hybride Ansatz verbindet die Vorteile beider Methoden zu lasten einer verminderten Genauigkeit [18]. Algorithmen die auf *Bloom Filter* basieren, erkennen nur $Upd(A,B)$, $Upd(B,A)$ sowie $Reg(B,A)$ nicht aber $Reg(A,B)$. Die fehlende Erkennung von $Reg(A,B)$ führt jedoch lediglich zu einer Verringerung der Konvergenzgeschwindigkeit.

Der RS-Dienst übermittelt die vom RC-Dienst ermittelte Differenz Δ. Diese enthält nur die Schlüssel der differierenden KVV-Tripel, jedoch nicht welches Tripel auf welchem der beiden Knoten veraltet oder verloren ist. Die Trennung von Δ in $Upd(A,B)$ und $Upd(B,A)$ erfolgt durch Übertragung von Δ an B, der alle aktualisierten Tripel entfernt $\Delta \setminus (Upd(B,A) \cup Reg(B,A)) = Upd(A,B)$ und das Ergebnis als Feedback an A zurück sendet.

4 Evaluation

Die folgenden Abschnitte zeigen die Leistungsfähigkeit der im vorherigen Kapitel vorgestellten drei *Set Reconciliation* (SR) Algorithmen, *Bloom Filter*, *Merkle-Tree* und ART. Die Analyse erfolgt in zwei Schritten. Im ersten Schritt, der *Set Reconciliation* Analyse, werden die SR-Algorithmen einzeln bei der Synchronisation einer Datenmenge betrachtet. Im zweiten Schritt erfolgt die Analyse der Algorithmen in Scalaris.

In der *Set Reconciliation* Analyse werden die einzelnen Parameter der SR-Algorithmen untersucht sowie der Einfluss verschiedener Szenarien auf die Leistungsfähigkeit. Um die Kerneigenschaften ohne äußere Einflüsse beobachten zu können, wird diese Untersuchung zwischen zwei Datenbeständen mit jeweils bekannter Differenz durchgeführt. Dies entspricht dem Abgleich zweier einzelner DHT-Knoten. Diese Testumgebung unterliegt keinen von der Partnerwahl abhängigen Schwankungen und bildet eine feste Basis für den Vergleich der SR-Algorithmen.

Im zweiten Schritt wird das *Anti-Entropie* Protokoll mit den SR-Algorithmen in Scalaris analysiert. Hierzu wird eine vollständige DHT mit definiertem Startzustand erstellt und deren Entwicklung unter Verwendung der unterschiedlichen Algorithmen beobachtet. Diese Auswertung soll die im ersten Schritt gesammelten Erkenntnisse im praktischen Einsatz widerspiegeln.

Ziel der Evaluation ist der Vergleich der Leistungsfähigkeit der drei SR-Algorithmen in verschiedenen Umgebungen, welche in 4.1 Szenarien erläutert werden. Anschließend werden in 4.2 die Metriken definiert, mit denen die Leistungsfähigkeit der Algorithmen bewertet wird. Der Simulationsaufbau wird in 4.3 beschrieben. Das Kapitel schließt mit den Ergebnissen in 4.4, indem die SR-Algorithmen zunächst getrennt analysiert werden. Die Ergebnisse werden abgeschlossen mit einer Analyse und einem Vergleich der Algorithmen in Scalaris.

4.1 Szenarien

Die Algorithmen werden in verschiedenen Szenarien analysiert, um die Abhängigkeit der Leistungsfähigkeit von der Umgebung zu untersuchen. Hierfür wird die Auswirkung verschiedener Fehler-, Daten- und Knotenverteilungen beobachet.

Fehlertyp

In der Praxis kann davon ausgegangen werden, dass verlorene und veraltete Kopien zugleich in einem verteilten System vorliegen. Zur genaueren Analyse dieser einzelnen Fehlertypen werden

jeweils Datenbestände mit reinen oder gemischten Fehlern betrachtet. Die $|\Delta|$ fehlerhaften Kopien sind veraltet bei Typ UPD und verloren bei REG. Der gemischte und praktische Fall wird MIX genannt. In diesem besteht Δ zu gleichen Teilen aus veralteten sowie verlorenen Kopien.

Fehlerverteilung

Die Fehlerverteilung bestimmt die Replikengruppen aus $\beta(\Gamma(1))$, welche eine fehlerhafte Kopie enthalten sollen. Bei der GLEICHMÄSSIGEN Verteilung von $|\Delta|$ fehlerhaften Kopien wird im Abstand von $\frac{|\beta(\Gamma(1))|}{|\Delta|}$ ein Schlüssel $i \in \beta(\Gamma(1))$ ausgewählt. Anschließend wird zufällig eine Kopie aus der Replikengruppe A_i gezogen, die einen durch den Fehlertyp bestimmten Fehler enthält. Die zweite Fehlerverteilung ist die BINOMIALVERTEILUNG mit $p = 0{,}2$ o.B.d.A. Diese führt zu einer lokalen Häufung der Fehler innerhalb eines Intervalls.

Datenverteilung

DHTs besitzen im Normalfall eine gleichmäßige Verteilung der belegten Schlüssel in I. Dies wird durch die Generierung von neuen Schlüsseln mittels einer Hashfunktion beim Einfügen sichergestellt. Durch die Verwendung eines anderen Schlüsselraumes, wie zum Beispiel mit Chord$^{\#}$ [82], kann es jedoch zu einer Häufung von Daten in Bereichen des Schlüsselraumes kommen. Aus diesem Grund wird die Auswirkung einer GLEICHMÄSSIGEN und einer BINOMIALVERTEILUNG betrachtet.

Die Daten werden über das Intervall $\Gamma(1)$ verteilt. Durch die symmetrische Replikation liegt die Datenverteilung von $\Gamma(1)$ anschließend in allen Quadranten des Rings vor. Werden N Datensätze in $\Gamma(1)$ gleichmäßig verteilt, ergeben sich die belegten Schlüssel als $\beta(\Gamma(1)) = \{\frac{i}{N} * \Gamma(1) | i = 0 \ldots N\}$. Bei der zweiten Verteilung werden die Schlüssel in $\Gamma(1)$ BINOMIALVERTEILT mit $p = 0{,}2$ o.B.d.A.

Knotenverteilung

In praktischen DHTs sind die Knoten K häufig nach der zu erwartenden Last in Bereichen des Schlüsselraums oder nach sonstigen Richtlinien verteilt. Die Verteilung der Knoten hat jedoch Einfluss auf die Größe gemeinsamer Intervalle $Z_{AB}, A\Theta B, \forall A,B \in K$ und somit auf die potenzielle Menge der identifizierten fehlerhaften Kopien in einer Synchronisation. Die Knotenverteilung beeinflusst damit die Konvergenzgeschwindigkeit der Repliken-Reparatur. Daher werden die ZUFÄLLIGE und die GLEICHMÄSSIGE Knotenverteilung gegenübergestellt. Bei der gleichmäßigen Knotenverteilung wird von Null beginnend alle $\frac{|I|}{|K|}$ Schlüssel ein Knoten platziert. Damit sind alle gemeinsamen Intervalle Z_{AB} relativ gleich groß und jedes Teilintervall eines Knotens A wird mit drei Knoten geteilt, daher gilt $\forall A \in K : |\{Z_{AB} | B \in K, B\theta A, B \neq A\}| = 3$. Diese Knotenanordnung stellt sicher, dass jeder Knoten mit maximal drei Knoten eine Synchronisation durchführen muss, um seinen vollständigen Intervall zu synchronisieren. Bei einer *zufälligen* Knotenplatzierung müssen bis zu $K - 1$ Knoten kontaktiert werden, um das Intervall eines Knotens zu synchronisieren. Beispielsweise

muss der Knoten A mit dem Zuständigkeitsbereich $Z_A = \Gamma(1) \cup \Gamma(2) \cup \Gamma(3)$ sich mit $K-1$ Knoten synchronisieren, wenn diese mit ihrem Zuständigkeitsbereich in $\Gamma(4)$ liegen.

4.2 Metriken

Die Leistungsfähigkeit der SR-Algorithmen wird anhand der vier Metriken, Genauigkeit, Übertragungskosten, Mitteilungsrunden und Redundanz gemessen.

Genauigkeit

Die Genauigkeit ist die Anzahl fehlerhafter Kopien $|\Delta|$ in einer betrachteten replizierten Datenmenge N. Die Metrik befindet sich in ihrem Optimum bei $|\Delta| = 0$, wenn keine veralteten oder verlorenen Kopien mehr existieren. Die Entwicklung von $|\Delta|$ gibt Aufschluss über die Genauigkeit der Algorithmen.

Übertragungskosten

Die Übertragungskosten sind die Summe der übertragenen Daten des RC- und RS-Dienstes. Die vom RC-Dienst verursachten Kosten werden in Bytes gemessen und resultieren aus der Übertragung des *Bloom Filter* bzw. der Knotensignaturen des *Merkle-Trees*. Die Kosten des RS-Dienstes werden als Anzahl Übertragener KVV-Tripel gemessen, wobei angenommen wird, dass die Übertragungskosten eines KVV-Tripels zwischen allen Knoten identisch ist. Damit die Kosten beider Dienste in einer Metrik zusammengeführt werden können, werden die KVV-Tripel in eine Bytegröße überführt indem diese mit 10 multipliziert werden. Damit ist die Größe eines KVV-Tripels auf 10 Byte fixiert.

Redundanz

Die Redundanz ergänzt die Aussagekraft der Übertragungskosten in Bezug auf den RS-Dienst und zeigt den vom RS-Dienst verursachten *Overhead* (Mehraufwand). Diese wird definiert als Quotient aus der Anzahl der übertragenen KVV-Tripel zur Anzahl der gefundenen Differenzen $|\Delta|$. Die Redundanz stellt die KVV-Kosten der Erkennung einer fehlerhaften Kopie dar. Das Optimum der Redundanz ist im Wert 1 erreicht, wenn nur die KVV-Tripel übertragen wurden die in der Mengendifferenz enthalten waren. Ein Redundanzwert von 2 dagegen bedeutet, dass für jedes gefundene Tripel aus Δ zwei Tripel übermittelt wurden.

Mitteilungsrunden

Die vierte Metrik bildet die Anzahl benötigter Mitteilungsrunden der SR-Algorithmen. Diese Metrik spiegelt die sequenzielle Kommunikationshäufigkeit wider und ist damit ein Maß der Algorithmenlatenz. Die gesamten Mitteilungsrunden entsprechen hauptsächlich den Mitteilungsrunden des RC-Dienstes. Der RS-Dienst benötigt maximal eine Mitteilungsrunde, weil die ihm übermittelten

Differenzen in einer einzigen Nachricht übertragen werden können und dieser sein *feedback* in einer Nachricht zurückübermittelt. Im Falle des *Merkle-Trees* und ART entstehen eine Vielzahl von RS-Anfragen. Diese werden jedoch parallel gesendet und können auch als eine einzelne zusammengefasst werden. Sie spiegeln damit nicht die vom Algorithmus verursachten Latenzkosten wider, wodurch o.B.d.A. der RS-Dienst eine Mitteilungsrunde beansprucht.

4.3 Simulationsaufbau

Die Evaluation erfolgt in einem separaten Modul innerhalb von Scalaris, welches über die Kommandozeile bedient wird. Es ist in der Lage eine Scalaris-DHT nach den oben genannten Szenarien flexibel zu initialisieren und die Metriken für die jeweiligen Algorithmen zu messen. Die Messwerte einer jeden Analyse werden in zwei Dateien exportiert. Die erste enthält die Rohdaten jeder Ausführung, wohingegen die zweite die über 100 Ausführungen gemittelten Messwerte enthält. Alle gesammelten Messwerte sind auf einer CD der Arbeit beigefügt. In den Simulationen werden synthetische Daten verwendet. Ein veraltetes synthetisches KVV-Tripel für einen Schlüssel i ist definiert als $\{i,old,1\}$, sowie alle nicht veralteten Tripel als $\{i,new,2\}$ definiert sind.

Die SR-Analyse untersucht die Synchronisation einer Datenmenge zwischen zwei Knoten A,B. Hierfür wird ein Scalaris Ring mit vier gleichmäßig verteilten Knoten angelegt, sodass $K = \{i * \frac{|I|}{4} | i \in \{0 \ldots 3\}\}$. Die fehlerhaften Kopien werden bei diesem Aufbau ausschließlich in den Restklassen $\Gamma(1)$ und $\Gamma(3)$ erzeugt. Der Knoten A mit Schlüssel $\frac{1}{4}|I|$ besitzt hierbei den Zuständigkeitsbereich $Z_A = \Gamma(1)$ sowie der Knoten B den Schlüssel $\frac{3}{4}|I|$ mit dem Zuständigkeitsbereich $Z_B = \Gamma(3)$. Die Synchronisation der Knoten A,B ist äquivalent zum Vergleich zweier Datenmengen mit Replikationsgrad 2 und einer gegebenen Differenz Δ. Die Untersuchung der Leistungsfähigkeit der Algorithmen erfolgt zwischen A,B mit $Z_{AB} = \Gamma(1)$ auf der Datenmenge $N = |\beta(Z_{AB})| = 10000$. Der Knoten B ist der Initiator und leitet eine einzelne Synchronisation mit A ein, nach der die Metriken gemessen werden.

Die Analyse der Algorithmen in einer DHT im Abschnitt 4.4.4 erfolgt in einem Scalaris Ring mit 16 Knoten. Die Messung der Metriken erfolgt in Runden, die zwischen den einzelnen Algorithmen vergleichbar sind. Eine Runde stellt die Synchronisation eines Knotens mit einem zufällig gewählten Partner dar. In den Auswertungen beginnt der Knoten mit dem niedrigsten Schlüssel die erste Runde gefolgt von seinem Nachfolger, der die zweite Runde durchführt *et cetera*.

Alle Messwerte werden auf einer DHT mit definiertem Startzustand durchgeführt. Ohne Beschränkung der Allgemeinheit können *churn* und andere dynamische Ereignisse vernachlässigt werden, weil Dynamik nur zu einem veränderten Startzustand in der jeweiligen Runde führt.

4.4 Simulationsergebnisse

Die Ergebnisse der SR-Algorithmen sind in einzelne Abschnitte für jeden Algorithmus eingeteilt und beginnen mit der Vorstellung der Algorithmenparameter. Darauf folgt die Analyse der Auswirkungen der Parameter auf die Metriken der Algorithmen und die Bestimmung geeigneter Konfigurationen zur weiteren Analyse. Anschließend wird die Auswirkung einer binomialen Fehlerverteilung und Datenverteilung untersucht sowie abschließend die Skalierbarkeit des Algorithmus.

Im Folgenden werden jeweils Differenzgrößen $|\Delta|$ zwischen 0% und 10% von N betrachtet. Dieser Bereich ist für die meisten DHT Anwendungen interessant, die seltene Replikenfehler besitzen.

4.4.1 *Bloom Filter*

Algorithmus

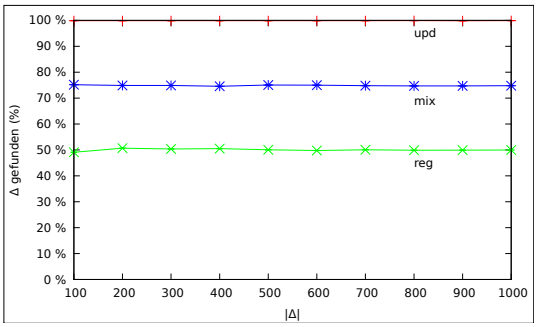
Der *Bloom Filter* besteht aus einem m Bit-Array sowie einer Anzahl von Hashfunktionen K. Diese Parameter können mit Hilfe der Anzahl der zu kodierenden Elemente N sowie der Fehlerwahrscheinlichkeit (Fpr) optimal nach den Formeln 2.6 und 2.5 gewählt werden. Die Synchronisation verläuft in zwei Schritten. Im ersten Schritt sendet der Initiator B seinen Zuständigkeitsbereich Z_B an A, welcher Z_{AB} bestimmt, einen *Bloom Filter* über $\beta(Z_{AB})$ erstellt und diesen an A sendet. Im zweiten Schritt ermittelt B die Differenz Δ und sendet diese an seinen RS-Dienst zur Auflösung (siehe Kapitel 3.3.1 und 3.4).

Genauigkeitsanalyse

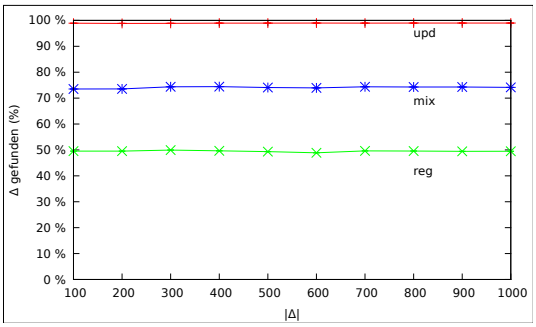
Zur Beobachtung der Auswirkungen verschiedener Fehlerwahrscheinlichkeiten wurden sechs verschiedene Fpr-Werte ausgewählt. Die Abbildungen 4.1 stellen die Genauigkeit dieser sechs Konfigurationen gegenüber. In diesem Szenario sind die Daten und Fehler in Z_{AB} gleichverteilt. Die Auswirkungen der Fpr-Werte sind gut aus den Abbildungen entnehmbar. Bei einer Fehlerwahrscheinlichkeit von 20% ($Fpr = 0{,}2$) in Abbbildung 4.1d werden entsprechend nur 80% der Differenzen gefunden. Dies ist Äquivalent für die Abbildungen 4.1a bis 4.1f. In der Abbildung 4.1b fehlen circa 10 veraltete Kopien weil $0{,}01 * 10000 = 10$ ist, wohingegen in Abbildung 4.1a nur eine veraltete Kopie nicht erkannt wird. Eine weitere Erhöhung der Genauigkeit des *Bloom Filter* hätte bei dieser Datenmenge keine Auswirkung mehr.

Die Genauigkeit bei dem Fehlertyp *reg* befindet sich in fast allen Fällen unter 50%. Die Ursache hierfür ist die in 3.3.1 beschriebene Einschränkung des *Bloom Filters*, der nur $\Delta \setminus Reg(B,A)$ identifiziert und die fehlenden Kopien des Initiators nicht auflöst. In Abbildung 4.1a liegt der prozentuale Anteil anfangs einmal leicht über 50%. Dies ist höchstwahrscheinlich auf die Fehlerverteilung zurückzuführen, die beim Münzwurf die fehlenden Kopien nicht perfekt gleichmäßig zwischen A und B verteilt hat.

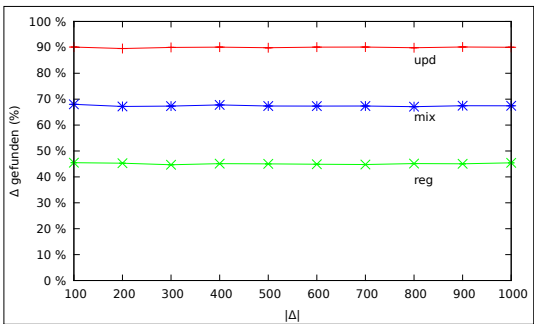
Die Regenerationsgenauigkeit entspricht in etwa der Hälfte der *upd* Genauigkeit. Dies ist zu



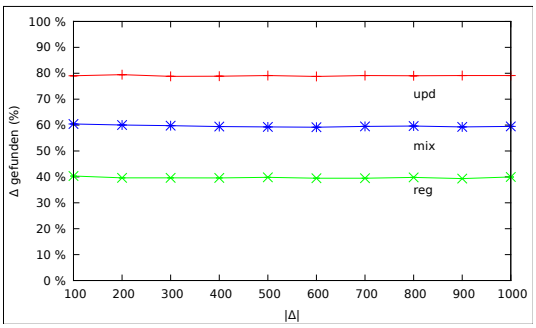
(a) Fpr = 0,001



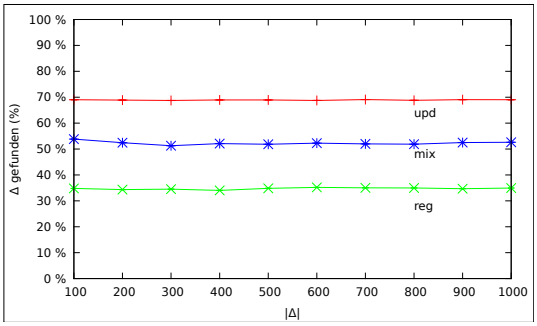
(b) Fpr = 0,01



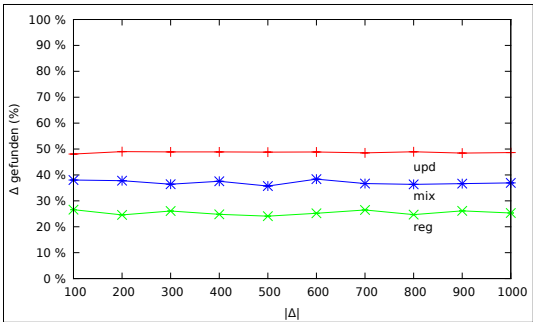
(c) Fpr = 0,1



(d) Fpr = 0,2



(e) Fpr = 0,3



(f) Fpr = 0,5

Abbildung 4.1: *Bloom Filter* Genauigkeit

erwarten gewesen, weil nur die Hälfte der fehlenden Kopien mit einer Fpr erkannt werden. Bei $Fpr = 0{,}2$ werden demzufolge 80% der Hälfte der verlorenen Kopien gefunden, was den in Abbildung 4.1d eingezeichneten 40% entspricht.

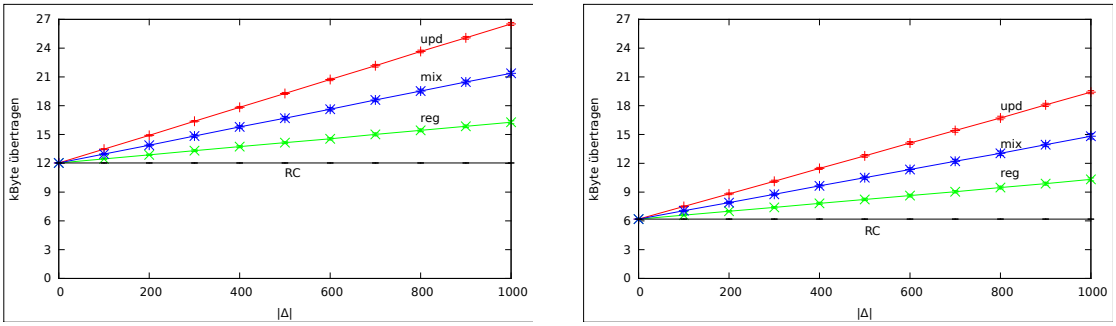
Die Erkennungsgenauigkeit bei gemischten Fehlern (*mix*), liegt relativ mittig zwischen der *upd* und der *reg* Genauigkeit, da beide ca. 50% der fehlerhaften Kopien stellen. Die Mischung der beiden Fehlertypen in *mix* bestimmt dessen Genauigkeit.

Die Analyse der Fehlerwahrscheinlichkeit zeigt auch, dass die Genauigkeit unabhängig ist von der Differenzgröße, aber abhängig vom Fehlertyp. Im Weiteren werden exemplarisch zwei *Bloom Filter* (A) mit $Fpr = 0{,}01$ und (B) $Fpr = 0{,}1$ betrachtet. Der *Bloom Filter* (B) bietet eine um 10% verringerte Genauigkeit gegenüber (A) ist jedoch nur halb so groß nach Formel 2.6.

Übertragungskosten

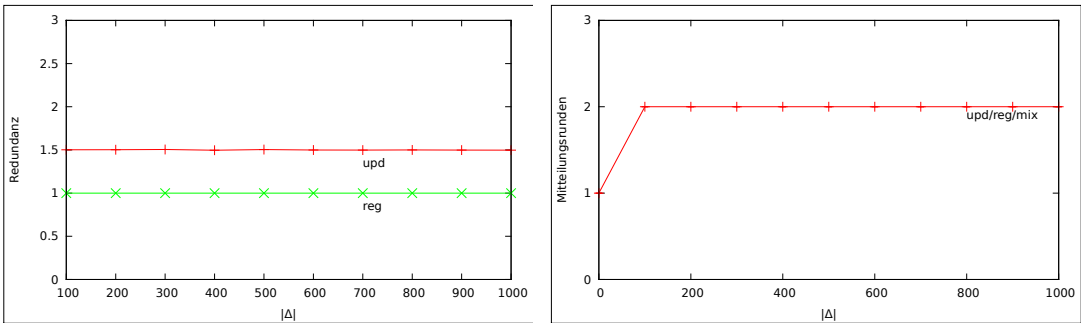
Die Übertragungskosten der ausgewählten *Bloom Filter* werden in Abbildung 4.2 für verschiedene Δ-Größen dargestellt. Der RC-Graph zeigt die Übertragungskosten des RC-Dienstes, welche nur aus dem *Bloom Filter* bestehen. Die weiteren drei Graphen in den Abbildungen 4.2 zeigen die gesamten Übertragungskosten für die drei verschiedenen Fehlertypen. Nach dem Abzug der konstanten RC-Kosten ist ersichtlich, dass die Erkennung veralteter Kopien (*upd*) dreimal so teuer ist, wie die verlorener (*reg*). Dies resultiert aus den erhöhten Kosten für die Erkennung veralteter Kopien sowie aus der doppelten Erkennungsgenauigkeit. Eine veraltete Kopie wird ca. 1,5 mal übertragen, wenn A und B zu gleichen Teilen veraltete Kopien besitzen. Dies zeigt die Redundanz-Abbildung 4.3a. Wird die doppelte Anzahl an erkannten Differenzen zu den eineinhalbfachen Kosten übertragen, entsprich dies den drei-fachen *reg*-Kosten. Der Redundanzwert von 1,5 bei der Erkennung von veralteten Kopien hat seinen Ursprung in der Trennung von $Upd(A,B)$ und $Upd(B,A)$ aus Δ. Wurde von B eine Differenz Δ identifiziert ist diesem nicht bekannt welche der Differenzen aus $Upd(A,B)$ bzw. $Upd(B,A)$ stammen. Dazu wird Δ an A übertragen der $Upd(A,B)$ ermittelt und aus Δ entfernt. Die verbleibende Menge $Upd(B,A)$ wird an B zurück gesendet (siehe Kap. 3.4).

Für den Fehlertyp *mix* ergeben sich die Übertragungskosten als Mischung der anderen beiden Fehlertypen. Der *mix* Fehlertyp war bei der Genauigkeit sowie bei den Übertragungskosten vollständig von den anderen beiden Fehlertypen abhängig weswegen dieser nicht weiter betrachtet wird.



(a) Fpr 0.01 **(b)** Fpr 0.1

Abbildung 4.2: *Bloom Filter* Übertragungskosten



(a) Redundanz **(b)** Mitteilungsrunden

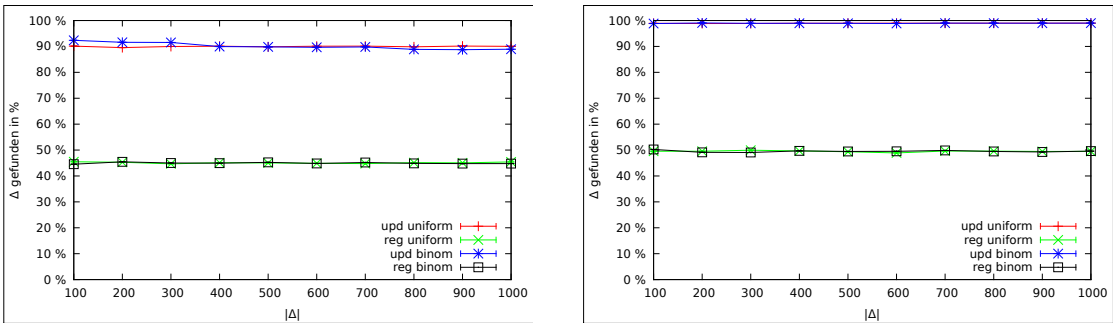
Abbildung 4.3: *Bloom Filter*

Mitteilungsrunden

Die Mitteilungsrunden liegen bei der Verwendung des *Bloom Filters* immer zwischen Eins und Zwei, wie die Abbildung 4.3b veranschaulicht. Zur Bestimmung der Differenz benötigt der RC-Dienst eine Runde sowie zur Auflösung dieser eine weitere. Bei einer Differenz von Null wird nur eine Nachrichtenrunde benötigt.

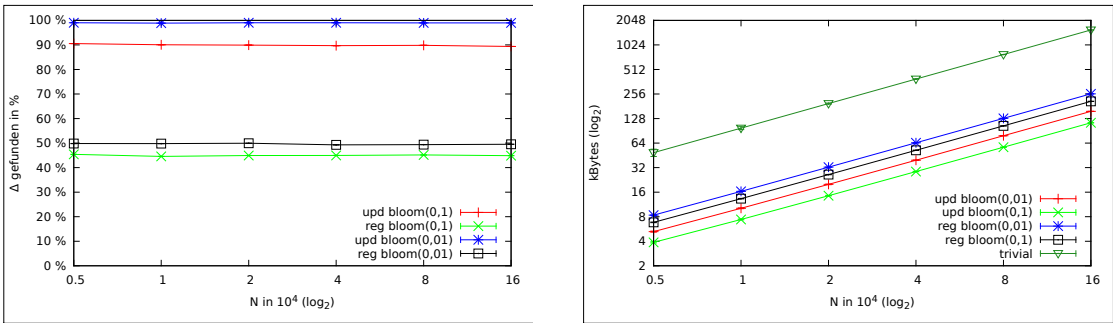
Fehler- und Datenverteilung

Der *Bloom Filter* ist invariant gegenüber der Fehler- und Datenverteilung. Für die Genauigkeit wird dies exemplarisch in der Abbildung 4.4a für die Fehlerverteilung sowie in 4.4b für die Datenverteilung gezeigt. Die Abbildungen zeigen den *Bloom Filter* mit Fpr = 0,1. Die mit *uniform* gekennzeichneten Graphen repräsentieren eine gleichmäßige Verteilung, jene mit *binom* gekennzeichneten die Binomialverteilung. Es ist ersichtlich, dass keine nennenswerten Schwankungen zwischen den verschiedenen Verteilungen vorliegen.



(a) Fehlerverteilung **(b)** Datenverteilung

Abbildung 4.4: *Bloom Filter* Fehler- und Datenverteilungssensitivität



(a) Genauigkeit **(b)** Übertragungskosten

Abbildung 4.5: *Bloom Filter* Skalierbarkeit

Skalierbarkeit

Bisher wurde der *Bloom Filter* mit einer konstanten Datenmenge N analysiert. Um dessen Skalierbarkeit zu analysieren wird die Differenzgröße auf $|\Delta| = 0.03N$ fixiert und N schrittweise erhöht, wie in den Abbildungen 4.5 dargestellt. Die Daten und Fehler sind in diesem Szenario wieder gleichmäßig verteilt. Die Genauigkeit des *Bloom Filters* gibt bei einer steigenden Datenmenge minimal nach und ähnelt der entsprechenden Genauigkeit aus 4.1b und 4.1c. Die Abbildung 4.5b zeigt die linear wachsenden Übertragungskosten sowie das Verhältnis der Kosten gegenüber der trivialen Problemlösung, dem Versenden der N Datensätze. Die beiden Abbildungen zeigen, dass der *Bloom Filter* sowohl bei der Genauigkeit als auch bei den Übertragungskosten skaliert.

Zusammenfassung

Zusammenfassend bietet der *Bloom Filter* eine konstante Genauigkeit zu der definierten Fehlerwahrscheinlichkeit. Der *Bloom Filter* ist unabhängig von der Fehler- und Datenverteilung sowie von der Größe von Δ. Die Übertragungskosten hängen von der gewählten Fehlerwahrscheinlichkeit und der Datenmenge N ab und sind gering gegenüber einer vollständigen Übertragung der Daten. Die Redundanz liegt beim *Bloom Filter* für verlorene Kopien bei optimalen 1,0 und für veraltete bei moderaten 1,5. Dies zeigt die Effizienz der Übertragungskosten.

4.4.2 *Merkle-Tree*

Algorithmus

Der *Merkle-Tree* besitzt die zwei Parameter Verzweigungsgrad v und *Bucket*-Größe b. Der Verzweigungsgrad v definiert die Anzahl der Kindknoten je Knoten. Die *Bucket*-Größe b bestimmt die Anzahl der Datenelemente, die in einem Blatt zusammengefasst werden können bzw. ab wann ein Knoten expandiert wird. Die Notation merkle(v,b) bezeichnet im Weiteren einen *Merkle-Tree* mit der angegebenen Parameterwahl.

Die Baumtiefe eines *Merkle-Trees* ist definiert als der längste Weg eines Knotens zur Wurzel. Die Weglänge ist dabei die Anzahl der Kanten, wodurch ein Baum der nur aus einer Wurzel besteht die Tiefe Null besitzt. Die Tiefe $t \in \mathbb{N}$ des *Merkle-Trees* kann nur für gleichmäßig verteilte Daten abgeschätzt werden. Die bekannte Formel für die Tiefe $t = \lceil \log_v(N) \rceil$ eines n-ären Baumes kann nur verwendet werden, wenn kein *Bucketing* ($b > 1$) verwendet wird.

Das *Bucketing* fasst Daten und damit Blätter des *Merkle-Tree* zusammen. Das Ziel ist die Verringerung der Knotenanzahl, von den Blättern beginnend. Ein Baum mit Verzweigungsgrad v besitzt bei einer Höhe t, v^t Blätter. Ein beliebiger Wert für b würde die Blattanzahl und vieleicht auch innere Knoten einsparen, wäre aber in seiner Auswirkung nicht abschätzbar. Darum wird versucht die Tiefe des Baumes zu verringern. Wird von einem Baum ohne *Bucketing* ($b = 1$) ausgegangen und soll dessen Tiefe mit Hilfe des *Bucketing* um $\varepsilon \in \mathbb{N}, \varepsilon < t$ gesenkt werden, so ergibt sich b nach der Formel:

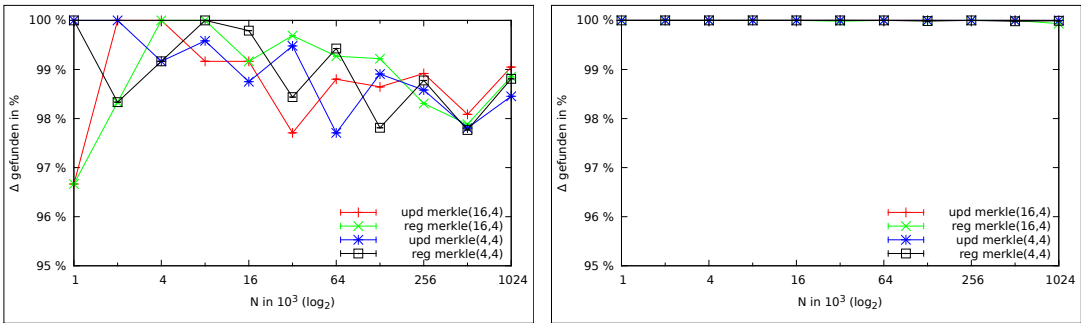
$$b = \left\lceil \frac{N}{v^{t-\varepsilon}} \right\rceil = \left\lceil \frac{N}{v^{\lceil \log_v(N) \rceil - \varepsilon}} \right\rceil \tag{4.1}$$

Das gewählte b entspricht dem notwendigen Verhältnis der Daten zu den Blättern sodass der Baum um $t - \varepsilon$ Ebenen verkürzt wird. Die Tiefe eines *Merkle-Tree* mit *Bucketing* kann für eine gleichmäßig verteilte Datenmenge mit N Elementen abgeschätzt werden. Die Baumtiefe für N mit $b = 1$ ist $\log_v(N)$. Wird ein $b > 1$ gewählt werden jeweils b Elemente von N zusammengefasst und der daraus resultierende Baum ist im besten Fall äquivalent ist zu einem Baum mit $N = \frac{N}{b}$ ohne *Bucketing*. Weil das *Bucketing* die Tiefe nicht erhöhen kann existiert auch eine schwache obere Grenze, sodass die Tiefe eines *Merkle-Tree* mit *Bucketing* $t_{mt}(v,b,N)$ abgeschätzt werden kann:

$$\lceil \log_v(\tfrac{N}{b}) \rceil \leq t_{mt}(v,b,N) \leq \lceil \log_v(N) \rceil \tag{4.2}$$

Genauigkeitsanalyse

Die Genauigkeit des *Merkle-Tree* hängt ausschließlich von der Qualität der Signaturen der Knoten ab. Im Falle einer Hashkollision kann es zum Auslassen eines ganzen Bereichs des Baumes kommen, wodurch Differenzen in diesem Teil nicht erkannt werden. Um Hashkollisionen zu vermeiden



(a) 8 Bit Signaturgröße **(b)** 16 Bit Signaturgröße

Abbildung 4.6: *Merkle-Tree* Genauigkeit

gilt es einen Hash zu finden der möglichst wenig Bit benötigt jedoch „stark“ genug ist die Wahrscheinlichkeit von Hashkollisionen zu minimieren. Die verwendete 160-Bit SHA-1 Prüfsumme verhindert bei den geprüften Datenbankgrößen Hashkollisionen völlig. Die Signaturgröße wurde verringert, indem nur jeweils die letzten x Bit der SHA-1 Prüfsumme verwendet wurden. Die Abbildung 4.6a zeigt, dass Hashkollisionen bei einer Signaturgröße von 8 Bit auftreten. Eine hundertprozentige Erkennungsgenauigkeit wird ab 16 Bit erreicht wie Abbildung 4.6b zeigt. Die Kollisionswahrscheinlichkeit steigt mit der Anzahl der Knoten des *Merkle-Trees*, weshalb in beiden Abbildungen ein sehr leichter Abwärtstrend in der Genauigkeit bei steigender Datenmenge zu erkennen ist. Für die Signaturgröße von 16 Bit ist der Abfall der Genauigkeit nur angedeutet in der Abbildung zu erkennen. Bei der Betrachtung der Daten, zeigt sich jedoch bei wachsendem N eine Zunahme nicht erkannter Differenzen in einstelliger Höhe.

In Abbildung 4.7 sind die Übertragungskosten des RC-Dienstes für die beiden Signaturgrößen abgebildet. Für beide Signaturgrößen sind die Kosten bei einem Verzweigungsgrad von 16 höher als bei einem mit 4, aufgrund der unterschiedlichen Baumgröße. Die Bäume mit jeweils identischem Verzweigungsgrad besitzen zu einem N jeweils die gleiche Knotenzahl, wodurch der Größenunterschied in kByte zwischen diesen ausschließlich auf die Veränderung der Signaturgröße zurückzuführen ist.

Die Genauigkeit ist unabhängig vom Verzweigungsgrad und der *Bucket*-Größe und liegt im Weiteren bei 100% durch das feste $N = 10000$. Dadurch sind weitere Genauigkeitsanalysen unnötig. Der Schwerpunkt liegt im Folgenden auf der Minimierung der Mitteilungsrunden und Übertragungskosten.

Analyse des *Bucketing*

Um die Auswirkung der *Bucket*-Größe auf die Übertragungskosten zu analysieren, wird die *Bucket*-Größe für verschiedene Verzweigungsgrade schrittweise erhöht. Die Daten sind gleichmäßig verteilt

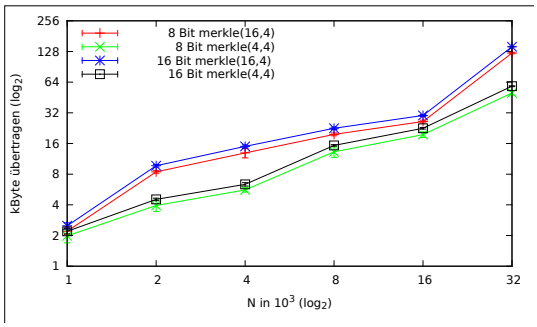
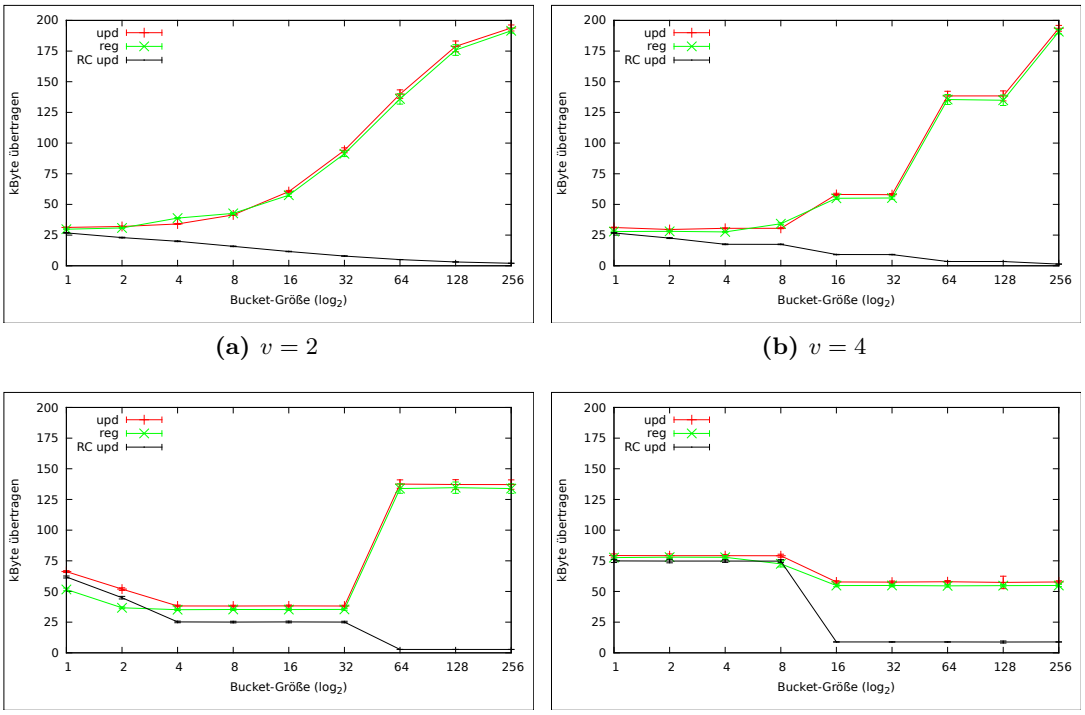


Abbildung 4.7: *Merkle-Tree* RC-Übertragungskosten bei 8 und 16 Bit Signaturen

sowie die $|\Delta| = 0.03N$ Fehler. Die Abbildungen 4.8 zeigen für die Verzweigungsgrade 2, 4, 16 und 32 die Entwicklung der Übertragungskosten. Es sind jeweils drei Graphen abgebildet, zwei für die Fehlertypen *upd* und *reg* und einer für die Kosten des RC-Dienstes im Falle des Fehlertyps *upd*. Abbildung 4.8a zeigt einen Binärbaum, der exemplarisch die zwei Effekte zeigt, die bei den anderen drei abgebildeten Bäumen abrupt vorkommen. Den ersten Effekt stellen die Übertragungskosten des RC-Dienstes dar, die bei steigender *Bucket*-Größe fallen. Der Formel 4.2 ist zu entnehmen, dass bei steigender *Bucket*-Größe die Baumtiefe sinkt. Eine sinkende Baumtiefe verringert die Knotenanzahl und somit die Anzahl der übertragenen Signaturen, wodurch die RC-Kosten mit steigender *Bucket*-Größe fallen. Die steigenden Übertragungskosten sind der zweite beobachtete Effekt. Diese werden vom RS-Dienst verursacht, da der Abstand zwischen dem *upd* und *reg* Graph zum RC Graphen steigt. Dies zeigt eine wachsende Redundanz, weil bei konstanter Fehleranzahl die Anzahl der übertragenen KVV-Tripel steigt. Das *Bucketing* verursacht demnach eine erhöhte Redundanz, da mehr KVV-Tripel bei nicht übereinstimmenden Blattsignaturen übertragen werden müssen. Die Abbildung 4.8d zeigt eine Senkung der gesamten Übertragungskosten obwohl auch hier die Redundanz ansteigt. Dies wird von der KVV-Tripelgröße von 10 Byte verursacht, würde diese 30 Byte betragen würden auch hier bei steigender *Bucket*-Größe die Übertragungskosten wachsen.

Die eben erläuterten Effekte verlaufen in den Abbildungen 4.8b bis 4.8d stufenweise. Die Stufengröße in kB erhöht sich und es sinkt die Stufenhäufigkeit mit steigendem Verzweigungsgrad. Die Quelle der Stufenbildung ist die Baumtiefe nach Formel 4.2. Die geraden Stellen in den Graphen der Abbildungen 4.8b bis 4.8d repräsentieren jeweils einen Baum mit gleicher Tiefe, nur mit mehr oder weniger KVV-Tripeln in den *Buckets*. Ab einem gewissen Punkt, der nach Formel 4.1 berechnet werden kann, reicht die *Bucket*-Größe einer höheren Ebene und die Baumtiefe schrumpft. In diesem Fall sinken die RC-Kosten abrupt und es steigen die redundant übertragenen KVV-Tripel. Dies veranschaulicht die Abbildung 4.8c im Punkt $b = 64$ sehr gut. Die Abnahme der Stufenhäufigkeit folgt aus der exponenziell von v abhängenden Blattanzahl.

Die Abbildungen 4.8 belegen, dass das *Bucketing* erhöhte RS-Kosten verursacht. In 4.8c ist ein



(a) $v = 2$ **(b)** $v = 4$

(c) $v = 16$ **(d)** $v = 32$

Abbildung 4.8: *Merkle-Tree* Übertragungskosten je *Bucket*-Größe

positiver Effekt auf die Übertragungskosten zu erkennen, da die Baumtiefe gesenkt wurde. Es ist vom Anwendungsfall abhängig, ob die durch das *Bucketing* verringerten RC-Kosten zu Lasten der erhöhten RS-Kosten einen Vorteil bilden.

Ein weiteres Indiz für die Senkung der RC-Kosten durch die *Bucket*-Größe bietet die Abbildung 4.9. Dort sind für die vier Verzweigungsgrade aus 2, 4, 16 und 32 für den Fehlertyp *upd* die Mitteilungsrunden je *Bucket*-Größe dargestellt. Beim *Merkle-Tree* verursacht jede Baumebene eine Mitteilungsrunde, sodass eine gesenkte Mitteilungsrundenanzahl die Anzahl der Blätter exponenziell verringert, aufgrund der Baumtiefenverringerung um 1. Die RC-Kostenersparnis ist dementsprechend beim Vergleich der Abbildungen 4.8 und 4.9 ersichtlich. Bei der *Bucket*-Größe 16 sinkt beispielsweise die Anzahl der Mitteilungsrunden auf 5, was in Abbildung 4.8d zu einer sehr starken Senkung der RC-Kosten führt.

Die Betrachtung der Mitteilungsrunden zeigt, dass bei $v = 4$ die Mitteilungsrunden mit 10 angegeben sind, obwohl $t_{mt}(4{,}1{,}10000) = 7$. Bei allen anderen Graphen liegt die eingezeichnete Rundenanzahl auch um 3 Runden über der Baumhöhe. Zwei dieser Runden benötigt das RC-Protokoll. Eine für die Bestimmung des gemeinsamen Intervalls sowie eine weitere für die Benachrichtigung des Partners über den Abschluss des Vergleichs. Die dritte Runde dient der Auflösung der

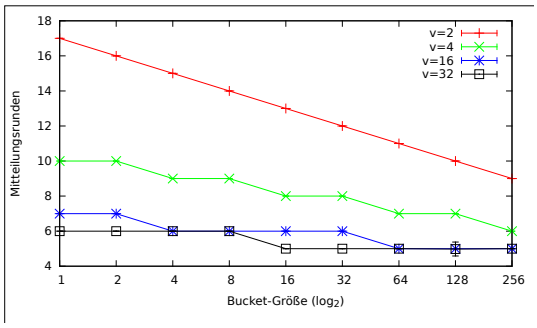


Abbildung 4.9: *Merkle-Tree* Mitteilungsrunden je *Bucket*-Größe

Blattdifferenzen durch den RS-Dienst.

Die Abbildung 4.9 gibt die Funktion t_{mt} (definiert auf Seite 54 in Formel 4.2) mit konstanten Zusatzkosten wieder, sodass die Stufenbildung, welche auch in Abbildung 4.8 zu beobachten ist erklärt werden kann. Denn diese wird von der Aufrundungsfunktion (*ceiling function*) in t_{mt} verursacht.

Analyse des Verzweigungsgrades

In den Abbildungen 4.8 zeigte sich bereits, dass die Übertragungskosten bei höheren Verzweigungsgraden im Allgemeinen geringer waren als beim Binärbaum. Zur genaueren Analyse des Einflusses von v wird der Verlauf der Übertragungskosten bei steigendem Verzweigungsgrad in der Abbildung 4.10 für $b = 4$ betrachtet. Die Betrachtung anderer *Bucket*-Größen ist nicht nötig, weil diese nach 4.2 nur N dividieren. So kann zu jedem b ein N gewählt werden, sodass deren Verhältnis dem hier Betrachteten Szenario mit $b = 4$ und $N = 10000$ entspricht. Die Daten und Fehler sind in diesem Szenario gleichverteilt und $|\Delta| = 0.03N$.

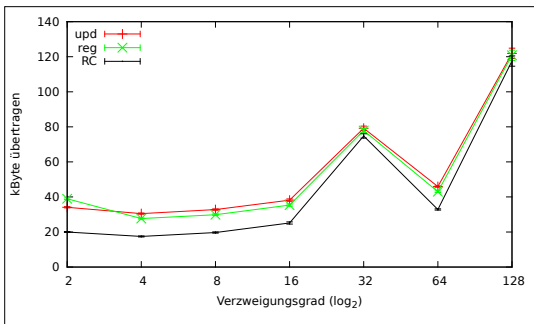
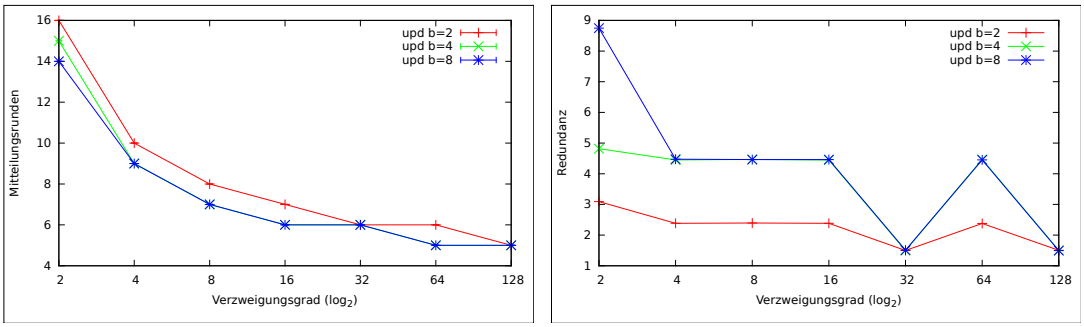


Abbildung 4.10: *Merkle-Tree* Übertragungskosten je Verzweigungsgrad

Der Graphen 4.10 zeigt im Punkt $v = 32$ und $v = 128$ zwei sehr deutliche abrupte Steigerungen. Dies zeigt, dass die Wahl des Verzweigungsgrads die Übertragungskosten stark beeinflusst. Ursache für diese Steigerungen ist die stark erhöhte Blattanzahl. Der Baum mit $v = 32$ besitzt eine Höhe



(a) Mitteilungsrunden **(b)** Redundanz

Abbildung 4.11: *Merkle-Tree* je Verzweigungsgrad

von $t_{mt}(32,4,N) = 3$ und $32^3 = 32768$ Blätter, wohingegen der Baum mit $v = 16$ mit der gleichen Höhe nur $16^3 = 4096$ Blätter besitzt. Weiterhin kann beobachtet werden, dass die Kosten des Baumes $v = 64$ mit Höhe 2 leicht erhöht sind gegenüber jenen vom Baum $v = 16$, trotz der gleichen Blattanzahl von $64^2 = 16^3 = 4096$. Dies resultiert aus der gestiegenen Anzahl innerer Knoten bei $v = 64$.

Die Anzahl der Blätter im Fall von $v = 32$ könnte stark verringert werden, wenn in der vorletzten Ebene des Baumes nicht der Verzweigungsgrad erzwungen wird. Eine solche Erweiterung erhöht die Komplextität des *Merkle-Tree* Protokolls und bietet erhebliches Potenzial für zukünftige Optimierungen.

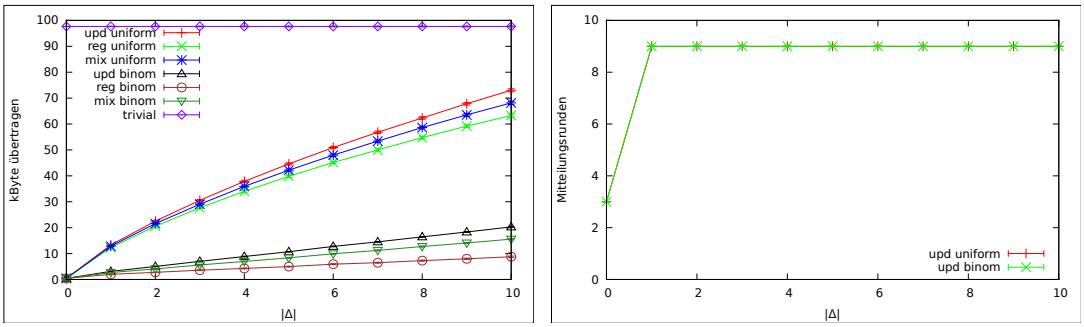
In der Abbildung 4.10 ist auch eine leichte Verringerung des Abstandes der *upd* und *reg* Graphen zum RC-Graphen zu erkennen. Dies signalisiert den Eintausch von RS-Kosten gegen RC-Kosten durch Steigerung des Verzweigungsgrades. Ursache hierfür ist die feinere Partitionierung des Intervalls, die sich senkend auf die Redundanz auswirkt, was die Abbildung 4.11b zeigt.

Abschließend wird in Abbildung 4.11a der Verlauf der Mitteilungsrunden bei steigendem Verzweigungsgrad betrachtet. Es ist zu sehen, dass bei der Verwendung von einem Verzweigungsgrad von Vier, ein Drittel der Runden eingespart werden kann gegenüber dem Binärbaum. Dieses Einsparungsverhältnis verringert sich jedoch immer weiter bei erhöhtem Verzweigungsgrad.

Nachdem die Auswirkungen des Verzweigungsgrads und des *Branching* analysiert wurden, werden für $v = 4, b = 4$ die Auswirkungen einer binomialen Fehler- und Datenverteilung analysiert. Diese Parameterkonstellation zeigt geringe Übertragungskosten sowie eine mittlere Anzahl an Mitteilungsrunden.

Fehlerverteilung

Der *Merkle-Tree* ist dafür ausgelegt einzelne Fehler effektiv zu finden. Bei der bisherigen Analyse waren die fehlerhaften Kopien im *Merkle-Tree*-Intervall gleichverteilt. Dies entspricht dem schlechtest



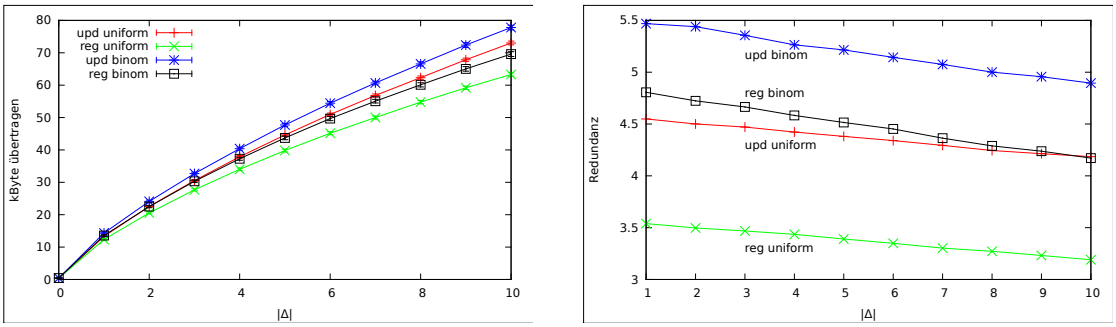
(a) Übertragungskosten **(b)** Mitteilungsrunden

Abbildung 4.12: Auswirkung der Fehlerverteilung auf merkle(4,4)

möglichen Fall des *Merkle-Trees*, da die gesamte Breite des Baumes durchsucht wird. In DHash [20] wurde bereits festgestellt, dass der *Merkle-Tree* nur bei kleinem $\Delta \leq 5\%$ effizient arbeitet. Die Abbildung 4.12a zeigt jedoch, dass ein erhöhter Verzweigungsgrad größer 2 diese Effizienzgrenze auf über 10% anheben kann. Eine binomiale Fehlerverteilung mit $p = 0{,}2$ stellt theoretisch einen sehr günstigen Fall für den *Merkle-Tree* dar, weil der rechte Teilbaum übersprungen werden kann. Dieses Verhalten kann in Abbildung 4.12a für die Übertragungskosten und in 4.12b für die Mitteilungsrunden beobachtet werden. In 4.12a sind die Übertragungskosten der beiden mit *binom* gekennzeichneten Graphen deutlich geringer als die *uniform* Graphen, die die Kosten der gleichmäßigen Fehlerverteilung darstellen. Die als „trivial“ eingezeichnete Linie zeigt die Kosten bei der vollständigen Übertragung aller KVV-Tripel mit $10000 * 10$ Byte. Die Differenzbestimmung mit Hilfe des *Merkle-Tree* ist nur sinnvoll, wenn die Übertragungskosten unter denen der trivialen Übermittlung liegen. Für eine Differenzgröße bis 10% ist dies der Fall.

Die Graphen der binomialen Fehlerverteilung steigen wesentlich langsamer bei wachsender Differenzgröße als jene der bei gleichmäßiger Fehlerverteilung. Dies zeigt den erwarteten Effekt der Sensibilität des *Merkle-Tree* auf die Fehlerverteilung. Die Kostenersparnis durch binomiale Fehlerverteilung liegt bei ca. 33%. Die Abbildung 4.12b zeigt die benötigten Mitteilungsrunden des *Merkle-Tree* für die jeweiligen Fehlerverteilungen anhand des Fehlertyps *upd*. Diese sind abhängig von der Baumtiefe und unabhängig vom Fehlertyp und der Fehlerverteilung, wodurch kein Unterschied auftreten kann.

Der Fehlertyp *mix* ist in 4.12a eingezeichnet und zeigt die Übertragungskosten bei gemischten Fehlern. Diese ergeben sich aus der Kombination der beiden Fehlerarten und werden daher nicht weiter betrachtet.



(a) Übertragungskosten **(b)** Redundanz

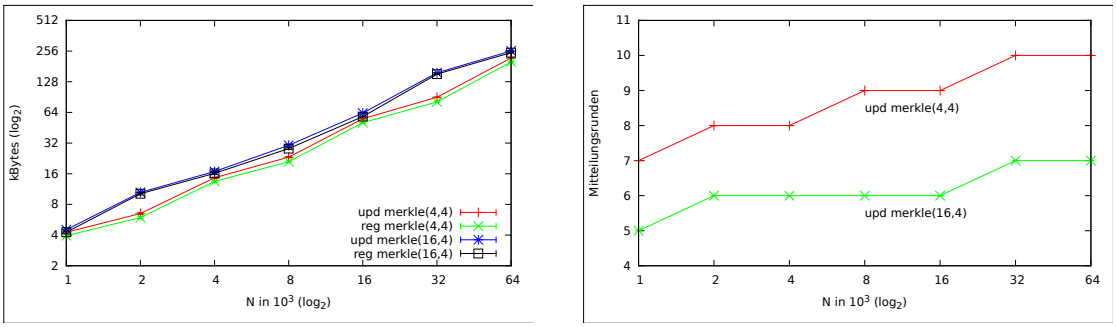
Abbildung 4.13: Auswirkung der Datenverteilung auf merkle(4,4)

Datenverteilung

Der *Merkle-Tree* verzweigt sich entsprechend der Datendichte einzelner Teilintervalle, sodass die binomiale Datenverteilung zu einem linkslastigen Baum führt. In der Abbildung 4.13a wird die Auswirkung einer binomialen Datenverteilung auf die Übertragungskosten für verschiedene Differenzgrößen dargestellt. Die Fehler sind in diesem Szenario gleichmäßig verteilt. In der Abbildung sind gesteigerte Übertragungskosten für beide Fehlertypen ersichtlich. Diese resultieren nicht aus gestiegenen Signaturkosten sondern aus der erhöhten Redundanz, wie die Abbildung 4.13b zeigt. Die Häufung der Daten im linken Teilbaum hat zu einer stärkeren Auslastung der *Buckets* geführt, wodurch die Redundanz und somit die Übertragungskosten gestiegen sind. Das *Bucketing* ist damit die Ursache für die Empfindlichkeit des *Merkle-Trees* gegenüber der binomialen Datenverteilung. Wäre die Redundanz nicht gestiegen, würden sich die Übertragungskosten nicht ändern. Damit ist bei ungleichmäßiger Datenverteilung eine kleine *Bucket*-Größe vorteilhaft.

Skalierbarkeit

Die Anaylse des *Merkle-Tree* schließt mit der Untersuchung der Skalierbarkeit. In diesem Szenario liegen die Daten und Fehler gleichmäßig verteilt vor und $|\Delta| = 0{,}03N$. Für die Genauigkeit wurde die Skalierbarkeit bereits in Abbildung 4.6b auf Seite 55 gezeigt. Das Verhalten der Übertragungskosten bei steigendem N wird in Abbildung 4.14a dargestellt. Es ist ersichtlich, dass diese linear mit N wachsen und der *Merkle-Tree* auch in diesem Punkt skaliert. Die Stufen, welche durch das Hinzufügen einer weiteren Ebene im Baum entstehen, sind in den Punkten $N = 2 * 10^3$ und $N = 32 * 10^3$ für merkle(16,4) zu erkennen sowie für merkle(4,4) in $N = 4 * 10^3$. In Abbildung 4.14b sind die Mitteilungsrunden bei wachsendem N abgebildet, die stufenweise mit der Baumtiefe wachsen. Dabei ist gut zu erkennen, dass ein hoher Verzweigungsgrad die Stufenlänge erhöht. Dies resultiert aus dem erhöhten Fassungsvermögen einer stäkeren Verzweigung gegenüber einer kleineren.



(a) Übertragungskosten **(b)** Mitteilungsrunden

Abbildung 4.14: Skalierbarkeit für merkle(4,4) und merkle(16,4)

Zusammenfassung

Zusammenfassend bietet der *Merkle-Tree* eine sehr gute Genauigkeit bei moderater Latenz. Die Latenz kann durch einen erhöhten Verzweigungsgrad zu Lasten geringfügig erhöhter Übertragungskosten gesenkt werden. Mit Hilfe des *Bucketing* kann die Latenz weiterhin leicht verringert werden auf Kosten einer steigenden Redundanz sowie einer leichten Empfindlichkeit gegenüber ungleichmäßiger Datenverteilungen. Weiterhin tauscht das *Bucketing* Übertragungskosten des RC-Dienstes gegen Redundanz ein, die die Übertragungskosten des RS-Dienstes erhöhen. Eine hohe *Bucket*-Größe ist daher nur in Anwendungsfällen vorteilhaft, welche viele kleine Tripel verwalten. Der *Merkle-Tree* arbeitet sehr effizient bei wenigen Fehlern sowie bei ungleichmäßigen Fehlerverteilungen, sodass diese Fälle zu einer starken Reduktion der Übertragungskosten führen. Die Analyse des *Bucketing* und des *Verzweigungsgrades* hat gezeigt, dass der *Merkle-Tree* sehr empfindlich ist bei der Wahl des Verzweigungsgrads. Das *Bucketing* verringert die Baumtiefe aufkosten der Redundanz und kann mit Hilfe der vorgestellten Formel 4.1 für eine beliebige Verringerung der Tiefe berechnet werden. Eine ungünstige Wahl des Verzweigungsgrades kann zu plötzlichen Sprüngen in den Übertragungskosten führen.

4.4.3 ART

Algorithmus

Der *Approximate Reconciliation Tree* (ART) stellt einen als *Bloom Filter* codierten *Merkle-Tree* dar und besitzt fünf Parameter. Zum einen die klassischen *Merkle-Tree* Parameter Verzweigungsgrad v sowie *Bucket*-Größe b, zum anderen die Fehlerwahrscheinlichkeit des *Bloom Filters* mit Fpr_I für die inneren Knoten sowie Fpr_B für die Blattknoten. Der fünfte Parameter ist das *correction level*, beschrieben in 2.4.4, welches die Genauigkeit der Suche in ART erhöhen soll und keinen Einfluss auf die Konstruktion besitzt. Die Signaturgröße für die Knoten des *Merkle-Tree* beträgt 16 Bit.

Die Messung besteht aus der Synchronisation eines Knotenpaars A,B mit B als Initiator. Die Synchronisation erfolgt in zwei Phasen. In der ersten Phase ermittelt der RC-Service von B die Differenz Δ und in der zweiten Phase, falls $|\Delta| > 0$ wird die Differenz durch den RS-Service aufgelößt. Nach diesen zwei Mitteilungsrunden ist die Synchronisation abgeschlossen und die Metriken können bestimmt werden.

Clustering der Parameterkonstellationen

Aufgrund des sehr großen Parameterraumes wurde eine Tabelle erstellt, die sich im Anhang A befindet und die Auswirkung verschiedenster Parameterkonstellationen auf die drei Metriken Genauigkeit, Übertragungskosten und Redundanz zeigt. In der Tabelle sind alle Parameterkonstellationen des kartesischen Produkts der Parametermengen $v = \{2, 4, 8, 16, 32, 64\}$, $b = \{1, 2, 4, 8\}$, $\mathrm{Fpr}_I = \{0{,}001;\ 0{,}01;\ 0{,}1;\ 0{,}4;\ 0{,}8\}$, $\mathrm{Fpr}_B = \{0{,}01;\ 0{,}1;\ 0{,}2\}$ aufgeführt. Im Folgenden werden Parameterkonstellationen auch als Konfiguration bezeichnet.

In der Tabelle kann beobachtet werden, dass viele Konfigurationen sich in der Redundanz sowie in der *reg* Genauigkeit stark gleichen. Die beobachteten Cluster sind in Tabelle 4.1 aufgeführt.

Cluster	**upd**	**reg**		Tripel
	Redundanz	Genauigkeit	Redundanz	je *Bucket*
C_A	$\approx 1{,}5$	$\leq 50\%$	1	≤ 1
C_A^*	$\approx 1{,}5$	55% - 58%	6,3 - 6,6	≤ 1
C_B	2,3 - 8,7	0% - 96%	3,4 - 9,6	> 1

Tabelle 4.1: ART Parameter Clustering

Die zwei Cluster C_A und C_B unterscheiden sich deutlich in der *reg* Redundanz, die bei C_A immer genau 1 ist und bei C_B stark schwankt und immer deutlich höher liegt. Hinzu kommt der *upd* Redundanzgrad, der bei C_A leicht um 1,5 schwankt, bei C_B jedoch wieder deutlich höher liegt. Der dritte beobachtete Clustering Parameter ist die *reg* Genauigkeit, die bei C_A immer unter 50% liegt und bei C_B über die gesamte Bandbreite schwankt.

Das Verhältnis der KVV-Tripel je *Bucket* ist die Ursache für die Entstehung der Cluster. Ein Redundanzwert von 1 bei *reg* kann in C_A nur entstehen, wenn in einem *Bucket* nur ein Element enthalten ist. Ist in jedem *Bucket* jeweils nur ein Element, läßt sich auch die *upd* Redundanz von

$\approx$ 1,5 erklären. Bei der Erkennung eines differierenden *Buckets* wird das Element dem Partner übertragen, welcher dieses in der Hälfte der Fälle dem Initiator zurück sendet. Das Zurücksenden erfolgt weil der Initiator die veraltete Version besaß. Dies entspricht der gleichen Trennung von Δ wie sie schon beim *Bloom Filter* vorkam und in 3.3.1 beschrieben ist. Diese Redundanzwerte können jedoch nur vorkommen, wenn in den differierenden Knoten nur ein Element in jedem *Bucket* vorhanden war, andernfalls wäre die Redundanz höher. In Cluster C_B besitzt jeweils höhere Redundanzwerte sodass dort mehr als ein Element je *Bucket* vorhanden ist. Die Datenmenge N wird bei einer Tiefe t auf v^t Blätter aufgeteilt. Das Verhältnis von N zu v^t entscheidet über die Clusterzugehörigkeit, wie die Formel verkürzt darstellt:

$$\frac{N}{v^t} = \frac{N}{v^{\lceil \log_v(\frac{N}{b}) \rceil}} = \begin{cases} \leq 1, & \text{Cluster } C_A \\ > 1, & \text{Cluster } C_B \end{cases} \tag{4.3}$$

Ursache für die schlechte Genauigkeit bei fehlenden Repliken in Cluster C_A sind die leeren Blätter. Das Verhältnis der Daten je Blatt ist bei C_A kleiner gleich 1, sodass ein fehlendes Datum zu einem Blatt mit leerem *Bucket* führt. Der Hashwert aller leeren *Buckets* ist identisch, sodass in den *Bloom Filter* von ART mit hoher Wahrscheinlichkeit ein leeres Blatt hinzugefügt wurde. Fehlen dem Initiator Daten und $\frac{N}{v^t} \leq 1$, dann stimmen die leeren Blätter immer mit dem *Bloom Filter* überein und es wird keine Differenz erkannt. Dies ist der Grund weshalb B nur die Menge $Reg(A,B)$ erkennt und nicht die Menge der ihm fehlenden Kopien $Reg(B,A)$. Dieser Effekt verringert sich für C_B bei einem Verhältnis von $\frac{N}{v^t} > 1$. In diesem Fall sinkt die Wahrscheinlichkeit, dass eine fehlende Kopie zu einem leeren Blatt führt weil durchschnittlich mehr als ein Element je Blatt vorhanden ist.

Im Weiteren wird analysiert, was für $v \in \mathbb{N}^+$ und $b \in \mathbb{N}^+$ gelten muß, um einem Cluster anzugehören.

$$\frac{N}{v^{\lceil \log_v(\frac{N}{b}) \rceil}} \leq 1 \tag{4.4}$$

$$\Leftrightarrow v^{\lceil \log_v(\frac{N}{b}) \rceil} \geq N$$

$$\Leftrightarrow \lceil \log_v(\tfrac{N}{b}) \rceil \geq \log_v(N)$$

$$\Leftrightarrow \lceil \log_v(N) - \log_v(b) \rceil \geq \log_v(N)$$

$$\Leftrightarrow \underbrace{\log_v(b) \leq 0}_{\rightarrow\ b=1} \vee \left(\underbrace{0 \leq \log_v(b) < 1}_{\rightarrow\ 1<b<v} \wedge \text{frac}(\log_v(N)) > \text{frac}(\log_v(b)) \right) \tag{4.5}$$

Aus der Herleitung 4.5 folgt, dass jede Konfiguration mit $b = 1$ dem Cluster C_A angehört sowie jede Konfiguration mit $1 < b < v$ und $\text{frac}(\log_v(N)) > \text{frac}(\log_v(b))$. Diese beiden Bedingungen

geben die Parameterkonstellationen der Tabelle wieder, in der dem Cluster C_A nur Konfigurationen mit $b = 1$ sowie mit $v = 32$ und $b \in \{2,4,8\}$ angehören. Dem Cluster C_B gehört jede Konfiguration die nicht in C_A ist, in jedem Fall alle mit $b \geq v$.

Der seltenste Cluster C_A^* enthält nur Konfigurationen mit $v = 32$ und $b = 8$ die in keinem anderen Cluster enthalten sind. In diesem ist das Verhälnis von Tripeln je *Bucket* 0,3 jedoch ist die *reg* Redundanz weit über 1. Dies scheint an der *Bucket*-Größe von 8 zu liegen, die zu einer anderen Konstruktion führt als bei der C_A Konfiguration $v = 32$ und $b = 4$ trotz gleichem Tripel je *Bucket* Verhältnis. Auf der Baumebene 2 mit 32^2 Blättern besteht das Verhältnis von 9,8 Tripeln je *Bucket*. Aus der Redundanz bei C_A^* kann nur geschlossen werden, das genau an den Punkten im Baum in denen Kopien fehlen die Tiefe 2 beträgt und bei allen anderen Stellen 3. Denn nur bei der Tiefe 3 ist maximal ein Element in jedem *Bucket* sodass die Redundanz für *upd* bei 1,5 liegen kann. Der Cluster C_A^* stellt einen seltenen Spezialfall von C_A dar und wird nicht weiter betrachtet.

Auswirkung der Fehlerwahrscheinlichkeit auf Konfigurationen

Für das Clustering war nur das Verhältnis der Tripel je *Bucket* entscheidend, welches von den Parametern v und b beeinflusst wird. In der Tabelle A kann weiterhin beobachtet werden, dass die Parameter Fpr_I und Fpr_B unabhängig von v und b sind.

Der Parameter Fpr_B beeinflusst die Genauigkeit der *upd* und *reg* Erkennung, sodass die Genauigkeit in der Regel um 10% zunimmt zwischen $\mathrm{Fpr}_B = 0{,}2$ und $\mathrm{Fpr}_B = 0{,}1$. Dies ist zurückzuführen auf die stark verringerte Anzahl an *false-positives* beim Abgleich der Blätter. Je schlechter der *Bloom Filter* der Blätter ist, je mehr Blätter werden fälschlicherweise als übereinstimmend angenommen und nicht abgeglichen.

Die Genauigkeit Fpr_I für die inneren Knoten hat nach der Tabelle A einen sehr leichten Einfluss auf die Genauigkeit welcher mit steigendem Fpr_I-Wert wächst. Ab $\mathrm{Fpr}_I = 0{,}8$ ist die Fehlerwahrscheinlichkeit so hoch, dass die Wurzel fälschlicherweise als übereinstimmend klassifiziert wird. Grund für die Geringe Schwankung ist das *correction level*, welches bei der Tabelle A auf 3 festgelegt war. Das *correction level* besitzt großen Einfluss auf die Genauigkeit wenn Fpr_I hoch ist. Das *correction level* wird im Folgenden detailiert betrachtet. Es ist weiterhin ein schwacher senkender Einfluss von Fpr_I auf die Übertragungskosten zu beobachten. Dieser resultiert aus der verringerten Genauigkeit des *Bloom Filters*.

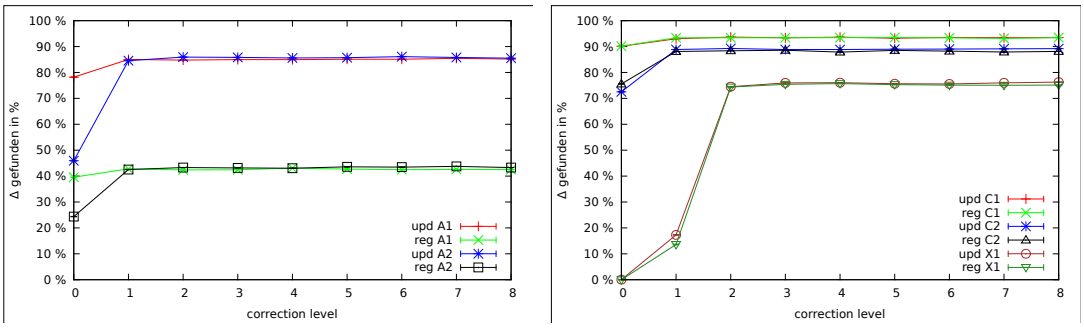
Ausgewählte Parameterkonstellationen

In den Folgenden Analysen werden zwei Konfiguration aus C_A und C_B verwendet, welche in der Tabelle 4.2 als $A*$ bzw. $C*$ aufgeführt sind. Für die Analyse des *correction level* wird eine weite Konfiguration $X1$ mit hoher Fehlerwahrscheinlichkeit Fpr_I verwendet. Die ausgewählten Konfigurationen besitzen eine gute Genauigkeit bei gleichzeitig niedrigen Übertragungskosten und geringer Redundanz. Weiterhin wurden diese im Anhang A grau unterlegt.

Name	v	b	Fpr_I	Fpr_B
A1	32	2	0,01	0,2
A2	4	1	0,01	0,01
C1	16	4	0,001	0,01
C2	8	2	0,01	0,01
X1	8	8	0,8	0,2

Tabelle 4.2: Ausgewählte ART Konfigurationen

Analyse des *correction level*



(a) Cluster C_A **(b)** Cluster C_B

Abbildung 4.15: ART *correction level*

Das *correction level* wurde in [18] vorgeschlagen um die Genauigkeit der Suche in ART zu erhöhen. Hierzu sollte verhindert werden, dass ein *false-positive* des *Bloom Filter* zum überspringen eines Teilbaumes führt. Die Abbildung 4.15a zeigt für die zwei C_A-Konfigurationen die Auswirkungen des *correction level*, sowie Abbildung 4.15a für drei C_B-Konfigurationen. Für beide Cluster nimmt die Genauigkeit bis circa *correction level* 2 zu. Die Zunahme ist bei den Konfigurationen A2, C2 und X1 besonder hoch. Für diese Konfigurationen ist die Fpr_I jeweils höher als bei den anderen, sodass die erhöhte Fehlerwahrscheinlichkeit offensichtlich durch das *correction level* ausgeglichen wird. Im extremen Fall von $\mathrm{Fpr}_I = 0{,}8$ bei Konfiguration X1 kommt es sogar beim *correction level* 0 zu einem Ausfall der Erkennung.

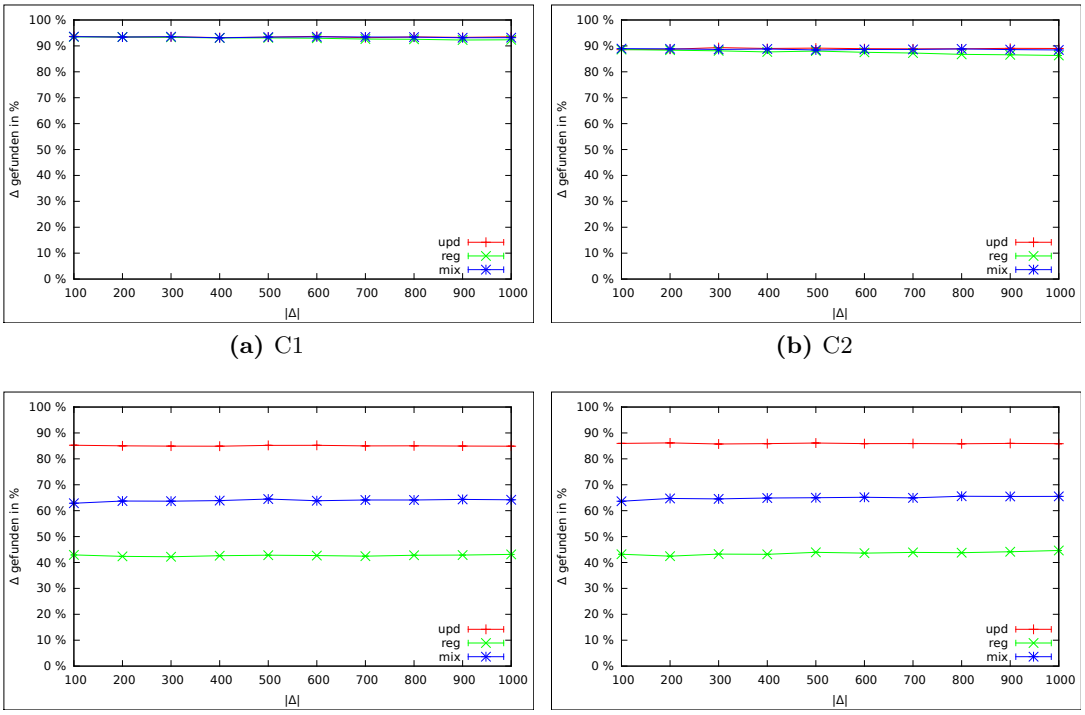
Der Grund für die Steigerung der Genauigkeit liegt in der sinkenden Wahrscheinlichkeit c aufeinanderfolgendende *false positives* zu generieren. Das *correction level* kann die Genauigkeit jedoch nur in dem Maße beeinflussen, wie diese vom *Bloom Filter* der Blätter beschränkt ist. Die maximale Genauigkeit von Konfiguration X1 ist eine Erkennungsgenauigkeit von 80% weil $\mathrm{Fpr}_B = 0{,}2$. Das *correction level* kann nur *false positives* in den inneren Knoten ausgleichen, nicht in den Blättern.

Der beschränkte Effekt des *correction level* bei A1 und C1 zeigt, dass die Fehlerwahrscheinlichkeit Fpr_I noch höher gewählt werden kann, um die Größe der ART-Datenstruktur noch weiter

zu verringern. Die Analyse von ART wird dennoch mit den bisher gewählten Konfigurationen fortgesetzt, weil die Konfigurationen mit noch höheren Fpr-Werten auch in einem der drei Cluster liegen werden und somit die gleichen Eigenschaften aufweisen wie die Ausgewählten.

Genauigkeitsanalyse

In den Abbildungen 4.16 ist die Genauigkeit der vier Konfigurationen abgebildet. In dem Szenario liegen die Fehler und Daten gleichmäßig verteilt vor. Die C_B-Konfigurationen (C1, C2) besitzen für alle drei Fehlertypen die gleiche Genauigkeit von 100% mit einem kleinen negativen Trend bei wachsender Differenzgröße. Die C_A-Konfigurationen (A1, A2) zeigen die gleiche Dreiteilung der Genauigkeit wie die *Bloom Filter* Abbildungen in 4.1 auf Seite 50. Die *reg* Genauigkeit entspricht bei den C_A Konfigurationen circa der Hälfte der *upd* Genauigkeit. Die Genauigkeit des gemischten Fehlertyps *mix* hängt direkt von den einzelnen Fehlertypen ab und unterliegt keinen mischungsbedingten Effekten.

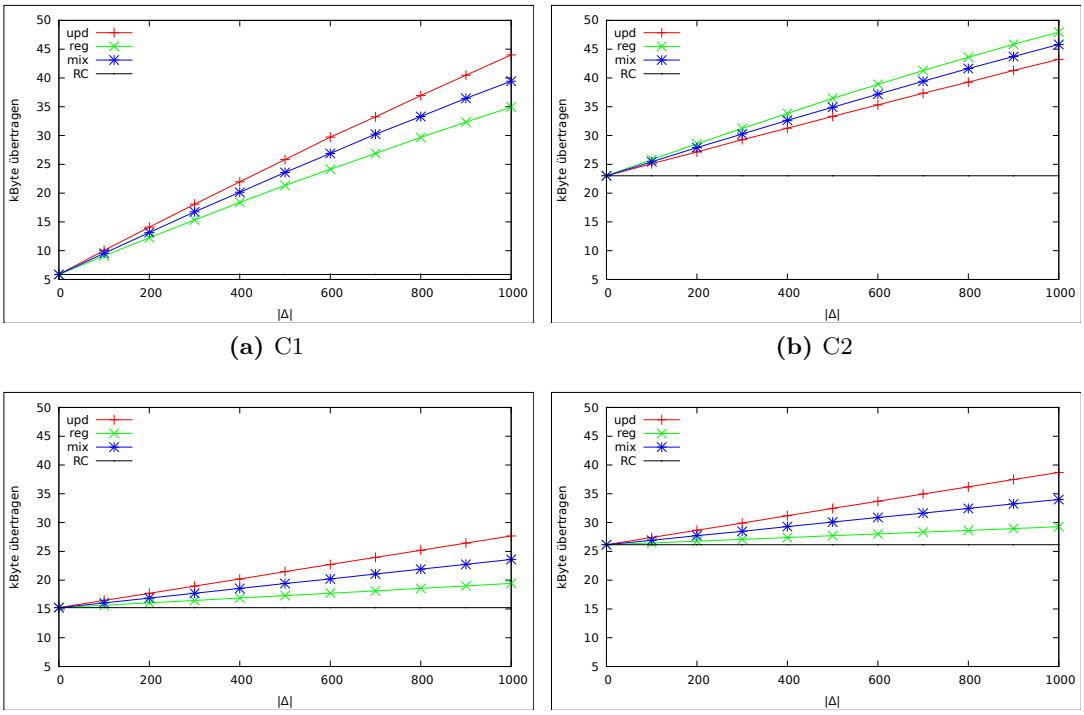


(a) C1 **(b)** C2

(c) A1 **(d)** A2

Abbildung 4.16: ART Genauigkeit in Abhängigkeit von $|\Delta|$

Die Übertragungskosten der vier Konfigurationen werden in den Abbildungen 4.17 dargestellt. In den Konfigurationen C1, A1 und A2 liegen die Übertragungskosten für den Fehlertyp *upd* über denen für *reg*, wohingegen bei C2 in 4.17b dieses Verhältnis umgekehrt ist. Die Ursache für C2 liegt



(a) C1 **(b)** C2

(c) A1 **(d)** A2

Abbildung 4.17: ART Übertragungskosten

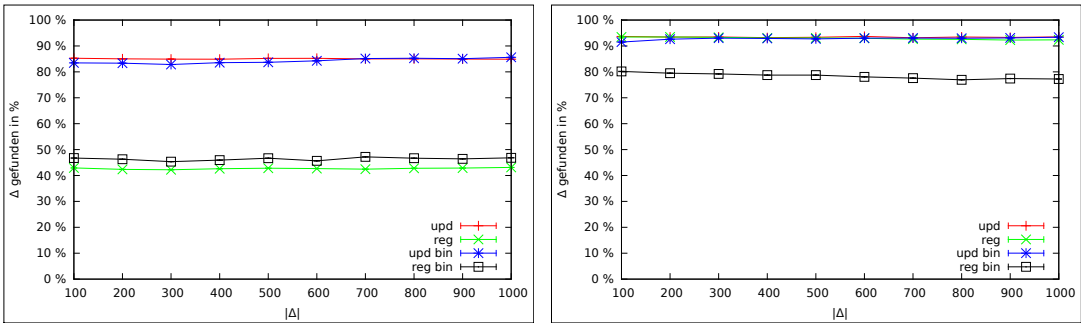
in der Kombination des Verzweigungsgrades 8 mit einer *Bucket*-Größe von 2. Demzufolge führen die fehlenden Kopien zu einer Verringerung der Tiefe von Teilbäumen, wodurch die Redundanz bei der Differenzerkennung steigt. Diese führt dann zu einer Erhöhung der Erkennungskosten bei *reg* Fehlern. Der Graph zum Fehlertyp *mix* liegt jeweils in der Mitte aufgrund der Mischung beider Fehlertypen und ihrer jeweiligen Redundanzkosten. Die Mischung der Fehler verursacht offensichtlich keinen erkennbaren Mehraufwand, sodass im Folgenden dieser Fehlertyp nicht weiter betrachtet wird.

In 4.17 kann noch eine weitere Beobachtung gemacht werden. Aus der zusätzlich als RC-Graph eingezeichneten Größe der ART-Datenstruktur kann abgelesen werden, dass eine Erhöhung des RC-Aufwands in Form einer größeren ART-Datenstruktur zu einer leichten Senkung der Redundanz führt. Dies kann zwischen C1 und C2 gut beobachtet werden. Dort sind die RC-Kosten von C2 viermal so groß wie die von C1. Ursache hierfür ist die erhöhte Baumtiefe von C2, die zu einer feineren Partitionierung des Intervalls führt die sich redundanzsenkend auswirkt. Dieser Effekt kann bei den C_A Konfigurationen A1 und A2 nicht beobachtet werden.

Zusammenfassend besitzen die C_B Konfigurationen eine nahe zu hundertprozentige Genauigkeit unabhängig vom Fehlertyp, sowie bei stärkerem Verzweigungsgrad einen geringeren Anstieg der

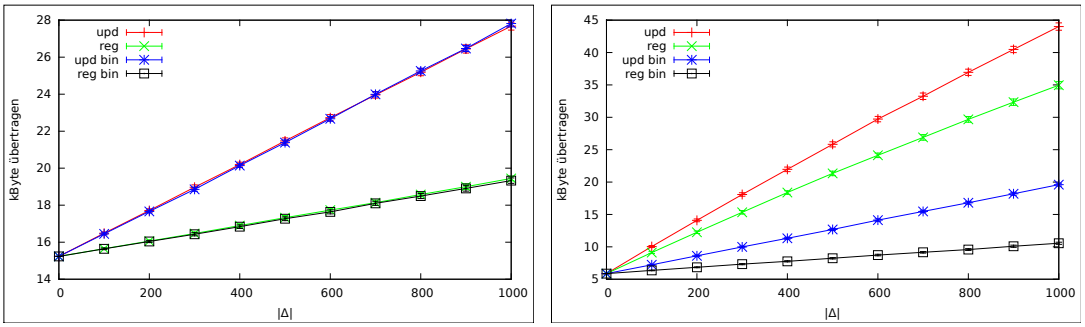
Übertragungskosten bei wachsendem Δ. Die C_A Konfigurationen besitzen unterschiedliche Genauigkeiten für die *reg* und *upd* Erkennung. Weiterhin sind die Übertragungskosten sowie deren Anstieg bei steigender Differenzgröße wesentlich geringer als bei den C_B Konfigurationen.

Fehlerverteilung



(a) A1 **(b)** C1

Abbildung 4.18: Auswirkung der binomialen Fehlerverteilung auf die Genauigkeit



(a) A1 **(b)** C1

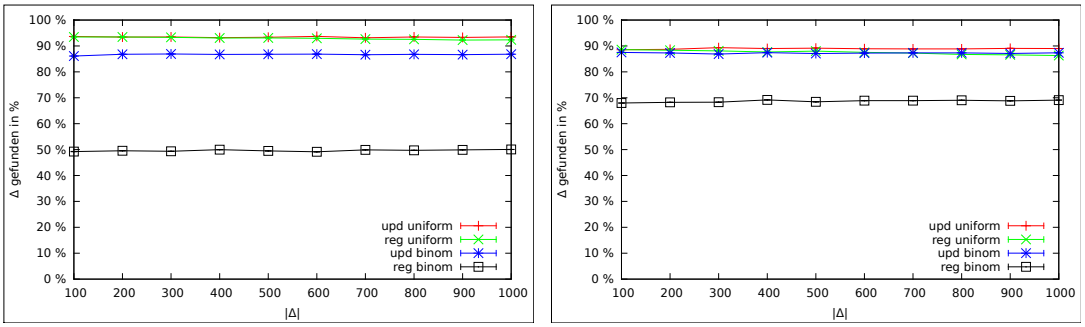
Abbildung 4.19: Auswirkung der binomialen Fehlerverteilung auf die Übertragungskosten

Die Auswirkung einer binomialen Fehlerverteilung wird auf die Genauigkeit in Abbildung 4.18 dargestellt sowie auf die Übertragungskosten in 4.19. Aufgrund identischer Werte ist jeweils nur eine Konfiguration der beiden Cluster abgebildet. Die Konfigurationen des Cluster C_A sind, wie die Abbildungen 4.19a und 4.18a zeigen, unabhängig von der Fehlerverteilung. Bei C_B-Konfigurationen senkt die Binomialverteilung die Erkennungsgenauigkeit um 10%. Der Grund des Genauigkeitsverlustes ist wahrscheinlich die *Bloom Filter* Einschränkung, die ART geerbt hat. Demzufolge ändert sich beim Initiator der *Merkle-Tree* Aufbau aufgrund der fehlenden Kopien, welche beim Abgleich mit dem *Bloom Filter* nicht verglichen werden können.

Zusammenfassend ist für die C_A Konfigurationen keine Änderung in Genauigkeit oder Über-

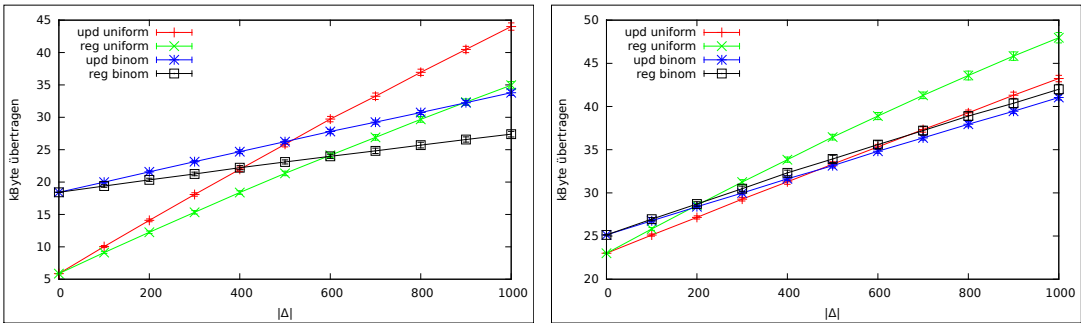
tragungskosten bei unterschiedlichen Fehlerverteilungen erkennbar. Für die C_B Konfigurationen ist zum einen eine Senkung der *reg* Genauigkeit sowie zum anderen eine starke Verringerung der Übertragungskosten bei binomial verteilten Fehlern zu beobachten.

Datenverteilung



(a) C1 **(b)** C2

Abbildung 4.20: Auswirkung einer binomialen Datenverteilung auf die Genauigkeit



(a) C1 **(b)** C2

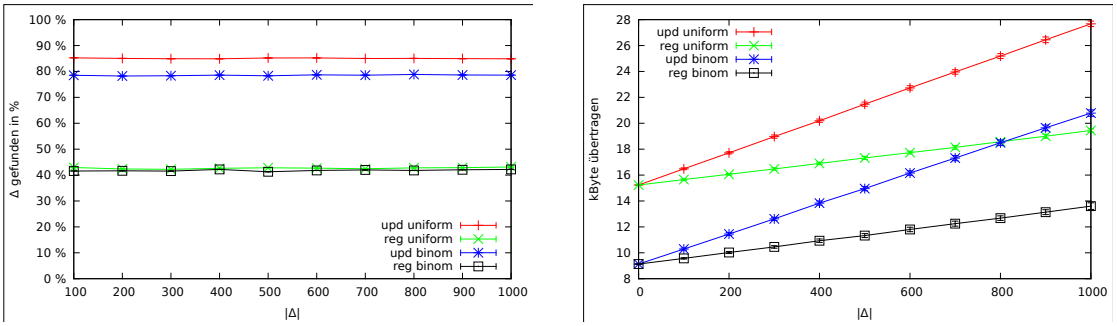
Abbildung 4.21: Auswirkung einer binomialen Datenverteilung auf die Übertragungskosten

Die Analyse der Datenverteilung stellt eine gleichmäßige Datenverteilung einer Binomialverteilung für die Fehlertypen *upd* und *reg* gegenüber. Die Analyse der Datenverteilung erfolgt zuerst an den C_B-Konfigurationen.

Die Abbildung 4.20a zeigt für C1 einen Genauigkeitsabfall von 40% bei der binomialen Datenverteilung gegenüber der gleichmäßigen im Falle des Fehlertyps *reg*. Im Fall von C2 fällt die Genauigkeit für *reg* um 10% bis 20% wie Abbildung 4.20b zeigt. Zur Erklärung dessen müssen die Übertragungskosten in 4.21a zusätzlich betrachtet werden. Diese zeigen, dass die Grundübertragungskosten bei $|\Delta| = 0$ von 6 auf leicht unter 20 gestiegen sind. Da in diesem Punkt keine RS-Kosten entstanden sind, entspricht dies der Größe der ART-Datenstruktur. Damit ist der Baum

von C1 sehr stark gewachsen und besitzt bei der binomialen Datenverteilung nun ein vielfaches an Knoten als zuvor. Die Abbildungen 4.17 von Seite 68 zeigten, dass ein flacher Anstieg der Übertragungskosten ein Zeichen geringer Redundanz ist. Daraus kann gefolgert werden, dass die gestiegene Anzahl an Knoten in C1 die Auslastung der *Buckets* auf kleiner Eins verringert hat und die Konfiguration C2 zu einer C_A-Konfiguration degradiert worden ist. Indiz hierfür ist weiterhin der geringe Anstieg der Übertragungskosten in 4.21a, der den C_A-Konfigurationen ähnlich ist.

Für C2 zeigt sich in 4.21b eine leichte Verringerung der Übertragungskosten im Falle einer binomialen Datenverteilung. Dies ist zurückzuführen auf eine stärkere Verzweigung innerhalb des Baumes, was zu einer Verringerung der *Bucket*-Auslastung führt. Indikator für die stärkere Verzweigung sind die leicht gestiegenen Kosten der ART-Datenstruktur, abzulesen im Punkt $|\Delta| = 0$. Diese steigen nur in Abhängigkeit der *Bloom Filter* Bitgröße. Diese hängt wiederum von der Anzahl der zu kodierenden Elemente ab, welche sich damit offensichtlich erhöht hat.



(a) Genauigkeit **(b)** Übertragungskosten

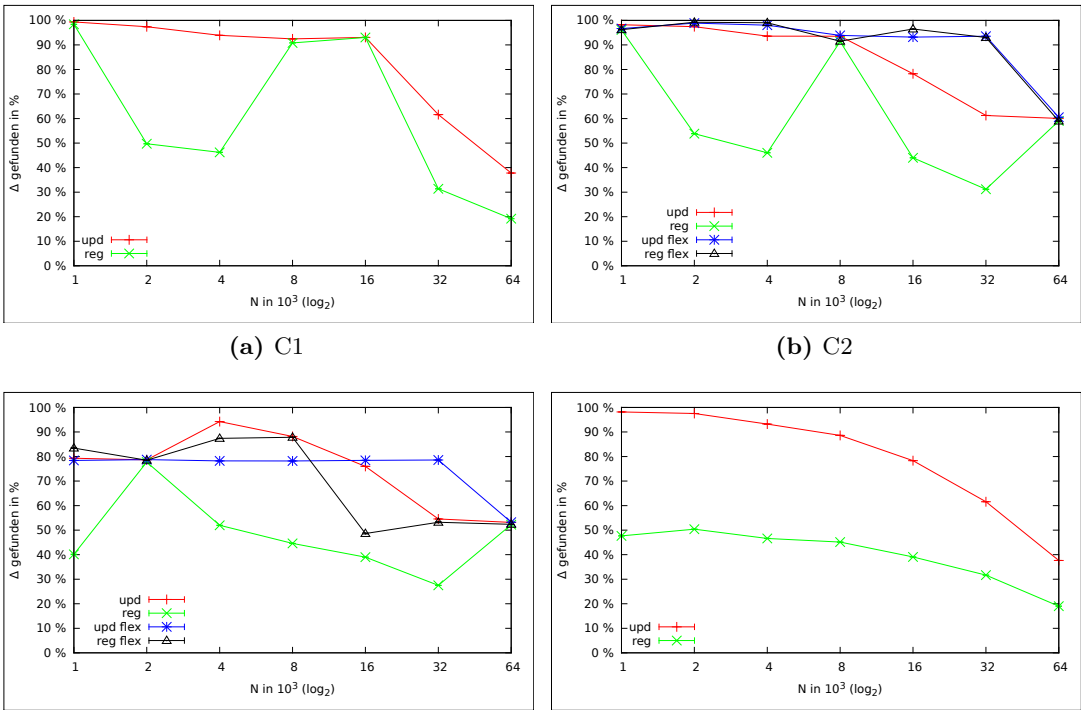
Abbildung 4.22: Auswirkung der binomialen Datenverteilung auf Konfiguration A1

Die C_A-Konfigurationen zeigen in Abbildung 4.22a einen leichten Rückgang der *upd* Genauigkeit um ca. 5% sowie einen sehr leichten bei *reg*. Die Übertragungskosten sinken dafür bei der Binomialverteilung der Daten um 40%, wie Abbildung 4.22b zeigt. Ursache für die stark Verringung der Übertragungskosten ist die kleine ART-Datenstruktur.

Zusammenfassend zeigt die Konfiguration C1 einen Wechsel in den C_A Cluster aufgrund der Änderung der Datenverteilung. Weiterhin nimmt bei den clusterkonstanten Konfigurationen C2 und A1 die Genauigkeit bei der Binomialverteilung leicht ab. Die Übertragungskosten steigen für die C_B Konfigurationen leicht an wohingegen diese bei den C_A Konfigurationen sinken.

Skalierbarkeit

Abschließend wird die Skalierbarkeit von ART analysiert, begonnen mit der Genauigkeit in den Abbildungen 4.23. Die Graphen *upd* und *reg* stellen in allen vier Abbildungen den Verlauf der Genauigkeit bei steigender Datenmenge dar. In den Abbildungen 4.23a bis 4.23c ist zu erkennen, dass die *reg* Genauigkeit teilweise stark schwankt. In Abbildung 4.23a beispielsweise wechselt die



(a) C1 **(b)** C2

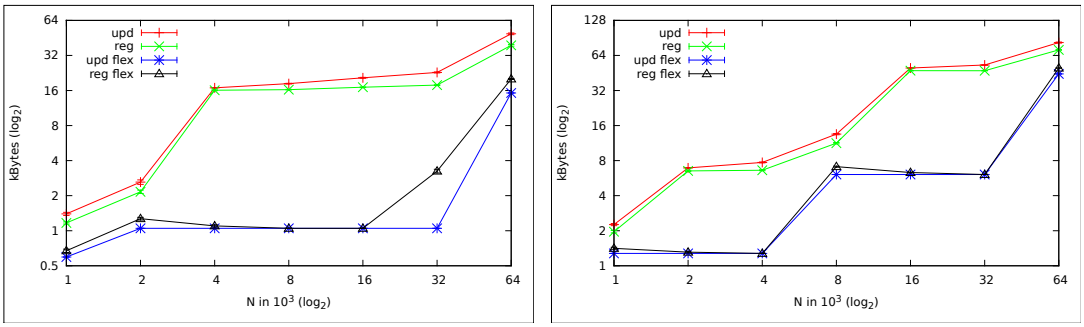
(c) A1 **(d)** A2

Abbildung 4.23: Skalierung der Genauigkeit

C_B-Konfiguration in den Cluster C_A in den Punkten 2, 4. Diese Wechsel resultieren aus dem sich ändernden Daten je *Bucket* Verhältnis.

Die Stabilität der ART Konfigurationen kann mit einer flexiblen Wahl von b erhöht werden. Die Graphen *upd flex* und *reg flex* in den Abbildungen 4.23b und 4.23c wählen b nach der beim *Merkle-Tree* vorgestellten Formel 4.1 (Seite 54) mit $\varepsilon = 1$. Daher wird für eine Datenmenge N ein Baum mit Verzweigungsgrad v verwendet, dessen *Bucket*-Größe automatisch so gewählt wird, dass die Tiefe um eins verringert wird. Die flexible Wahl von b mit der Baumtiefenverringerung von $\varepsilon = 1$ stellt mit hoher Wahrscheinlichkeit sicher, dass die resultierende Konfiguration dem Cluster C_B angehört.

Die *flex* Graphen mit der flexiblen Wahl von b sind wesentlich konstanter als jede mit fest gewählten b. Jedoch sinkt dennoch im Punkt $N = 64$ die Genauigkeit bei C2. Die *flex* Graphen treffen sich an diesem Punkt mit denen bei festem b. Diese Schnittpunkte, welche auch in $N = 8$ und $N = 1$ existieren, entstehen genau dort indenen das berechnete b identisch ist mit dem aus der Konfiguration. Bei genauerer Betrachtung der *flex* Graphen ist auch ohne die abrupten starken Genauigkeitsabfälle eine leicht sinkende Tendenz zu erkennen. In Abbildung 4.23d ist $b = 1$ fest und es ist dennoch ein Genauigkeitsverlust bei wachsendem N zu beobachten. Dies sowie die *flex*



(a) A1 **(b)** C2

Abbildung 4.24: Skalierung der Übertragungskosten

Graphen zeigen, dass die *Bucket*-Größe nicht die Ursache für das nicht skalierende Verhalten von ART ist.

In den Abbildungen 4.24 sind die Übertragungskosten von ART bei wachsendem N abgebildet. Diese steigen in Stufen linear an. Die starken abrupten Anstiege resultieren aus der Erhöhung der Baumtiefe, beispielsweise in 4.24b bei $N = 2$ bzw. $N = 16$ für die Graphen *upd* und *reg*. Moderate Zuwächse, wie zwischen $N = 2$ und $N = 8$ der gleichen Graphen, resultieren aus einer stärkeren Auslastung der *Buckets*.

Zusammenfassend skaliert ART bei der Genauigkeit nicht. Mit Hilfe der Konfiguration A2 mit $b = 1$ sowie mit den *flex*-Graphen wurde gezeigt, dass die *Bucket*-Größe nicht für das nicht skalierende Verhalten verantwortlich ist. Die *flex*-Graphen, deren b nach Formel 4.1 berechnete ist, zeigen jedoch, dass Clusterwechsel verhindert werden können. Das nicht skalieren von ART ist unerwartet, weil die zugrunde liegenden Strukturen *Bloom Filter* und *Merkle-Tree* skalieren.

Zusammenfassung

Zusammenfassend ist ART eine komplexe Datenstruktur deren Konfigurationen in zwei Cluster eingeteilt werden können, die jeweils unterschiedliche Eigenschaften besitzen. Der Cluster C_A enthält Konfigurationen deren Tripel je *Bucket* Verhältnis kleiner gleich Eins ist und der Cluster C_B enthält jene mit größer Eins.

Die C_A-Konfigurationen zeigen in ihrem Verhalten bei der Genauigkeit, Übertragungskosten und der Unempfindlichkeit gegenüber unterschiedlicher Fehlerverteilungen eine große Ähnlichkeit zum *Bloom Filter*. Die *reg* Erkennungsgenauigkeit entsprich der Hälfte der *upd* Genauigkeit und die Übertragungskosten sind gering, weil die Redundanz für *reg* den Wert 1 und für *upd* 1,5 besitzt. Liegen die Daten binomial verteilt vor, ist ein leichter Rückgang der Erkennungsgenauigkeit für beide Fehlertypen zu verzeichnen, obwohl der *Bloom Filter* unempfindlich gegenüber der Datenverteilung war. Weiterhin sinken die Übertragungskosten bei der Binomialverteilung der Daten erheblich,

sodass ein *Merkle-Tree* Vorteil bei den *Bloom Filter* ähnlichen Konfigurationen zu beobachten ist. Damit besitzt eine C_A Konfiguration nur den Vorteil der sinkenden Übertragungskosten bei binomial verteilten Daten gegenüber einem normalen *Bloom Filter*.

Cluster C_B-Konfigurationen gleichen in ihren Eigenschaften sehr stark dem *Merkle-Tree*. So besitzen diese eine vom Fehlertyp unabhängige Genauigkeit von annähernd 100%, sowie einen ähnlichen Anstieg der Übertragungskosten bei steigender Differenzgröße. Letzteres zeigt die erhöhte Redundanz gegenüber dem *Bloom Filter* und C_A Konfigurationen. Abweichend vom *Merkle-Tree* sinkt die *reg* Genaugikeit bei der binomialen Fehlerverteilung leicht. Ursache hierfür sind die fehlenden Knoten beim Initiator, die dieser nicht im *Bloom Filter* suchen kann. Bei der binomialen Datenverteilung konnte in Clusterwechsel von C_B zu C_A beobachtet werden sowie leicht steigende Übertragungskosten.

Ein großes Problem stellt der Wechsel der Konfigurationen zwischen den Clustern dar, wodurch die Eigenschaften von ART je nach Umgebung *Bloom Filter* oder *Merkle-Tree* ähnlich sind. Clusterwechsel wurden beim Wechsel der Datenverteilung sowie bei steigender Datenmenge beobachtet. Diese konnten erfolgreich verhindert werden, wenn b nach der Formel 4.1 für eine N, v Kombination berechnet wird.

ART benötigt nur zwei Mitteilungsrunden aus den gleichen Gründen wie der *Bloom Filter* und bietet aufgrund der geringen Latenz und der erhöhten Genauigkeit einen guten jedoch nicht skalierenden Ersatz für den *Merkle-Tree*. Das Problem der abnehmenden Genauigkeit bei größeren Datenmengen kann mit einem *divide-and-conquer* Ansatz umgangen werden, indem das gemeinsame Intervall in Teile mit kleineren Datenmengen zerlegt wird.

4.4.4 Repliken-Reparatur in Scalaris

Die Untersuchung des Repliken-Reparatur-Service in Scalaris stellt den Abschluss der Evaluation dar, indem die SR-Algorithmen in Kombination mit dem *Anti-Entropie* Protokoll in einer DHT fehlerhafte Repliken korrigieren. Dieser Abschnitt analysiert zum einen die Konvergenz der fehlerhaften Repliken in Scalaris gegen Null, zum anderen sollen die zuvor beobachteten Eigenschaften der SR-Algorithmen bestätigt werden.

Im Folgenden wird ein Scalaris-Ring mit 16 Knoten und gleichverteilten Daten verwendet, indem jeweils zwei Konfigurationen eines jeden SR-Algorithmus untersucht werden. Dadurch können die Auswirkungen der jeweiligen SR-Parameter im Gesamtsystem beobachtet werden. Die Tabelle 4.3 listet die ausgewählten Konfigurationen auf.

Die Messung erfolgt in Runden. Hierfür wird eine DHT mit $N = 40000$ Daten und $|\Delta| = 0.03N = 1200$ erstellt, in der 32 Runden durchgeführt werden. Eine Runde entspricht der Synchronisation eines DHT-Knotenpaares, wobei in den 32 Runden jeder Knoten zweimal als Initiator auftritt.

Der Abschnitt beginnt mit der Analyse der Algorithmen bei gleichmäßiger Knotenverteilung. Anschließend wird die zufällige der gleichmäßigen Knotenverteilung gegenübergestellt, um den Einfluss der Knotenverteilung auf die Leistungsfähigkeit zu bestimmen. Darauf folgt die Beobachtung einer binomialen Fehlerverteilung auf die Knotenverteilungen und die Zusammenfassung der Ergebnisse.

Die Auswertung der Algorithmen für die zwei Knoten- und Fehlerverteilungen erfordert eine verkürzte Notation *ring*(Knotenverteilung, Fehlerverteilung). Die Knotenverteilung besitzt das Kürzel u für die gleichmäßige und z für die zufällige Verteilung. Die Fehlerverteilung verwendet b für eine binomiale und u für eine gleichmäßige Verteilung.

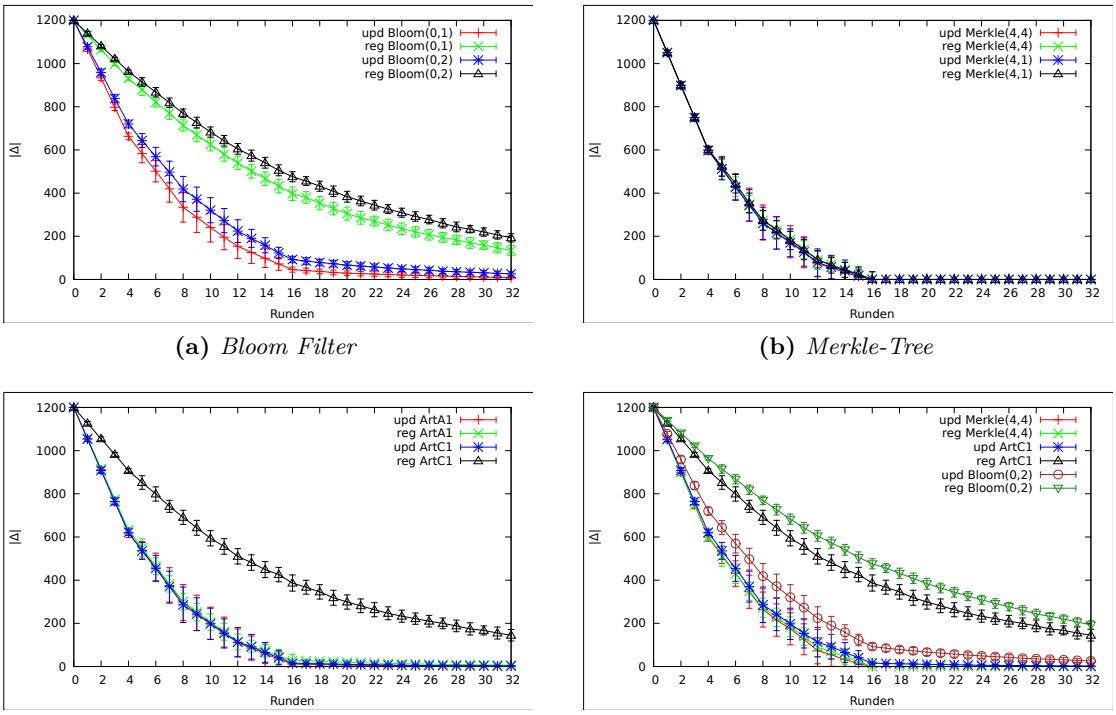
Name	**Konfiguration**			
Bloom(0,1)	Fpr = 0,1			
Bloom(0,2)	Fpr = 0,2			
Merkle(4,4)	$v = 4$	$b = 4$		
Merkle(4,1)	$v = 4$	$b = 1$		
ArtA1	$v = 32$	$b = 2$	$\text{Fpr}_I = 0{,}01$	$\text{Fpr}_B = 0{,}2$
ArtC1	$v = 16$	$b = 4$	$\text{Fpr}_I = 0{,}001$	$\text{Fpr}_B = 0{,}01$

Tabelle 4.3: Untersuchte Konfigurationen der SR-Algorithmen

Genauigkeit bei gleichmäßiger Knotenverteilung

In der systemweiten Analyse wird die Darstellung der Abbildungen leicht geändert, auf der x-Achse wird die jeweilige Runde dargestellt und auf der y-Achse die verbleibenden fehlerhaften Kopien oder die Übertragungskosten. Die x-Achse spiegelt damit die Anzahl der Synchronisationen wider.

Die Abbildungen 4.25 stellen jeweils die Genauigkeit der zwei Konfigurationen eines SR-Algorithmus gegenüber. Die vierte Abbildung 4.25d zeigt den Vergleich der Algorithmen. In dem beobachteten Szenario sind Knoten und Fehler gleichverteilt. Die *Bloom Filter* in Abbildung



(a) *Bloom Filter* **(b)** *Merkle-Tree*

(c) ART **(d)** Vergleich

Abbildung 4.25: Genauigkeit

4.25a zeigen in beiden Konfigurationen eine wesentlich langsamere Konvergenz bei der Auflösung von *reg* Fehlern gegenüber den *upd* Fehlern. Dies resultiert aus der um die Hälfte geringeren *reg*-Genauigkeit der *Bloom Filter*. Weiterhin ist der Abstand zwischen den beiden *upd* Graphen sowie zwischen den *reg* Graphen eher klein trotz des Unterschieds in der Fehlerwahrscheinlichkeit von 10%. Dies zeigt, dass die Fehlerwahrscheinlichkeit nur einen leichten Einfluss auf die Konvergenzgeschwindigkeit besitzt.

Der *Merkle-Tree* wird in Abbildung 4.25b in seinen zwei Konfigurationen gezeigt, zwischen denen fast kein Unterschied zu erkennen ist. Dies bestätigt die Unabhängigkeit der Genauigkeit des *Merkle-Tree* von der *Bucket*-Größe, da die zwei Konfigurationen sich nur in dieser unterscheiden. Die hohe Genauigkeit des *Merkle-Tree* führt desweiteren zu einer vollständigen Synchronisation des Rings nachdem jeder Knoten mindestens einmal in einer Synchronisation beteiligt war.

In Abbildung 4.25c sind die beiden ART Konfigurationen der Cluster C_A und C_B abgebildet. Die Konfiguration ArtC1, welche eigentlich die Eigenschaften des *Merkle-Trees* besitzen sollte, weist die Erkennungsgenauigkeit eines *Bloom Filter* im Fehlertyp *reg* auf. Dies zeigt eine Umkehr der Clusterzugehörigkeit der Konfigurationen A1 und C1, sodass C1 nun zu Cluster C_A gehört und A1 zu Cluster C_B. Der Clusterwechsel ist auf die Änderung des Verhältnisses der Daten je

Bucket zurückzuführen. Weiterhin kann beobachtet werden, dass A1 alle *upd* und *reg* Fehler in 16 Runden findet, wohingegen C1 dies nur für *upd* erzielt.

In Abbildung 4.25d wird die Genauigkeit der SR-Algorithmen an je einem Vertreter gegenübergestellt. Der Abbildung zufolge ist die Konvergenzgeschwindigkeit des *Merkle-Tree* in beiden Fehlertypen am höchsten gefolgt von ART. Der *Bloom Filter* besitzt die geringste Konvergenzgeschwindigkeit. Der *Merkle-Tree* sowie ART erreichen eine hundertprozentige Auflösung aller Differenzen in 16 Runden, wohingegen der *Bloom Filter* bei *upd* ca. 91% erreicht und bei *reg* sogar nur 59%. Für ART wurde die *Bloom Filter* ähnliche Konfiguration ausgewählt, um die Genauigkeit der *reg* Erkennung mit dem *Bloom Filter* zu vergleichen. Es ist deutlich zu sehen, dass ART hier die bessere Erkennungsgenauigkeit besitzt. ART ist jedoch maximal so genau wie ein *Bloom Filter* der gleichen Größe [18]. Daher resultiert der Genauigkeitsunterschied bei *reg* aus der erhöhten Größe der ART-Datenstruktur, wie später in Abbildung 4.26 gezeigt wird. ART besitzt trotz der eben genannten Einschränkung eine wesentlich größere Genauigkeit bei *upd* aufgrund des *Bucketing*.

Die Fehlerbalken in den Abbildungen 4.25 stellen die Standardabweichung σ der gefundenen fehlerhaften Kopien dar. Diese zeigen bei allen SR-Algorithmen eine anfangs geringe Schwankung in der Erkennungszahl, welche langsam ansteigt bis diese zwischen Runde 11 bis 15 am größten ist und danach wieder absinkt. Dieses Verhalten spiegelt die Wahrscheinlichkeit des Auffindens eines Synchronisationspartners wieder, der viele fehlerhafte Kopien besitzt. Diese Wahrscheinlichkeit ist zu Beginn hoch, weil alle Knoten in etwa die gleiche Fehleranzahl besitzen. Die Wahrscheinlichkeit sinkt im Weiteren durch die verringerte Anzahl von Knoten mit hoher Fehleranzahl. Der Zeitraum, in dem wenige Knoten mit vielen Fehlern existieren, besitzt damit die höchste Schwankung, was sich in den Abbildungen widergespiegelt. Letztlich wird die Differenz im Gesamtsystem immer kleiner und die Anzahl der gefundenen fehlerhaften Kopien sinkt, sodass die Partnerwahl nur noch zu sehr kleinen Schwankungen führt.

Übertragungskosten bei gleichmäßiger Knotenverteilung

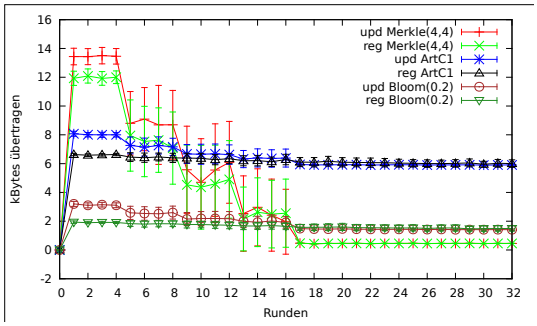
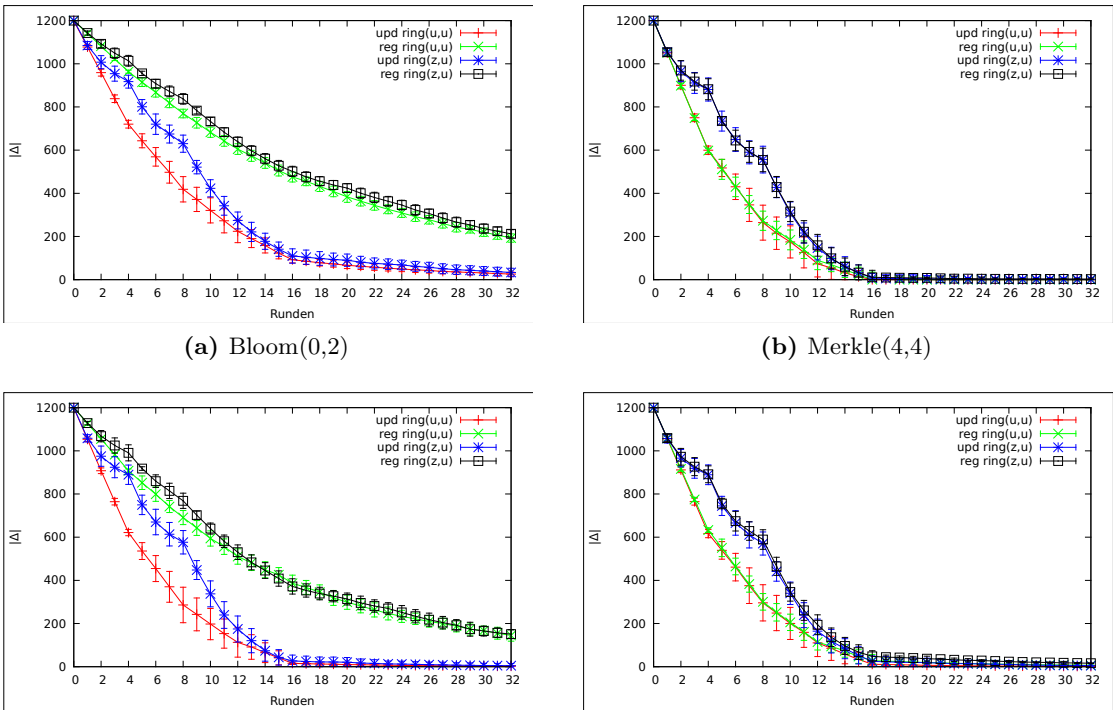


Abbildung 4.26: Bandbreite

Die Übertragungskosten für unterschiedliche Konfigurationen der SR-Algorithmen wurden bereits betrachtet, weshalb in Abbildung 4.26 jeweils eine Konfiguration jedes SR-Algorithmus dargestellt

ist. Die Abbildung zeigt den Verlauf der Übertragungskosten. Es ist bei jedem Graphen eine stufenweise Verringerung der Kosten zu beobachten, wobei die Stufengröße bei *Merkle-Tree* am höchsten ist und beim *Bloom Filter* und ART klein ist. Im Maximum betragen die Übertragungskosten des *Bloom Filter* 16% gegenüber denen des *Merkle-Tree*. Die sinkenden Kosten resultieren aus der fallenden Menge an Differenzen. Die Stufen repräsentieren die Quadranten des Rings und die damit verbundenen Wahrscheinlichkeiten eine bestimmte Anzahl von Fehlern in einer Runde zu finden. Der *Merkle-Tree* verursacht ab Runde 16 mit $|\Delta| = 0$ (nach Abb. 4.25b) minimale Übertragungskosten und zeigt damit, dass der Vergleich der obersten Baumebene ausreicht, um die Synchronität zweier Knoten festzustellen. Die Graphen von ART und *Bloom Filter* besitzen die gleiche Form jedoch auf unterschiedlichen Niveau. Die Gegenüberstellung der Übertragungskosten der einzelnen Algorithmen zeigt, das das Auffinden von Differenzen mittels *Merkle-Tree* innerhalb einer Runde am teuersten ist. Die verursachten Übertragungskosten des *Bloom Filters* sowie von ART sind relativ konstant, sodass die Gefahr einer *positive feedback loop* (siehe 2.2.5) verringert ist und die Bandbreitennutzung kaum Spitzen aufweist.

Auswirkung der Knotenverteilung



(a) Bloom(0,2) **(b)** Merkle(4,4)

(c) ArtC1 **(d)** ArtA1

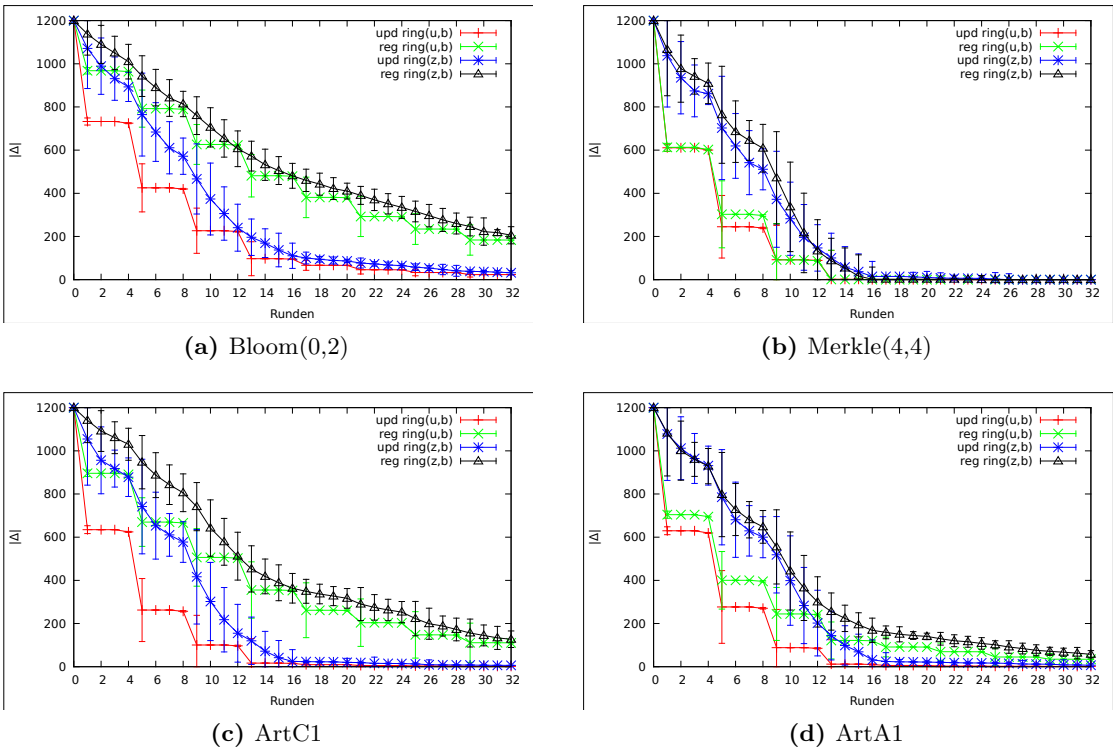
Abbildung 4.27: Auswirkung der Knotenverteilung auf die Genauigkeit

Die Abbildungen 4.27 zeigen bei gleichmäßiger Fehlerverteilung die Auswirkung der Knotenverteilung auf die Genauigkeit der einzelnen SR-Algorithmen und Konfigurationen. Es ist bei zufälliger Knotenverteilung mit einem Rückgang der Konvergenzgeschwindigkeit zu rechnen, weil die Größe der gemeinsamen Intervalle gesunken ist.

Die Senkung der Konvergenzgeschwindigkeit ist für alle Konfigurationen in den Abbildungen 4.27 erkennbar. Am stärksten zeigt sich diese zwischen den Runden 2 und 12. Die Verringerung ist beim *Merkle-Tree* und ArtA1 auf beide Fehlertypen identisch. Beim *Bloom Filter* und ArtC1 hingegen ist die Verringerung der Konvergenzgeschwindigkeit bei *reg* geringer ausgeprägt als bei *upd*. Ursache hierfür ist wahrscheinlich die halbierte Genauigkeit bei *reg* Fehlern, sodass die zufällige Knotenverteilung sich dadurch kleiner darstellt.

Insgesamt ist die Auswirkung der zufälligen Knotenverteilung auf die Konvergenzgeschwindigkeit deutlich, sodass der Unterschied in der achten Runde in allen vier Abbildungen 200 Differenzen beträgt. Damit ist die Konvergenzgeschwindigkeit gegenüber der gleichmäßigen Knotenverteilung um circa 30% verringert.

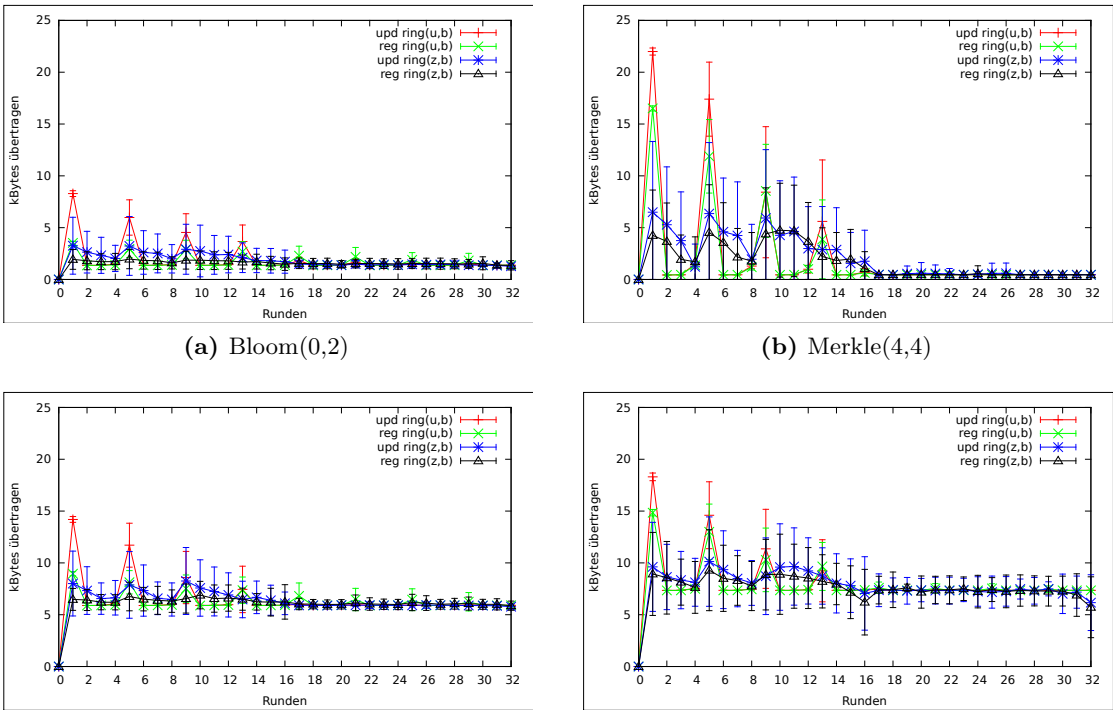
Auswirkung der Fehlerverteilung



(a) Bloom(0,2) **(b)** Merkle(4,4)

(c) ArtC1 **(d)** ArtA1

Abbildung 4.28: Auswirkung der Fehlerverteilung auf die Genauigkeit



(a) Bloom(0,2) **(b)** Merkle(4,4)

(c) ArtC1 **(d)** ArtA1

Abbildung 4.29: Auswirkung der Fehlerverteilung auf die Übertragungskosten

Eine binomiale Fehlerverteilung erzeugt im Scalaris-Ring in jedem Quadranten eine lokale Häufung der Fehler. Dieses Szenario soll den praktischen Fall sogenannter „hot spots“ abbilden. Bei diesen handelt es sich um langsame Knoten, die dadurch wesentlich häufiger fehlerhafte Kopien besitzen als andere Knoten des verteilten Systems. Der Effekt der binomialen Verteilung wird verstärkt durch die Symmetrie, sodass bei $p = 0{,}2$ jeweils die Knoten am Anfang eines Quadranten *hot spots* bilden.

Die Abbildungen 4.28 stellen die Genauigkeit der Algorithmen bei gleichmäßiger und zufälliger Knotenverteilung gegenüber. In dem Szenario liegen die Fehler bei allen Graphen binomial verteilt vor. Bei dem Vergleich der Abbildungen 4.28 mit denen bei gleichmäßiger Fehlerverteilung in 4.27 fällt die starke Ähnlichkeit beider auf.

Die Graphen ring(u,b) sind stark stufenartig gegenüber den ring(u,u) Graphen aus 4.27. Die Stufen in den Runden 1, 5, 9, 13 resultieren aus den gefundenen *hot spots*, die zu einer abrupten Verringerung der Differenz führen. Das Auffinden der *hot spots* nach jeweils 4 Runden wird von der festen Reihenfolge der Initiatoren im Rundensystem hervorgerufen. Bei der gleichmäßigen Knotenverteilung ist jeder vierte Knoten für den Anfang eines Schlüsselquadranten verantwortlich, der bei der binomialen Verteilung die meisten fehlerhaften Kopien besitzt.

Die Graphen ring(z,b) verlaufen fast genauso wie jene ring(z,u) mit dem Unterschied einer wesentlich höheren Standardabweichung. Diese resultiert bei der zufälligen Knotenverteilung aus der Möglichkeit jede Runde einen *hot spot* zu finden. Weiterhin bleibt das Verhältnis zwischen gleichmäßiger und zufälliger Knotenverteilung auf bei der binomialen Fehlerverteilung erhalten sodass die Genauigkeit bei zufälliger Knotenverteilung im Mittel schlechter ist als bei der gleichmäßigen.

Bei der Betrachtung der Übertragungskosten in den Abbildungen 4.29 sind die *hot spots* als Spitzen deutlich bei den zuvor genannten Runden zu beobachten. Der *Bloom Filter* in Abb. 4.29a sowie ArtC1 besitzen kleine Spitzen und zeigen relativ gleichbleibende Übertragungskosten, die nie einen minimalen Wert unterschreiten. Die Höhe dieser unteren Schwelle hängt von der Größe des *Bloom Filters* ab und beträgt 5 kByte in Abbildung 4.29c oder circa 2 kByte in 4.29a.

Die Spitzen in den Übertragungskosten hängen von der Redundanz ab, sodass die Spitzen bei hoher Redundanz höher sind als bei niedriger. Dies zeigte bereits die einzel Betrachtung des *Merkle-Trees*. Dadurch besitzen der *Bloom Filter* sowie die dem ähnliche ArtC1 Konfiguration wesentlich kleinere Spitzen als der *Merkle-Tree* sowie dem ähnliche ArtA1 Konfiguration.

Zusammenfassend ist die Konvergenzgeschwindigkeit bei zufälliger Knotenverteilung leicht geringer als bei der gleichmäßigen Knotenverteilung. Die binomiale Fehlerverteilung erhöht die Standardabweichung aufgrund der latenten Möglichkeit des auffindens eines *hot spots*. Aufgrund der ungleichen Fehlerverteilung ist die Konvergenzgeschwindigkeit leicht höher als bei der gleichmäßigen Fehlerverteilung. Dies ist in Abbildung 4.29b zu beobachten, indem nach 13 Runden schon eine völlige Synchronität des Systems vorliegen kann.

Zusammenfassung

Die Konvergenz der SR-Algorithmen in einer DHT wurde vom *Anti-Entropie* Protokoll garantiert. Die Konvergenzgeschwindigkeit wird am stärksten von der Genauigkeit des SR-Algorithmus beeinflusst, sodass der *Bloom Filter* bei *reg*-Fehlern die geringste Konvergenzgeschwindigkeit vorweist. Die zufällige Knotenverteilung verringert bei allen SR-Algorithmen die Konvergenzgeschwindigkeit aufgrund der kleineren gemeinsamen Intervalle um bis zu 30% gegenüber gleichmäßig verteilten Knoten. Eine binomiale Fehlerverteilung erhöht die Konvergenzgeschwindigkeit leicht, weil weniger Synchronisationen notwendig sind um alle Differenzen aufzulösen. Der *Merkle-Tree* sowie ART führen wesentlich schneller zu einer vollständigen Auflösung aller fehlerhaften Kopien als der *Bloom Filter*, dessen Konvergenzgeschwindigkeit mit sinkender Fehleranzahl stark abnimmt.

Die vorgestellte Lösung kann unter Umständen in Scalaris zum Verlust der Konsistenz führen. Demnach regeneriert der Repliken-Reparatur-Service verlorene Kopien auch bei einer Partitionierung des DHT-Rings. Befand sich in der Partition ausschließlich eine veraltete Version, kann nach der Regeneration eine Mehrheit veralteter Kopien entstehen, wodurch Anfragen inkonsistent beantwortet werden können. Dies kann vermieden werden, wenn Kopien nur dann regeneriert werden wenn auch ein Quorum gebildet werden kann. Dies führt jedoch zu einer starken Steigerung der Regenerationskosten.

5 Schlussbemerkungen

5.1 Zusammenfassung

In Kapitel 2 wurde der Aufbau einer *distributed hash table* beschrieben, die die Basis für einen effizienten verteilten Schlüssel-Wert-Speicher darstellt. Verteilte Datenbanksysteme benötigen Redundanz, um die Verfügbarkeit der Daten trotz wechselnder Systemkomponenten sicherzustellen. Das Design einer Redundanzschicht für DHTs wurde ausführlich beschrieben sowie die Notwendigkeit der Aufrechterhaltung der Redundanz. Die Redundanzaufrechterhaltung erfordert die Regeneration verlorener Kopien sowie der Aktualisierung veralteter Kopien.

Um effizient verlorene oder veraltete Kopien erkennen zu können, werden zumeist *Set Reconciliation* Algorithmen verwendet. Vier dieser Algorithmen sowie einige Varianten wurden detailliert vorgestellt und in einer allgemeinen Klassifikation eingeordnet. In zwei Abschnitten wurde jeweils der Stand der Forschung zur Regeneration und zur Aktualisierung dargelegt sowie weiterführende Probleme beschrieben. Weiterhin wurde dargelegt, dass nur in der neusten Version regeneriert werden darf und die Anzahl der Kopien bei verschiedenen Quorumsystemen eine obere Schranke nicht überschreiten darf. Bei der Aktualisierung wurden insbesondere Kommunikationsstrategien vorgestellt, die das Auffinden von veralteten oder verlorenen Repliken ermöglichen, ohne das jeder Knoten mit allen anderen Kontakt aufnehmen muss.

Im Kapitel 3 wurde das Systemmodell definiert, welches für die Evaluation verwendet wurde. Anschließend wurde die Architektur des Repliken-Reparatur-Service erläutert, der aus einem *Set Reconciliation* Dienst, einem Aktualisierungsdienst sowie dem *Anti-Entropie*-Protokoll besteht. Es wurde das Design dieser Komponenten beschrieben sowie insbesondere die Umsetzung der drei ausgewählten *Set Reconciliation* Algorithmen, *Bloom Filter*, *Merkle-Tree* und ART.

In der Evaluation Kapitel 4 wurde der Aufbau der Simulationsumgebung beschrieben, sowie die Szenarien in denen die Leistungsfähigkeit der SR-Algorithmen gemessen wurde. Weiterhin wurden die Metriken definiert, anhand derer die Leistungsfähigkeit gemessen und verglichen wird. Die Simulation beginnt mit einer Einzelbeobachtung der SR-Algorithmen. In dieser wurden zuerst die Parameter studiert sowie nachfolgend die Stärken und Schwächen der Algorithmen in verschiedenen Szenarien. Anschließend wurden die Algorithmen zusammen mit dem *Anti-Entropie* Protokoll in Scalaris beobachtet, um deren Eigenschaften in einer konkreten DHT zu untersuchen.

Die Evaluation zeigte, dass der *Merkle-Tree* die höchste Genauigkeit sowie die schnellste Konvergenzgeschwindigkeit bei der Erkennung verlorener sowie veralteter Kopien aufweist. Dieser führt

spätestens nach einer initiierten Synchronisation eines jeden Knotens zur vollständigen Synchronität in der DHT. Der *Bloom Filter* besitzt zwar eine frei wählbare Genauigkeit, konvergiert jedoch gerade bei der Erkennung verlorener Kopien um die Hälfte langsamer, weil nur auf einer Seite des Synchronisationspaares verlorene Kopien identifiziert werden können.

Der *Bloom Filter* verursacht eine relativ konstante Last in der Höhe seiner Bitgröße, welche leicht erhöht wird durch die Übertragung identifizierter Differenzen. Diese Kontinuität der Übertragungskosten behält der *Bloom Filter* auch bei unterschiedlichen Daten- und Fehlerverteilungen bei. Der *Merkle-Tree* hingegen verursacht stäkere Schwankungen in den Übertragungskosten, die von der Differenzgröße, der Fehlerverteilung sowie von der *Bucket*-Größe abhängen. Die *Bucket*-Größe beeinflusst als einziger Parameter die Redundanz und fügt jedem identifizierten Fehler einen konstanten Mehraufwand hinzu. Die starken Schwankungen resulieren aus dem Prinzip des *Merkle-Tree*, der im besten Fall nur eine Signatur übertragen muss und im schlechtesten Fall alle Knoten sowie alle Daten überträgt. Der Anstieg der Übertragungskosten in Abhängigkeit von der Differenzgröße hängt von der Verteilung der differierenden Elemente ab. Die Kosten steigen bei binomial verteilten Fehlern wesentlich langsamer als bei gleich verteilten.

ART besitzt je nach Konfiguration die Eigenschaften eines *Bloom Filter* (C_A) oder die eines *Merkle-Tree* (C_B). Eine Formel zur Überprüfung wurde angegeben, sodass der Nutzer das Verhalten von ART bestimmen kann. Eine C_A-Konfiguration ist in Genauigkeit und Übertragungskosten fast identisch mit einem *Bloom Filter*, sodass diese keine Vorteile gegenüber dem *Bloom Filter* zeigt. Die C_B-Konfigurationen besitzen eine leicht schlechtere Genauigkeit gegenüber dem *Merkle-Tree* aufgrund des *Bloom Filter* für den Blattabgleich, dessen Fehlerwahrscheinlichkeit größer ist als eine Hashkollision bei dem Vergleich zweier Knotensignaturen. ART benötigt nur eine Mitteilungsrunde um die Differenz zu bestimmen, wohingegen der *Merkle-Tree* $\log_v(\frac{N}{b})$ Runden benötigt. Damit besitzen C_B-Konfigurationen eine wesentlich geringere Latenz zu Lasten einer leichten Verringerung der Genauigkeit. Aufgrund der Verwendung von *Bloom Filtern* verursacht ART bei dem Vergleich zweier synchroner Knoten mehr Übertragungskosten als ein *Merkle-Tree*. Das größte Problem bei der Verwendung von ART ist die fehlende Skalierbarkeit, sodass die Genauigkeit mit wachsender Datenmenge abnimmt.

Die Analyse der SR-Algorithmen in Scalaris bestätigte die zuvor einzeln beobachteten Eigenschaften der SR-Algorithmen und zeigte die Konvergenz. Der *Merkle-Tree* sowie die C_B ART-Konfigurationen erreichten spätestens nach einer initiierten Runde eines jeden Knotens eine vollständige Synchronität aller Repliken. Der *Bloom Filter* erreichte in diesem Zeitraum eine Synchronität von über 90% bei veralteten Kopien und 80% bei veralteten. Weiterhin sank die Konvergenzgeschwindigkeit je weniger Differenzen im System existierten. Die Betrachtung des praktischen Falls der zufälligen Knotenverteilung zeigte, dass die Konvergenzgeschwindigkeit nur leicht verringert wurde gegenüber einer gleichmäßigen Verteilung der Knoten. Die Konvergenzgeschwindigkeit ändert sich im Mittel auch nicht, wenn nur einige wenige Knoten, sogenannte *hot spots*, fehlerhafte Kopien besitzen. Im Allgemeinen zeigte sich, dass die Konvergenzgeschwindigkeit

von der Knotenverteilung sowie der Genauigkeit der SR-Algorithmen abhängt.

Der *Merkle-Tree* arbeitet effizient, wenn auf einem Knoten weniger als 10% der Daten veraltet oder verloren sind. In der Praxis sind die fehlerhaften Elemente innerhalb eines Knotens eher gleich verteilt als binomial, wodurch der *Merkle-Tree* nicht in einer optimalen Umgebung arbeitet. Der *Bloom Filter* ist unabhängig von der Verteilung der Fehler und Daten und bietet eine gute Genauigkeit bei geringeren Übertragungskosten. Ein Nachteil des *Bloom Filter* ist die höhere Grundlast der Übertragungskosten gegenüber dem *Merkle-Tree* sodass der Vergleich synchroner Knoten teurer ist als mit dem *Merkle-Tree*. Weiterhin werden kleinere Differenzen zwischen zwei Knoten nur schlecht vom *Bloom Filter* erkannt und die Konvergenzgeschwindigkeit für die Erkennung verlorener Kopien ist halbiert gegenüber der Erkennung veralteter Kopien. Die *Bloom Filter* ähnlichen Konfigurationen von ART bieten keinen Vorteil, wohingegen die C_B Konfigurationen eine bessere Latenz als der *Merkle-Tree* bieten bei gleicher Genauigkeit.

5.2 Ausblick

Während der Implementation und Evaluation der Algorithmen boten sich Möglichkeiten und Ideen zur weiteren Optimierung der Algorithmen an. Diese wurden jedoch aufgrund der damit steigenden Komplexität für spätere Untersuchungen offen gelassen.

Die RS-Phase des *Bloom Filters* kann optimiert werden, indem Bandbreite für Latenz eingetauscht wird. Die Bestimmung von $Upd(A,B)$ und $Upd(B,A)$ aus Δ erfolgt über das Senden aller KVV-Tripel aus Δ an den Partner B. Würden nur die Schlüssel-Versions Tupel vom Initiator A an B übertragen werden, könnte B die Menge $Upd(A,B)$ bestimmen und die entsprechenden Tripel A zusenden. Daraufhin kann A die Menge $Upd(B,A)$ bestimmen und diese KVV-Tripel an B senden. Dadurch wird die doppelte Übermittlung der Werte von $Upd(A,B)$ vermieden zu lasten einer weiteren Mitteilungsrunde.

Algorithmen zur Abschätzung der Differenzgröße zweier Mengen, wie *min-wise-sketches* [17] oder der *Strata Estimator* [34], können den Aufwand der Repliken-Reparatur verringern. Ist die Größe der Mengendifferenz bekannt, kann der jeweils geeignete SR-Algorithmus zur Synchronisation verwendet werden. Bei einer kleinen Differenz ist die Verwendung des *Merkle-Tree* vorteilhaft, wohingegen bei einer großen der *Bloom Filter* effektiver ist. Weiterhin kann bei einer kleinen Differenz die Fehlerwahrscheinlichkeit des *Bloom Filters* erhöht werden um eine hundertprozentige Synchronisation anzustreben.

Eine weitere Möglichkeit den Aufwand der Repliken-Reparatur zu verringern ist die Synchronisation eines zufälligen Teilintervalls des gemeinsamen Intervalls. Die zu synchronisierende Datenmenge wird verringert obwohl über kurz oder lang alle Daten einmal synchronisiert werden. Diese Anpassung erfolgt zu lasten der Konvergenzgeschwindigkeit, wodurch die Verweildauer von veralteten und fehlenden Repliken erhöht wird.

Literaturverzeichnis

[1] *Unified Modeling Language*. Object Management Group. `http://www.omg.org/spec/UML/2.2/`. – besucht am 19.06.2012 (Zitiert auf Seite 37)

[2] Agrawal, Divyakant ; Abbadi, Amr E.: The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data. In: *VLDB*, Morgan Kaufmann, 1990. – ISBN 1–55860–149–X, S. 243–254 (Zitiert auf Seiten 11 und 12)

[3] Aguilera, Marcos K. ; Keidar, Idit ; Dahlia, Malkhi ; Shraer, Alexander: Dynamic atomic storage without consensus. In: *Proc. of the 28th ACM symposium on Principles of distributed computing*, 2009, S. 17–25 (Zitiert auf Seite 11)

[4] Attiya, Hagit ; Bar-Noy, Amotz ; Dolev, Danny: Sharing memory robustly in message-passing systems. In: *Journal of the ACM* 42 (1995), Januar, Nr. 1, S. 124–142 (Zitiert auf Seite 11)

[5] Balakrishnan, Hari ; Kaashoek, M. F. ; Karger, David ; Morris, Robert ; Stoica, Ion: Looking up data in P2P systems. In: *Communications of the ACM* 46 (2003), Februar, Nr. 2, S. 43–48 (Zitiert auf Seite 6)

[6] Bartlang, Udo ; Müller, Jörg P.: DhtFlex: A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays. In: *Proc. of the 3rd ICIW08*, 2008, S. 377–384 (Zitiert auf Seiten 9, 13, 23 und 27)

[7] Bernstein, Philip A. ; Goodman, Nathan: An algorithm for concurrency control and recovery in replicated distributed databases. In: *ACM Trans. on Database Systems* 9 (1984), Dezember, Nr. 4, S. 596–615 (Zitiert auf Seiten 11 und 12)

[8] Bernstein, Philip A. ; Shipman, D. ; Rothnie, J. B.: Concurrency control in a system for distributed databases (SDD-1). In: *ACM Trans. on Database Systems* 5 (1980), März, Nr. 1, S. 1–17 (Zitiert auf Seite 12)

[9] Bhagwan, R. ; Tati, K. ; Cheng, Y.-C. ; Savage, S. ; Voelker, G. M.: Total Recall: System support for automated availability management. In: *Proc. of the 1st Symposium on Networked Systems Design and Implementation*, 2004, S. 25–39 (Zitiert auf Seiten 9, 13, 23, 25, 27 und 28)

[10] Bhagwan, Rianjita ; Moore, David ; Savage, Stefan ; Geo: Replication Strategies for Highly Available Peer-to-Peer Storage. In: *Proc. of Future directions in Distributed Computing*, 2002 (Zitiert auf Seite 8)

[11] Blake, Charles ; Rodrigues, Rodrigo: High availability, scalable storage, dynamic peer networks: Pick two. In: *9th Workshop on Hot Topics in Operation Systems (HotOS)*, 2003, S. 18–21 (Zitiert auf Seite 23)

[12] Blömer, Johannes ; Kalfane, Malik ; Karp, Richard ; Karpinski, Marek ; Luby, Michael ; Zuckerman, David: An XOR-Based Erasure-Resilent Coding Scheme / ICSI. 1995 (TR-95-048). – Forschungsbericht (Zitiert auf Seite 9)

[13] Bloom, Burton H.: Space/time tradeoffs in hash coding with allowable errors. In: *Communications of the ACM* 13 (1970), Juli, Nr. 7, S. 422–426 (Zitiert auf Seiten 16 und 22)

[14] Bonomi, Flavio ; Mitzenmacher, Michael ; Panigrah, Rina ; Singh, Sushil ; Varghese, George: Beyond bloom filters: from approximate membership checks to approximate state machines. In: *SIGCOMM Comput. Commun. Rev.* 36 (2006), August, S. 315–326. – ISSN 0146–4833 (Zitiert auf Seite 17)

[15] Brewer, Eric A.: Towards robust distributed systems. In: *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 2000, S. 7–10 (Zitiert auf Seite 6)

[16] Broder, Andrei ; Mitzenmacher, Michael ; Broder, Andrei: Network Applications of Bloom Filters: A Survey. In: *Internet Mathematics* Bd. 1, 2002, S. 636–646 (Zitiert auf Seite 16)

[17] Broder, Andrei Z. ; Charikar, Moses ; Frieze, Alan M. ; Mitzenmacher, Michael: Min-wise independent permutations. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1998 (STOC '98). – ISBN 0–89791–962–9, 327–336 (Zitiert auf Seite 84)

[18] Byers, John W. ; Considine, Jeffrey ; Mitzenmacher, Michael: Fast Approximate Reconciliation of Set Differences / Boston Univ. 2002 (BUCS-TR-2002-019). – Forschungsbericht. – `http://hdl.handle.net/2144/1469` (Zitiert auf Seiten 3, 20, 21, 22, 44, 66 und 77)

[19] Byers, John W. ; Considine, Jeffrey ; Mitzenmacher, Michael ; Rost, Stanislav: Informed content delivery across adaptive overlay networks. In: *IEEE/ACM Trans. Netw.* 12 (2004), October, S. 767–780. – ISSN 1063–6692 (Zitiert auf Seite 3)

[20] Cates, Josh: *Robust and efficient data management for a distributed hash table*, Massachusetts Institute of Technology, Diplomarbeit, 2003 (Zitiert auf Seiten 13, 20, 23, 27, 31, 40 und 60)

[21] Charles, Denis ; Chellapilla, Kumar: Bloomier Filters: A Second Look. In: *Proceedings of the 16th annual European symposium on Algorithms*. Berlin, Heidelberg : Springer-Verlag, 2008 (ESA '08). – ISBN 978–3–540–87743–1, S. 259–270 (Zitiert auf Seite 17)

[22] Chazelle, Bernard ; Kilian, Joe ; Rubinfeld, Ronitt ; Tal, Ayellet: The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In: *Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA, 2004 (SODA '04), S. 30–39 (Zitiert auf Seite 17)

[23] Cheung, Shun Y. ; Ammar, Mostafa H. ; Ahamad, Mustaque: The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. In: *IEEE Transactions on Knowledge and Data Engineering* 4 (1992), Dezember, Nr. 6, S. 582–592 (Zitiert auf Seite 12)

[24] Chun, Brent N. ; Vahdat, Amin: Workload and failure characterization on a large-scale federated testbed / Intel Research Technical Report. 2003 (IRB-TR-03-040). – Forschungsbericht. – `http://intel-research.net/Publications/Berkeley/111220031149_179.pdf` (Zitiert auf Seite 24)

[25] Chun, Byung-Gon ; Dabek, Frank ; Haeberlen, Andreas ; Sit, Emil ; Weatherspoon, Hakim ; Kaashoek, M. F. ; Kubiatowicz, John ; Morris, Robert: Efficient replica maintenance for distributed storage systems. In: *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*. Berkeley, CA, USA : USENIX Association, 2006 (NSDI'06), S. 4–18 (Zitiert auf Seiten 9, 25 und 27)

[26] Clarke, Ian ; Sandberg, Oskar ; Wiley, Brandon ; Hong, Theodore W.: Freenet: a distributed anonymous information storage and retrieval system. In: *International workshop on Designing privacy enhancing technologies: design issues in anonymity and unobservability*. New York, NY, USA : Springer-Verlag New York, Inc., 2001. – ISBN 3–540–41724–9, S. 46–66 (Zitiert auf Seite 6)

[27] Dabek, Frank ; Kaashoek, M. F. ; Karger, David ; Morris, Robert ; Stoica, Ion: Wide-area cooperative storage with CFS. In: *SIGOPS Oper. Syst. Rev.* 35 (2001), Dezember, Nr. 5, S. 202–215 (Zitiert auf Seite 9)

[28] DeCandia, Giuseppe ; Hastorun, Deniz ; Jampani, Madan ; Kakulapati, Gunavard ; Lakshman, Avinash ; Pilchin, Alex ; Sivasubramanian, Swaminathan ; Vosshall, Peter ; Vogels, Werner: Dynamo: Amazon's Highly Available Key-value Store. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ACM, 2007, S. 205–220 (Zitiert auf Seiten 2, 20 und 32)

[29] Demers, Alan ; Greene, Dan ; Hauser, Carl ; Irish, Wes ; Larson, John ; Shenker, Scott ; Sturgis, Howard ; Swinehart, Dan ; Terry, Doug: Epidemic Algorithms for replicated

Database Maintenance. In: *Proc. of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87)*, ACM, August 1987, S. 1–12 (Zitiert auf Seiten 43 und 44)

[30] Deutsch, P. ; Gailly, J-L.: *ZLIB Compressed Data Format Specification version 3.3.* RFC 1950. `http://www.ietf.org/rfc/rfc1950`. Version: 1996 (Zitiert auf Seite 37)

[31] Donnet, Benoit ; Baynat, Bruno ; Friedman, Timur: Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In: *Proceedings of the 2006 ACM CoNEXT conference.* New York, NY, USA : ACM, 2006 (CoNEXT '06). – ISBN 1–59593–456–1, S. 13:1–13:12 (Zitiert auf Seite 17)

[32] Druschel, Peter ; Rowstron, Antony: PAST: A large-scale, persistent peer-to-peer storage utility. In: *Proc. of 8th Workshop on HotOS*, 2001, S. 75–80 (Zitiert auf Seite 2)

[33] Eastlake, D. ; Jones, P.: *US Secure Hash Algorithm 1 (SHA1).* RFC 3174. `http://www.ietf.org/rfc/rfc3174`. Version: 2001 (Zitiert auf Seite 39)

[34] Eppstein, David ; Goodrich, Michael T. ; Uyeda, Frank ; Varghese, George: What's the difference?: efficient set reconciliation without prior context. In: *SIGCOMM Comput. Commun. Rev.* 41 (2011), August, Nr. 4, S. 218–229. – ISSN 0146–4833 (Zitiert auf Seiten 3 und 84)

[35] Fan, Li ; Cao, Pei ; Almeida, Jussara ; Broder, Andrei Z.: Summary cache: a scalable wide-area web cache sharing protocol. In: *IEEE/ACM Transactions on Networking* 8 (2000), Juni, Nr. 3, S. 281–293 (Zitiert auf Seite 17)

[36] Fetzer, Christof: Perfect Failure Detection in Timed Asynchronous Systems. In: *IEEE Transactions on Computers* 52 (2003), Februar, Nr. 2, S. 99–112. – ISSN 0018–9340 (Zitiert auf Seite 13)

[37] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1994. – ISBN 978–0201633610 (Zitiert auf Seite 35)

[38] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google file system. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), Oktober, Nr. 5, S. 29–43 (Zitiert auf Seite 9)

[39] Ghodsi, Ali: *Distributed k-ary System: Algorithms for Distributed Hash Tables*, KTH - Royal Institute of Technology, Diss., Oktober 2006 (Zitiert auf Seite 6)

[40] Ghodsi, Ali ; Alima, Luc O. ; Haridi, Seif: Symmetric replication for structured peer-to-peer systems. In: *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing.* Berlin, Heidelberg : Springer-Verlag, 2007 (DBISP2P'05/06). – ISBN 978–3–540–71660–0, S. 74–85 (Zitiert auf Seite 9)

[41] Gilbert, Seth ; Lynch, Nancy ; Shvartsman, Alex: RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In: *Proceedings of the International Conference on Dependable Systems and Networks*, 2003, S. 259–268 (Zitiert auf Seite 27)

[42] Goodrich, Michael T. ; Mitzenmacher, Michael: Invertible Bloom Lookup Tables. In: *ArXiv e-prints* abs/1101.2245. (2011) (Zitiert auf Seite 17)

[43] Gray, Jim ; Lamport, Leslie: Consensus on transaction commit. In: *ACM Trans. on Database Systems* 31 (2006), Nr. 1, S. 133–160 (Zitiert auf Seite 14)

[44] Haeberlen, Andreas ; Mislove, Alan ; Druschel, Peter: Glacier: Highly durable, decentralized storage despite massive correlated failures. In: *Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005, S. 143.158 (Zitiert auf Seiten 14 und 26)

[45] Heinz, Steffen ; Zobel, Justin ; Williams, Hugh E.: Burst Tries: A Fast, Efficient Data Structure for String Keys. In: *ACM Trans. on Information Systems* 20 (2002), April, Nr. 2, S. 192–223 (Zitiert auf Seite 20)

[46] Jiang, Jehn-Ruey: The Column Protocol: A high availability and low message cost solution for managing replicated data. In: *Information Systems* 20 (1995), Nr. 8, S. 687–696 (Zitiert auf Seite 12)

[47] Jiang, Jehn-Ruey ; King, Chung-Ta ; Liao, Chi-Hsiang: MUREX: A mutable replica control scheme for structured peer-to-peer storage systems. In: *Lecture Notes in Computer Science* 3947 (2006), S. 93–102 (Zitiert auf Seiten 13, 23 und 26)

[48] Karp, R. ; Schindelhauer, C. ; Shenker, S. ; Vöcking, B.: Randomized rumor spreading. In: *In IEEE Symposium on Foundations of Computer Science*, 2000, S. 565–574 (Zitiert auf Seite 44)

[49] Kirk, Patrick: *Gnutella Protocol Specification v0.4*, 2003. – `http://rfc-gnutella.sourceforge.net/developer/stable/index.html`, besucht am 16.05.2012 (Zitiert auf Seite 6)

[50] Kirsch, Adam ; Mitzenmacher, Michael: Less hashing, same performance: building a better bloom filter. In: *Proceedings of the 14th conference on Annual European Symposium - Volume 14*. London, UK : Springer-Verlag, 2006. – ISBN 3–540–38875–3, S. 456–467 (Zitiert auf Seiten 16 und 37)

[51] Kubiatowicz, John ; Bindel, David ; Chen, Yan ; Czerwinski, Steven ; Eaton, Patrick ; Geels, Dennis ; Gummadi, Ramakrishna ; Rhea, Sean ; Weatherspoon, Hakim ; Weimer, Westley ; Wells, Chris ; Zhao, Ben: Oceanstore: an architecture for global-scale persistent

storage. In: *ACM SIGARCH Computer Architecture News* 28 (2000), Dezember, Nr. 5, S. 190–201 (Zitiert auf Seiten 2 und 13)

[52] Kumar, Akhil: Hierarchical quorum consensus: a new algorithm for managing replicated data. In: *IEEE Transactions on Computers* 40 (1991), September, Nr. 9, S. 996–1004 (Zitiert auf Seiten 11 und 12)

[53] Lamport, Leslie: Time, clocks, and the ordering of events in a distributed system. In: *Communications of the ACM* 21 (1978), Juli, Nr. 7, S. 558–565 (Zitiert auf Seite 4)

[54] Lamport, Leslie: Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. In: *ACM Trans. Program. Lang. Syst.* 6 (1984), April, S. 254–280. – ISSN 0164–0925 (Zitiert auf Seite 11)

[55] Lamport, Leslie: The part-time parliament. In: *ACM Trans. on Computer Systems* 16 (1998), Mai, Nr. 2, S. 133–169 (Zitiert auf Seite 14)

[56] Lin, W. K. ; Chiu, D. M. ; Lee, Y. B.: Erasure Code Replication Revisited. In: *Proc. of the 4th international conf. on Peer-to-Peer Computing*. Washington, DC, USA : IEEE Computer Society, August 2004 (P2P '04), S. 90–97 (Zitiert auf Seiten 8 und 9)

[57] Luby, Michael: *Benchmark comparisons of erasure codes*. `http://www.icsi.berkeley.edu/~luby/erasure.html`. – besucht am 16.05.2012 (Zitiert auf Seite 9)

[58] Luby, Michael ; Mitzenmacher, Michael ; Shokrollahi, M. A.: Analysis of random processes via And-Or tree evaluation. In: *Proc. of the 9th ACM-SIAM symposium on Discrete algorithms*, 1998, S. 364–373 (Zitiert auf Seite 9)

[59] Luby, Michael ; Mitzenmacher, Michael ; Shokrollahi, M. A. ; Spielman, Daniel A. ; Stemann, Volker: Practical loss-resilient codes. In: *Proc. of 29th ACM Symposium on Theory of computing*, 1997, S. 150–159 (Zitiert auf Seite 9)

[60] Lynch, Nancy ; Gilbert, Seth: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *ACM SIGACT News* 33 (2002), Juni, Nr. 2, S. 51–59 (Zitiert auf Seiten 6 und 7)

[61] Lynch, Nancy ; Shvartsman, Alex: RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In: *Proceedings of the 16th International Conference on Distributed Computing*, 2002, S. 173–190 (Zitiert auf Seite 27)

[62] Merkle, Ralph C.: A certified digital signature. In: *CRYPTO '89*, Springer-Verlag, 1990 (Lecture Notes in Computer Science Volume 435), S. 218–238 (Zitiert auf Seiten 19 und 22)

[63] Mills, David L.: Internet time synchronization: the network time protocol. In: *IEEE Transactions on Communications* 39 (1991), Oktober, Nr. 19, S. 1482–1493 (Zitiert auf Seite 4)

[64] Minsky, Yaron ; Trachtenberg, Ari: Efficient Reconciliation of Unordered Databases / Cornell University Ithaca. 1999. – Forschungsbericht (Zitiert auf Seite 15)

[65] Minsky, Yaron ; Trachtenberg, Ari: Practical Set Reconciliation / Boston University. 2002 (BU-ECE-2002-01). – Forschungsbericht. – http://cis.poly.edu/westlab/papers/ref/practical.pdf (Zitiert auf Seite 19)

[66] Minsky, Yaron ; Trachtenberg, Ari ; Zippel, Richard: Set Reconciliation with Nearly Optimal Communication Complexity. In: *IEEE Transactions on Information Theory* 49 (2003), September, Nr. 9, S. 2213–2218 (Zitiert auf Seiten 3, 15, 17, 18 und 22)

[67] Mitzenmacher, Michael: Compressed bloom filters. In: *IEEE/ACM Transactions on Networking* 10 (2002), Nr. 5, S. 604–612. – ISSN 1063–6692 (Zitiert auf Seiten 16 und 17)

[68] Moser, Monika ; Haridi, Seif: Atomic Commitment in transactional DHTs. In: *Proc. of the CoreGRID Symposium*, 2007 (Zitiert auf Seite 15)

[69] Murray, James D.: *Mathematical Biology.* Springer, 1993. – ISBN 978–3–540–57204–X (Zitiert auf Seite 43)

[70] Muthitacharoen, Athicha ; Gilbert, Seth ; Morris, Robert: Etna: a Fault-tolerant Algorithm for Atomic Mutable DHT Data / MIT. 2005 (MIT-CSAIL-TR-2005-044). – Forschungsbericht. – http://hdl.handle.net/1721.1/30555 (Zitiert auf Seiten 13, 23 und 27)

[71] Nath, Suman ; Yu, Haifeng ; Gibbons, Phillip B. ; Seshan, Srinivasan: Tolerating Correlated Failures in Wide-Area Monitoring Services / Intel Research. 2004 (IPR-TR-04-09). – Forschungsbericht. – http://www.intel-research.net/Publications/Pittsburgh/101220041252_263.pdf (Zitiert auf Seiten 27, 30 und 31)

[72] Oram, Andy (Hrsg.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2001. – ISBN 978–059600110–0 (Zitiert auf Seite 6)

[73] Patterson, David A. ; Gibson, Garth ; Katz, Randy H.: A case for redundant arrays of inexpensive disks (RAID). In: *Proc. of the 1988 ACM SIGMOD international conference on Management of data.* New York, NY, USA : ACM, 1988 (SIGMOD '88). – ISBN 0–89791–268–3, S. 109–116 (Zitiert auf Seite 9)

[74] Pittel, Boris: On spreading a rumor. In: *SIAM Journal on Applied Mathematics* 47 (1987), S. 213–223 (Zitiert auf Seiten 31 und 43)

[75] Renesse, Robbert van ; Schneider, Fred B.: Chain replication for supporting high throughput and availability. In: *Proc. of the 6th conference on Symposium on Operation Systems Design & Implementation*, 2004, S. 7–21 (Zitiert auf Seite 11)

[76] Rhea, Sean ; Geels, Dennis ; Roscoe, Timothy ; Kubiatowicz, John: Handling churn in a DHT. In: *Proceedings of the USENIX Annual Technical Conference*, 2004, S. 127–140 (Zitiert auf Seite 13)

[77] Rhea, Sean C.: *OpenDHT: A Public DHT Service.* Berkeley, CA, USA, University of California at Berkeley, Diss., 2005 (Zitiert auf Seiten 20, 30 und 31)

[78] Rivest, Ronald: *The MD5 Message-Digest Algorithm.* RFC 1321. `http://www.ietf.org/rfc/rfc1321`. Version: April 1992 (Zitiert auf Seite 37)

[79] Rodrigues, Rodrigo ; Liskov, Barbara: High availability in DHTs: Erasure coding vs. replication. In: *Lecture Notes in Computer Science* 3640 (2005), S. 226–239 (Zitiert auf Seite 9)

[80] Rowstron, Antony I. T. ; Druschel, Peter: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: *Middleware 2001* (2001), S. 329–350 (Zitiert auf Seite 6)

[81] Schintke, Florian ; Reinefeld, Alexander ; Haridi, Seif ; Schütt, Thorsten: Enhanced Paxos Commit for Transactions on DHTs. In: *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, IEEE Computer Society, Mai 2010. – ISBN 978–0–7695–4039–9, S. 448–454 (Zitiert auf Seiten 14 und 15)

[82] Schütt, Thorsten ; Schintke, Florian ; Reinefeld, Alexander: Structured Overlay without Consistent Hashing: Empirical Results. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops*, 2006, S. 8–16 (Zitiert auf Seiten 6 und 46)

[83] Schütt, Thorsten ; Schintke, Florian ; Reinefeld, Alexander: Scalaris: reliable transactional p2p key/value store. In: *ERLANG '08: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG.* New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–065–4, S. 41–48 (Zitiert auf Seiten 2 und 14)

[84] Shafaat, Tallat M. ; Moser, Monika ; Ghodsi, Ali ; Schütt, Thorsten ; Haridi, Seif ; Reinefeld, Alexander: On Consistency of Data in Structured Overlay Networks. In: *Proceedings of the 3rd CoreGRID Integration Workshop*, 2008 (Zitiert auf Seiten 13, 25 und 26)

[85] Shafaat, Tallat M. ; Moser, Monika ; Schütt, Thorsten ; Reinefeld, Alexander ; Ghodsi, Ali ; Haridi, Seif: Key-based consistency and availability in structured overlay networks. In: *Proceedings of the 3rd international conference on Scalable information systems.* ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (InfoScale '08). – ISBN 978–963–9799–28–8, 13:1–13:5 (Zitiert auf Seite 26)

[86] Sit, Emil ; Haeberlen, Andreas ; Dabek, Frank ; Chun, Byung-Gon ; Weatherspoon, Hakim ; Morris, Robert ; Kaashoek, M. F. ; Kubiatowicz, John: Proactive replication for data durability. In: *Proc. of IPTPS*, 2006 (Zitiert auf Seiten 14 und 26)

[87] Skjegstad, Magnus ; Maseng, Torleiv: Low complexity set reconciliation using Bloom filters. In: *Proceedings of the 7th ACM ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing*. New York, NY, USA : ACM, 2011 (FOMC '11). – ISBN 978–1–4503–0779–6, S. 33–41 (Zitiert auf Seite 3)

[88] Starobinski, David ; Trachtenberg, Ari ; Agarwal, Sachin: Efficient PDA Synchronization. In: *IEEE Transactions on Mobile Computing* 2 (2003), January, Nr. 1, S. 40–51. – ISSN 1536–1233 (Zitiert auf Seite 3)

[89] Stoica, Ion ; Morris, Robert ; Liben-Nowell, D. ; Karger, D. R. ; Kaashoek, M. F. ; Dabek, Frank ; Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM Comput. Commun. Rev.* 31 (2001), August, S. 149–160. – ISSN 0146–4833 (Zitiert auf Seite 6)

[90] Tarkoma, Sasu ; Rothenberg, Christian E. ; Lagerspetz, Eemil: Theory and Practice of Bloom Filters for Distributed Systems. In: *IEEE Communications Surveys & Tutorials* 14 (2012), April, Nr. 1, S. 131–155 (Zitiert auf Seite 17)

[91] Tati, Kiran ; Voelker, Geoffrey M.: On object maintenance in peer-to-peer systems. In: *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006 (Zitiert auf Seiten 24 und 25)

[92] Tharakan, Royans: *CAP-Theorem*. `www.royans.net/arch/brewers-cap-theorem-on-distributed-systems/`. – besucht am 16.05.2012 (Zitiert auf Seite 7)

[93] Thomas, Robert H.: A majority consensus approach to concurrency control for multiple copy databases. In: *ACM Trans. on Database Systems* 4 (1979), Nr. 2, S. 180–207 (Zitiert auf Seite 12)

[94] Trutna, Jesse: Nye's Trie and Floret Estimators: Techniques for Detecting and Repairing Divergence in the SCADS Distributed Storage Toolkit / UC Berkeley. 2010 (EECS-2010-30). – Forschungsbericht. – `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-30.pdf` (Zitiert auf Seiten 20 und 39)

[95] van Renesse, Robbert ; Dumitriu, Dan ; Gough, Valient ; Thomas, Chris: Efficient reconciliation and flow control for anti-entropy protocols. In: *Proc. of 2nd Workshop on Large-Scale Distributed Systems and Middleware*, ACM, September 2008, S. 1–7 (Zitiert auf Seiten 30 und 31)

[96] Weatherspoon, Hakim: *Design and Evaluation of Distributed Wide-Area On-line Archival Storage Systems*, University of California at Berkeley, Diss., 2006 (Zitiert auf Seite 25)

[97] Weatherspoon, Hakim ; Kubiatowicz, John: Erasure Coding vs. Replication: A Quantitative Comparison. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002, S. 328–338 (Zitiert auf Seite 9)

[98] Wu, Di ; Tian, Ye ; Ng, Kam-Wing ; Datta, Anwitaman: Stochastic analysis of the interplay between object maintenance and churn. In: *Comput. Commun.* 31 (2008), February, S. 220–239. – ISSN 0140–3664 (Zitiert auf Seite 13)

[99] Yu, Haifeng: Overcoming the Majority Barrier in Large-Scale Systems. In: *DISC: Proc. of the 17th International Symposium on Distributed Computing*, 2003, S. 352–366 (Zitiert auf Seite 12)

[100] Yu, Haifeng: Signed quorum systems. In: *Distributed Computing* 18 (2006), März, Nr. 4, S. 307–323. – ISSN 0178–2770 (Zitiert auf Seite 12)

[101] Yu, Haifeng ; Vahdat, Amin: Consistent and Automatic Replica Regeneration. In: *ACM Trans. on Storage* 1 (2005), Februar, Nr. 1, S. 3–37 (Zitiert auf Seiten 23, 27 und 28)

[102] Zhao, Jing ; Yu, Hongliang ; Zhang, Kun ; Zheng, Weimin ; Wu, Jie ; Hu, Jinfeng: Achieving Reliability through Replication in a Wide-Area Network DHT Storage System. In: *ICPP '07: Proc. of the 2007 International Conference on Parallel Processing*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2933–X, S. 29–37 (Zitiert auf Seiten 13 und 26)

A ART Parameter Messwerte

Die dargestellte Tabelle zeigt die Auswirkung der Kombination der vier ART Parameter auf drei Metriken für die Fehlertypen *upd* und *reg*. Die Metriken sind Genauigkeit (G), Übertragungskosten (B) und Redundanz (Red). Die Anzahl der übertragenen Schlüssel-Wert-Versions Tripel (KVV) ist zusätzlich angegeben. Die Werte wurden bei der Synchronisation zweier Knoten bei gleichmäßiger Fehler- und Datenverteilung gemessen, mit $N = 10000$, $|\Delta| = 0{,}03N$ und *correction factor* $= 3$. Die Messwerte entsprechen dem Mittel über 10 Ausführungen.

Die Tabelle zeigt für das kartesische Produkt der Parametermengen $v = \{2;\ 4;\ 8;\ 16;\ 32;\ 64\}, b = \{1;\ 2;\ 4;\ 8\}, \mathrm{Fpr}_I = \{0{,}001;\ 0{,}01;\ 0{,}1;\ 0{,}4;\ 0{,}8\}$ und $\mathrm{Fpr}_B = \{0{,}01;\ 0{,}1;\ 0{,}2\}$ die Auswirkung auf die verschiedenen Metriken. Es sind drei Cluster anhand der Redundanz und der *reg* Genauigkeit erkennbar, deren Parameterkonstellationen ähnliche Werte in den Metriken aufweisen.

Tabelle A.1: Art Parameter Test

		Bloom Fpr		**Update**				**Reg**			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
					Cluster C_B						
2	8	0,001	0,01	95,9667%	31836	8,7162	2509,4	96,3000%	34199,6	9,5424	2756,8
2	8	0,01	0,01	96,2000%	30848	8,7775	2533,2	95,9333%	32406,7	9,3753	2698,2
2	8	0,1	0,01	95,4000%	29298	8,7379	2500,8	95,6000%	31574,1	9,5370	2735,2
2	8	0,4	0,01	94,9667%	28633	8,8034	2508,1	95,6000%	30662,8	9,4718	2716,5
4	8	0,1	0,01	92,9667%	18808	4,4729	1247,5	94,0667%	23824,9	6,2527	1764,5
4	8	0,8	0,01	92,5333%	17988	4,4643	1239,3	93,7667%	23406,1	6,3800	1794,7
16	4	0,4	0,01	93,8667%	18174	4,4705	1258,9	93,6667%	15353	3,4762	976,8
16	4	0,1	0,01	93,0000%	18142	4,4656	1245,9	93,6000%	15350	3,4427	966,7
4	4	0,001	0,01	93,4333%	20482	4,4627	1250,9	93,5667%	17531	3,4051	955,8
16	8	0,4	0,01	92,8333%	17948	4,4391	1236,3	93,5000%	15408	3,5020	982,3
2	4	0,01	0,01	92,2333%	23063	4,7741	1321	93,4667%	31209,9	7,7297	2167,4
64	8	0,4	0,01	93,5333%	18103	4,4822	1257,7	93,4333%	15099	3,4153	957,3
16	8	0,8	0,01	92,7667%	17903	4,4452	1237,1	93,4000%	15271	3,4757	973,9
16	4	0,8	0,01	93,0333%	18023	4,4755	1249,1	93,3333%	15157	3,4375	962,5
16	8	0,1	0,01	92,9667%	18081	4,4453	1239,8	93,3000%	15439	3,4855	975,6
16	8	0,001	0,01	94,3333%	18662	4,4707	1265,2	93,3000%	15690	3,4584	968
64	8	0,1	0,01	93,7333%	18128	4,4723	1257,6	93,2333%	15179	3,4419	962,7
8	4	0,1	0,01	93,7000%	18341	4,4365	1247,1	93,2333%	15489	3,4390	961,9
4	4	0,1	0,01	93,0000%	18734	4,4434	1239,7	93,2000%	16012	3,4603	967,5

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
8	4	0,8	0,01	93,8667%	17999	4,4197	1244,6	93,1667%	15306	3,4894	975,3
8	4	0,001	0,01	92,3667%	18965	4,4728	1239,4	93,1667%	16280	3,4737	970,9
4	8	0,01	0,01	93,5667%	19727	4,4806	1257,7	93,1333%	25163,1	6,5082	1818,4
64	8	0,01	0,01	92,9000%	18087	4,4837	1249,6	93,1333%	15309	3,4782	971,8
8	8	0,4	0,01	93,1000%	18188	4,4859	1252,9	93,0333%	15302	3,4550	964,3
4	8	0,4	0,01	93,3000%	18359	4,4723	1251,8	93,0000%	23875,5	6,5158	1817,9
16	4	0,01	0,01	93,2000%	18347	4,4707	1250	92,9667%	15575	3,4880	972,8
8	8	0,001	0,01	93,4667%	19104	4,4697	1253,3	92,9333%	16118	3,4243	954,7
64	4	0,8	0,01	92,8000%	17893	4,4465	1237,9	92,8000%	15194	3,4770	968
16	4	0,001	0,01	93,7333%	18647	4,4940	1263,7	92,8000%	15646	3,4612	963,6
4	8	0,001	0,01	93,5000%	20582	4,4973	1261,5	92,7000%	25957	6,5430	1819,6
64	4	0,001	0,01	92,8333%	18090	4,4740	1246	92,7000%	15394	3,5110	976,4
8	8	0,1	0,01	93,1000%	18358	4,4712	1248,8	92,6667%	15599	3,4996	972,9
8	4	0,01	0,01	93,3667%	18695	4,4538	1247,5	92,6667%	15896	3,4806	967,6
64	4	0,1	0,01	93,1000%	17795	4,3835	1224,3	92,6667%	15219	3,4773	966,7
64	4	0,01	0,01	92,7667%	17978	4,4510	1238,7	92,5667%	15194	3,4580	960,3
2	4	0,1	0,01	93,2333%	20968	4,7998	1342,5	92,5000%	28892,2	7,7813	2159,3
64	8	0,8	0,01	93,5000%	17997	4,4503	1248,3	92,5000%	15104	3,4559	959
64	4	0,4	0,01	93,0667%	17940	4,4463	1241,4	92,3000%	15163	3,4803	963,7
2	4	0,4	0,01	91,3333%	19437	4,8482	1328,4	92,2333%	27466	7,7705	2150,1
16	8	0,01	0,01	93,0667%	18138	4,4022	1229,1	92,2333%	15438	3,4662	959,1
4	4	0,4	0,01	92,8667%	18282	4,4641	1243,7	92,2333%	15293	3,4145	944,8
2	4	0,001	0,01	93,8000%	25719	4,8173	1355,6	92,2000%	33255,6	7,7628	2147,2
4	4	0,01	0,01	93,0667%	19624	4,4660	1246,9	92,1333%	16738	3,4671	958,3
8	4	0,4	0,01	93,4667%	18114	4,4419	1245,5	92,1333%	15210	3,4555	955,1
8	8	0,01	0,01	93,0667%	18626	4,4434	1240,6	92,0333%	15856	3,4900	963,6
64	8	0,001	0,01	92,7667%	17983	4,4387	1235,3	91,9000%	15170	3,4603	954
8	8	0,8	0,01	92,5333%	18005	4,4856	1245,2	91,7000%	15097	3,4693	954,4
4	4	0,8	0,01	92,1000%	17979	4,4806	1238	91,0667%	15193	3,5117	959,4
2	2	0,1	0,01	88,7333%	19445	3,0887	822,2	90,6000%	20522,8	3,4750	944,5
2	8	0,1	0,1	87,0667%	25925	8,7523	2286,1	90,2000%	28688,5	9,4863	2567
2	2	0,01	0,01	88,9333%	23113	3,1308	835,3	90,1667%	23967,2	3,4769	940,5
2	2	0,001	0,01	89,3000%	26533	3,0743	823,6	89,5667%	27344,9	3,4592	929,5
64	2	0,8	0,2	88,5667%	56477	2,3786	632	89,4667%	57332,7	3,4683	930,9
64	2	0,01	0,2	89,1333%	58744	2,4121	645	89,1000%	59374,1	3,4979	935
64	2	0,001	0,1	88,1333%	81018	2,3869	631,1	89,0667%	80709	3,4491	921,6
64	2	0,001	0,2	89,3000%	59774	2,3733	635,8	88,8333%	60444,4	3,4841	928,5
64	2	0,1	0,01	88,7000%	149590	2,4051	640	88,8333%	146512,1	3,4507	919,6
8	2	0,4	0,01	89,1000%	27665	2,3954	640,3	88,8000%	29839	3,4741	925,5
64	2	0,4	0,01	89,7000%	148983	2,4039	646,9	88,7667%	145766,5	3,5047	933,3
16	2	0,1	0,1	88,8000%	26875	2,3671	630,6	88,6333%	29104,7	3,5085	932,9
2	2	0,4	0,01	64,3000%	15082	3,1042	598,8	88,6333%	18262,8	3,4915	928,4
16	2	0,001	0,2	88,0333%	23770	2,4002	633,9	88,6333%	25957,3	3,4520	917,9

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
4	2	0,4	0,01	89,2000%	19141	2,3778	636,3	88,6000%	21611	3,4391	914,1
16	2	0,4	0,1	89,6667%	26287	2,4048	646,9	88,5667%	28302	3,4810	924,9
16	2	0,001	0,01	88,7667%	48086	2,3710	631,4	88,5667%	49415,5	3,4554	918,1
64	2	0,1	0,1	89,0000%	78910	2,4150	644,8	88,5333%	78642,6	3,4703	921,7
16	2	0,01	0,1	89,1000%	28168	2,3764	635,2	88,5000%	30198,4	3,4621	919,2
2	8	0,01	0,1	87,5000%	27357	8,7874	2306,7	88,4333%	29729,8	9,6155	2551
16	2	0,4	0,2	89,2000%	20606	2,3991	642	88,4333%	22879,2	3,4742	921,7
4	2	0,001	0,01	88,5667%	24066	2,3861	634	88,4000%	26628,1	3,5256	935
64	2	0,8	0,1	89,3000%	77755	2,3542	630,7	88,4000%	77657,4	3,4872	924,8
64	2	0,001	0,01	89,9000%	151868	2,3852	643,3	88,3000%	148298,8	3,5315	935,5
16	2	0,8	0,2	88,0000%	20128	2,3932	631,8	88,1667%	22515,1	3,4836	921,4
16	2	0,001	0,1	88,2667%	29379	2,3852	631,6	88,1333%	31418,9	3,4898	922,7
16	2	0,1	0,2	89,4000%	21345	2,3893	640,8	88,0667%	23718,5	3,5269	931,8
16	2	0,4	0,01	89,8667%	44912	2,3683	638,5	88,0000%	46282,9	3,4841	919,8
2	8	0,4	0,1	86,2000%	24895	8,7274	2256,9	87,9667%	27140,1	9,4153	2484,7
64	2	0,4	0,2	87,7667%	56772	2,3840	627,7	87,9333%	57534,7	3,4909	920,9
8	2	0,01	0,01	89,5333%	29901	2,3608	634,1	87,9333%	31963,1	3,4685	915
64	2	0,01	0,01	88,0333%	150633	2,3930	632	87,8333%	147314,3	3,5051	923,6
4	2	0,01	0,01	89,2000%	22114	2,3505	629	87,8333%	24575	3,4630	912,5
4	2	0,1	0,01	88,8000%	20272	2,3833	634,9	87,7000%	22757,9	3,4884	917,8
64	2	0,1	0,2	89,8000%	57656	2,4072	648,5	87,6000%	58180,3	3,4935	918,1
16	2	0,1	0,01	89,4333%	45626	2,3660	634,8	87,5667%	46922,1	3,4682	911,1
8	2	0,8	0,01	89,8333%	27263	2,3870	643,3	87,5667%	29262	3,4568	908,1
16	2	0,8	0,1	89,5000%	25818	2,3747	637,6	87,5333%	27844,1	3,4474	905,3
16	2	0,01	0,01	88,4667%	46765	2,3512	624	87,5000%	48148,2	3,4648	909,5
64	2	0,8	0,01	89,0667%	148588	2,3997	641,2	87,4667%	145434,1	3,5217	924,1
8	2	0,1	0,01	88,8667%	28480	2,3833	635,4	87,4333%	30661	3,5254	924,7
16	2	0,8	0,01	89,4333%	44593	2,4010	644,2	87,3000%	45778,6	3,4318	898,8
8	2	0,001	0,01	88,5667%	31335	2,3865	634,1	87,1000%	33188,2	3,4447	900,1
2	8	0,001	0,1	87,4000%	28440	8,7429	2292,4	86,8000%	30475,2	9,6240	2506,1
8	2	0,01	0,1	88,3000%	19799	2,3703	627,9	86,8000%	22179,4	3,4950	910,1
8	2	0,4	0,1	87,8667%	17506	2,3839	628,4	86,7333%	19924,6	3,4723	903,5
8	2	0,1	0,1	89,1000%	18372	2,3517	628,6	86,7333%	20687,8	3,4473	897
8	2	0,8	0,1	89,1333%	17193	2,3945	640,3	86,7000%	19521,5	3,4852	906,5
64	2	0,01	0,1	88,8667%	80016	2,4122	643,1	86,6667%	79571,8	3,5023	910,6
16	8	0,4	0,1	85,3333%	14540	4,4566	1140,9	86,6333%	12069	3,4390	893,8
16	2	0,01	0,2	87,9333%	22502	2,3950	631,8	86,5667%	24662,9	3,4910	906,6
64	2	0,4	0,1	89,6000%	78192	2,3832	640,6	86,5000%	77963	3,5233	914,3
64	4	0,4	0,1	85,3333%	14547	4,4824	1147,5	86,4000%	11984	3,4383	891,2
16	4	0,1	0,1	84,3667%	14579	4,4844	1135	86,3000%	12152	3,4465	892,3
16	8	0,8	0,1	85,4333%	14570	4,4838	1149,2	86,2667%	12174	3,5147	909,6
8	2	0,4	0,2	87,6333%	14449	2,3769	624,9	86,2333%	16871,3	3,4523	893,1
4	8	0,4	0,1	85,4333%	14809	4,4557	1142	86,1667%	20352	6,5915	1703,9

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
8	2	0,01	0,2	87,2333%	16761	2,3932	626,3	86,1667%	19144,6	3,4750	898,3
4	8	0,8	0,1	83,1333%	14378	4,5048	1123,5	86,0667%	19579,3	6,3919	1650,4
64	8	0,01	0,1	85,3333%	14565	4,4641	1142,8	86,0000%	12073	3,4636	893,6
8	2	0,001	0,1	88,1333%	21215	2,3680	626,1	85,9667%	23489	3,4905	900,2
64	8	0,001	0,1	84,5667%	14508	4,4667	1133,2	85,9667%	12109	3,4637	893,3
64	8	0,8	0,1	84,9667%	14434	4,4621	1137,4	85,9333%	12010	3,4717	895
16	4	0,8	0,1	86,9000%	14628	4,4304	1155	85,9333%	11988	3,4562	891
2	4	0,1	0,1	84,5333%	17337	4,7733	1210,5	85,8667%	25055,2	7,7570	1998,2
8	8	0,8	0,1	86,1333%	14660	4,4741	1156,1	85,8333%	11951	3,4377	885,2
16	4	0,4	0,1	84,7000%	14537	4,4888	1140,6	85,7667%	12122	3,4944	899,1
8	8	0,1	0,1	85,9000%	14922	4,4649	1150,6	85,6000%	12261	3,4443	884,5
4	4	0,001	0,1	85,7000%	17099	4,5041	1158	85,5333%	14492	3,4969	897,3
4	4	0,1	0,1	85,3333%	15289	4,4555	1140,6	85,5000%	12775	3,4667	889,2
8	4	0,001	0,1	86,8000%	15755	4,4693	1163,8	85,4333%	13096	3,5033	897,9
8	4	0,01	0,1	85,1333%	15224	4,4863	1145,8	85,4333%	12612	3,4514	884,6
8	2	0,8	0,2	86,6000%	14057	2,4207	628,9	85,4000%	16536,3	3,5203	901,9
64	4	0,8	0,1	85,7667%	14484	4,4400	1142,4	85,3667%	11943	3,4686	888,3
8	8	0,001	0,1	85,1000%	15594	4,4955	1147,7	85,2333%	12920	3,4427	880,3
8	2	0,001	0,2	86,4333%	18179	2,4092	624,7	85,2000%	20533,4	3,5192	899,5
16	8	0,01	0,1	85,7667%	14914	4,4777	1152,1	85,2000%	12237	3,4601	884,4
64	4	0,1	0,1	85,5333%	14556	4,4653	1145,8	85,2000%	11889	3,4394	879,1
4	8	0,01	0,1	84,8333%	16027	4,4515	1132,9	85,1333%	21307,4	6,5478	1672,3
16	8	0,001	0,1	86,1667%	15080	4,4580	1152,4	85,1333%	12494	3,4996	893,8
2	4	0,01	0,1	85,8000%	19921	4,8092	1237,9	85,1000%	27165,8	7,7759	1985,2
64	8	0,1	0,1	84,7333%	14467	4,4725	1136,9	85,0333%	12078	3,5202	898
8	8	0,01	0,1	85,3667%	15169	4,4526	1140,3	85,0333%	12456	3,4065	869
2	4	0,001	0,1	85,1333%	22075	4,7858	1222,3	84,9667%	29254,8	7,7352	1971,7
4	8	0,1	0,1	85,2667%	15350	4,4836	1146,9	84,9667%	20131,6	6,4104	1634
4	2	0,8	0,01	83,6667%	18258	2,4112	605,2	84,9667%	20833,3	3,5029	892,9
64	4	0,01	0,1	84,1000%	14339	4,4400	1120,2	84,9333%	11963	3,4639	882,6
4	8	0,001	0,1	86,0000%	17077	4,4814	1156,2	84,9000%	21902,4	6,4884	1652,6
8	4	0,8	0,1	84,9333%	14575	4,5039	1147,6	84,7667%	11946	3,4790	884,7
4	4	0,01	0,1	85,9667%	16089	4,4157	1138,8	84,7667%	13520	3,4680	881,9
16	4	0,01	0,1	86,2667%	14942	4,4625	1154,9	84,7333%	12267	3,4910	887,4
4	4	0,4	0,1	84,6000%	14737	4,4704	1134,6	84,7333%	12245	3,4831	885,4
4	4	0,8	0,1	83,5333%	14345	4,4693	1120	84,7333%	11970	3,4717	882,5
64	8	0,4	0,1	85,7000%	14479	4,4368	1140,7	84,6667%	11991	3,5114	891,9
8	2	0,1	0,2	86,3333%	15227	2,3795	616,3	84,5667%	17711,3	3,5235	893,9
64	4	0,001	0,1	85,6667%	14671	4,4728	1149,5	84,5667%	12076	3,5081	890
8	8	0,4	0,1	85,7667%	14655	4,4501	1145	84,5667%	11994	3,4643	878,9
8	4	0,4	0,1	85,5333%	14637	4,4552	1143,2	84,2000%	11959	3,4656	875,4
4	2	0,001	0,1	85,4000%	18085	2,3669	606,4	84,0000%	20577,6	3,5115	884,9
4	2	0,1	0,1	85,5333%	14285	2,3644	606,7	83,9667%	16835,3	3,4986	881,3

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
16	8	0,1	0,1	87,0000%	14896	4,4701	1166,7	83,9333%	12001	3,4837	877,2
8	4	0,1	0,1	84,3000%	14684	4,4555	1126,8	83,8000%	12297	3,5326	888,1
2	4	0,4	0,1	85,2000%	16212	4,8396	1237	83,4000%	23031,5	7,7138	1930
4	2	0,01	0,1	84,7667%	16313	2,4357	619,4	83,2667%	18498,3	3,4516	862,2
16	4	0,001	0,1	84,4333%	14938	4,4935	1138,2	83,2000%	12193	3,4603	863,7
4	2	0,8	0,1	80,4667%	12262	2,3865	576,1	82,5667%	14844,8	3,4275	849
4	2	0,4	0,1	86,9000%	13236	2,3640	616,3	81,8333%	15663,5	3,5695	876,3
2	2	0,1	0,1	82,7667%	15369	3,0942	768,3	81,5333%	16106,7	3,4828	851,9
2	2	0,001	0,1	82,1667%	22356	3,0815	759,6	81,5000%	22967,3	3,4352	839,9
2	2	0,01	0,1	84,0667%	18925	3,0539	770,2	81,4333%	19647,8	3,5113	857,8
2	8	0,1	0,2	77,9333%	23165	8,7553	2047	79,0667%	25222,6	9,5139	2256,7
2	8	0,001	0,2	78,8667%	25746	8,7063	2059,9	79,0667%	27233,8	9,3457	2216,8
16	4	0,1	0,2	75,4667%	12577	4,4549	1008,6	78,6333%	10623	3,4472	813,2
2	2	0,4	0,1	61,6667%	11287	3,0973	573	78,4000%	13737,7	3,5072	824,9
16	8	0,4	0,2	77,6000%	12824	4,4807	1043,1	78,3333%	10620	3,5009	822,7
4	2	0,1	0,2	78,3333%	12072	2,3706	557,1	77,7667%	14506,8	3,4942	815,2
4	2	0,01	0,2	79,7333%	13996	2,3386	559,4	77,7667%	16292,6	3,4646	808,3
4	2	0,001	0,2	79,0000%	16000	2,4034	569,6	77,6667%	18092,4	3,4412	801,8
4	2	0,4	0,2	78,0333%	10891	2,3644	553,5	77,5333%	13331,8	3,4819	809,9
64	4	0,01	0,2	76,9333%	12678	4,4536	1027,9	77,5000%	10357	3,4228	795,8
16	4	0,001	0,2	76,2000%	13034	4,4689	1021,6	77,1000%	10804	3,4527	798,6
2	8	0,01	0,2	76,7667%	24076	8,7516	2015,5	77,0667%	25788,9	9,4827	2192,4
16	4	0,01	0,2	75,7333%	12831	4,4789	1017,6	76,9333%	10741	3,5035	808,6
16	4	0,8	0,2	76,5667%	12666	4,4954	1032,6	76,9333%	10412	3,4974	807,2
64	4	0,001	0,2	77,7667%	12964	4,5118	1052,6	76,8667%	10337	3,4254	789,9
4	8	0,001	0,2	75,8667%	14917	4,4552	1014	76,7667%	19492,5	6,4386	1482,8
64	8	0,01	0,2	75,9333%	12594	4,4754	1019,5	76,7667%	10330	3,4438	793,1
2	8	0,4	0,2	77,9000%	22451	8,7694	2049,4	76,7000%	23856,9	9,5280	2192,4
8	4	0,01	0,2	74,0333%	12924	4,4557	989,6	76,7000%	11046	3,4846	801,8
16	8	0,001	0,2	77,5000%	13112	4,4275	1029,4	76,7000%	10745	3,4450	792,7
16	8	0,1	0,2	76,6333%	12643	4,4158	1015,2	76,5667%	10457	3,4680	796,6
16	8	0,8	0,2	76,5000%	12601	4,4710	1026,1	76,5667%	10213	3,4275	787,3
4	2	0,8	0,2	74,2000%	10212	2,4385	542,8	76,3333%	12671,3	3,4904	799,3
8	8	0,8	0,2	77,0000%	12721	4,4848	1036	76,2000%	10249	3,4506	788,8
4	4	0,001	0,2	76,3667%	15084	4,4972	1030,3	76,0667%	12684	3,4632	790,3
8	4	0,001	0,2	75,4333%	13428	4,4406	1004,9	76,0667%	11271	3,4584	789,2
4	4	0,1	0,2	75,9667%	13307	4,4590	1016,2	76,0333%	11101	3,4879	795,6
16	4	0,4	0,2	76,3667%	12631	4,4688	1023,8	76,0000%	10291	3,4640	789,8
64	8	0,8	0,2	76,2667%	12384	4,3977	1006,2	75,9667%	10190	3,4524	786,8
2	4	0,1	0,2	75,6333%	15517	4,8391	1098	75,8000%	22107,8	7,7872	1770,8
16	8	0,01	0,2	76,9000%	12921	4,4499	1026,6	75,8000%	10550	3,4719	789,5
8	8	0,4	0,2	75,8000%	12717	4,5075	1025	75,8000%	10244	3,4200	777,7
64	8	0,1	0,2	75,2000%	12467	4,4801	1010,7	75,7333%	10130	3,4199	777

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
64	8	0,001	0,2	74,9000%	12478	4,4682	1004	75,7000%	10332	3,4760	789,4
4	8	0,01	0,2	75,9667%	13985	4,3989	1002,5	75,6333%	18592,7	6,4910	1472,8
4	8	0,4	0,2	75,8667%	12828	4,4714	1017,7	75,5667%	17432,6	6,5457	1483,9
64	4	0,4	0,2	74,6667%	12323	4,4594	998,9	75,5667%	10244	3,4892	791
8	4	0,8	0,2	76,6333%	12535	4,4254	1017,4	75,5667%	10256	3,4826	789,5
4	4	0,8	0,2	75,4333%	12493	4,4569	1008,6	75,5667%	10248	3,4588	784,1
4	4	0,4	0,2	75,3333%	12834	4,5049	1018,1	75,5667%	10456	3,4420	780,3
4	4	0,01	0,2	77,2000%	14349	4,4845	1038,6	75,4667%	11793	3,4585	783
2	4	0,001	0,2	75,3333%	20026	4,8093	1086,9	75,3000%	26439,7	7,7795	1757,4
8	4	0,4	0,2	75,5333%	12556	4,4523	1008,9	75,2333%	10380	3,5060	791,3
8	4	0,1	0,2	76,6000%	12925	4,4591	1024,7	75,0333%	10429	3,4434	775,1
64	4	0,8	0,2	76,8667%	12546	4,4337	1022,4	74,9667%	10126	3,4700	780,4
64	8	0,4	0,2	77,1667%	12564	4,4190	1023	74,8333%	10119	3,4677	778,5
2	4	0,4	0,2	73,8667%	13831	4,8213	1068,4	74,8000%	20364	7,7108	1730,3
8	8	0,01	0,2	76,0667%	13192	4,4540	1016,4	74,6333%	10870	3,5025	784,2
8	8	0,001	0,2	76,7667%	13678	4,4720	1029,9	74,6333%	11212	3,4984	783,3
8	8	0,1	0,2	75,4000%	12792	4,4713	1011,4	74,6000%	10399	3,4500	772,1
4	8	0,1	0,2	76,5667%	13337	4,4380	1019,4	74,4000%	17423,7	6,4310	1435,4
2	2	0,001	0,2	74,9667%	20744	3,1343	704,9	74,1667%	21206	3,4580	769,4
2	4	0,01	0,2	76,4000%	17907	4,8255	1106	74,0333%	24024,3	7,8307	1739,2
64	4	0,1	0,2	75,7000%	12435	4,4364	1007,5	74,0000%	9998	3,4405	763,8
2	2	0,01	0,2	73,5000%	16953	3,0816	679,5	73,9000%	17672,7	3,4456	763,9
4	8	0,8	0,2	75,3000%	12401	4,4250	999,6	73,6333%	16523	6,4138	1416,8
2	2	0,1	0,2	73,9333%	13521	3,1109	690	73,5667%	14177,8	3,4608	763,8
2	2	0,4	0,2	50,7333%	9237	3,1176	474,5	73,3667%	12054,5	3,4593	761,4
Cluster C_A^*											
32	8	0,4	0,01	86,1667%	43964	1,4998	387,7	58,1333%	50021	6,5470	1141,8
32	8	0,1	0,01	86,0000%	44370	1,5124	390,2	58,0333%	50532	6,6175	1152,1
32	8	0,1	0,2	84,7000%	18821	1,5226	386,9	57,7667%	25070,7	6,1160	1059,9
32	8	0,8	0,01	86,4000%	43804	1,5073	390,7	57,4333%	50036,6	6,6709	1149,4
32	8	0,4	0,1	85,5667%	24361	1,5138	388,6	57,3667%	30749,1	6,3806	1098,1
32	8	0,01	0,01	85,8333%	45005	1,5161	390,4	56,9333%	50844,5	6,5287	1115,1
32	8	0,001	0,01	85,3000%	45598	1,5104	386,5	56,7000%	51507,5	6,6414	1129,7
32	8	0,001	0,1	85,8000%	26052	1,5272	393,1	56,4333%	32553,8	6,6096	1119
32	8	0,1	0,1	85,1000%	24720	1,5135	386,4	56,4000%	31167,2	6,4994	1099,7
32	8	0,8	0,1	87,3000%	24226	1,5048	394,1	56,4000%	30307	6,3174	1068,9
32	8	0,01	0,2	85,6333%	19481	1,5165	389,6	56,3667%	26119,4	6,5482	1107,3
32	8	0,4	0,2	84,1333%	18348	1,4964	377,7	56,0333%	25377,4	6,7216	1129,9
32	8	0,001	0,2	85,7333%	20117	1,5163	390	55,8667%	26396	6,3842	1070
32	8	0,8	0,2	85,1667%	18257	1,5170	387,6	55,2000%	24626,4	6,4662	1070,8
32	8	0,01	0,1	85,3667%	25313	1,4932	382,4	55,0000%	31560,2	6,5339	1078,1
Cluster C_A											
2	1	0,1	0,01	86,2333%	22478	1,5052	389,4	46,1667%	19721,3	1,0000	137,3

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
16	1	0,1	0,2	85,5333%	34517	1,5016	385,3	45,6333%	31521	1,0000	136,2
4	1	0,001	0,01	86,2667%	33952	1,5073	390,1	45,6000%	31034,2	1,0000	136,2
32	2	0,1	0,1	85,4000%	24664	1,4785	378,8	45,4000%	22238	1,0000	136,2
32	2	0,1	0,01	85,6333%	44338	1,4916	383,2	45,2000%	41862	1,0000	135,6
64	1	0,4	0,2	85,0333%	115117	1,4790	377,3	45,2000%	110785,8	1,0000	134,7
16	1	0,001	0,01	85,7333%	90867	1,5171	390,2	45,0333%	86925,8	1,0000	134,3
64	1	0,001	0,01	86,6000%	325956	1,4858	386	45,0000%	317738,7	1,0000	134,6
32	2	0,01	0,2	86,0000%	19408	1,4764	380,9	44,8000%	16943	1,0000	134,4
32	1	0,1	0,1	84,5000%	24701	1,5089	382,5	44,6333%	22215	1,0000	133,9
4	1	0,4	0,01	86,1333%	25405	1,4957	386,5	44,6000%	22610,6	1,0000	132,5
32	2	0,4	0,01	86,7333%	44005	1,4915	388,1	44,5000%	41459	1,0000	133,5
8	1	0,4	0,01	86,2000%	44832	1,4896	385,2	44,5000%	41693,6	1,0000	132,9
32	2	0,001	0,1	84,6667%	25966	1,5055	382,4	44,4667%	23476	1,0000	133,4
64	1	0,001	0,1	85,6333%	168866	1,4691	377,4	44,4333%	163713,6	1,0000	132,6
32	1	0,8	0,01	86,4000%	43847	1,5096	391,3	44,3667%	41265	1,0000	133,1
16	1	0,01	0,1	85,2667%	48917	1,4934	382	44,3667%	45674,4	1,0000	131,7
64	1	0,1	0,2	85,7000%	116706	1,5018	386,1	44,3333%	112346,8	1,0000	131,2
8	1	0,001	0,1	86,1667%	32508	1,4928	385,9	44,3000%	29548,5	1,0000	131,7
4	1	0,1	0,01	85,6000%	27340	1,4918	383,1	44,2000%	24511,1	1,0000	131,5
32	2	0,8	0,1	86,4333%	24145	1,4813	384,1	44,1000%	21627	1,0000	132,3
8	1	0,4	0,2	85,1667%	19247	1,4881	380,2	44,0667%	16539,8	1,0000	131,4
4	1	0,01	0,01	85,8000%	30657	1,5062	387,7	44,0000%	27757,3	1,0000	131,2
64	1	0,001	0,2	85,5667%	121642	1,4854	381,3	44,0000%	116962	1,0000	131,1
32	1	0,8	0,2	83,9333%	18158	1,4948	376,4	43,9667%	15713	1,0000	131,9
16	1	0,4	0,1	85,0333%	44724	1,4971	381,9	43,9000%	41536,5	1,0000	130,7
32	4	0,1	0,2	85,7000%	18845	1,5088	387,9	43,8000%	16280	1,0000	131,4
8	1	0,8	0,2	85,6000%	18524	1,5276	392,3	43,8000%	15697	1,0000	130,9
16	1	0,001	0,2	85,3333%	39739	1,5004	384,1	43,8000%	36643,5	1,0000	130,4
32	1	0,8	0,1	85,4333%	24164	1,5060	386	43,7333%	21616	1,0000	131,2
16	1	0,01	0,01	85,7333%	88237	1,5121	388,9	43,6667%	84375,4	1,0000	130,4
32	1	0,01	0,2	85,3000%	19463	1,5100	386,4	43,6333%	16908	1,0000	130,9
32	2	0,01	0,01	86,4000%	44999	1,4892	386	43,6000%	42447	1,0000	130,8
8	1	0,1	0,2	85,2667%	21003	1,5129	387	43,5667%	18183,3	1,0000	129,7
32	2	0,001	0,2	85,3333%	20045	1,4895	381,3	43,5333%	17538	1,0000	130,6
16	1	0,1	0,1	86,4667%	46358	1,4950	387,8	43,5000%	43153,1	1,0000	129,6
8	1	0,1	0,1	84,8000%	26862	1,5016	382	43,5000%	24005,7	1,0000	129,8
16	1	0,001	0,1	84,3667%	51495	1,4939	378,1	43,4667%	48240,8	1,0000	129,6
32	2	0,1	0,2	86,2000%	18835	1,4961	386,9	43,4333%	16269	1,0000	130,3
64	1	0,01	0,01	85,9333%	323499	1,5109	389,5	43,4000%	315476,7	1,0000	129,4
32	1	0,01	0,01	85,9000%	45015	1,5041	387,6	43,4000%	42441	1,0000	130,2
8	1	0,01	0,01	85,4000%	49298	1,4934	382,6	43,4000%	46112,5	1,0000	129,7
2	1	0,001	0,01	85,7333%	34461	1,5152	389,7	43,3667%	31480,3	1,0000	128,5
32	1	0,4	0,1	84,9000%	24325	1,5041	383,1	43,3000%	21793	1,0000	129,9

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
64	1	0,01	0,2	86,2667%	119192	1,4896	385,5	43,2667%	114764,8	1,0000	128,9
32	4	0,8	0,01	85,6000%	43784	1,4992	385	43,2667%	41232	1,0000	129,8
32	1	0,001	0,2	85,4667%	20057	1,4918	382,5	43,2667%	17530	1,0000	129,8
32	4	0,1	0,01	85,9333%	44369	1,4984	386,3	43,2000%	41802	1,0000	129,6
32	1	0,4	0,2	85,5000%	18405	1,4897	382,1	43,2000%	15880	1,0000	129,6
8	1	0,001	0,2	84,8667%	26543	1,4937	380,3	43,1667%	23705,3	1,0000	128,6
32	2	0,4	0,2	84,7667%	18436	1,5147	385,2	43,1333%	15878	1,0000	129,4
16	1	0,8	0,2	86,1000%	32215	1,5153	391,4	43,1000%	29152,5	1,0000	128,3
32	4	0,01	0,01	85,6000%	45007	1,5062	386,8	43,1000%	42432	1,0000	129,3
4	1	0,1	0,1	84,2667%	17465	1,4909	376,9	43,1000%	14806,5	1,0000	128,6
32	4	0,001	0,01	86,1333%	45613	1,4865	384,1	43,0333%	43063	1,0000	129,1
32	1	0,01	0,1	84,8333%	25336	1,5037	382,7	43,0000%	22799	1,0000	129
4	1	0,01	0,1	82,6333%	20692	1,5026	372,5	43,0000%	18026,6	1,0000	128,2
64	1	0,01	0,1	86,8667%	166532	1,5088	393,2	42,9667%	161322,7	1,0000	128,5
16	1	0,8	0,1	85,0667%	43914	1,4879	379,7	42,9667%	40729,6	1,0000	127,9
64	1	0,1	0,01	86,4333%	320996	1,4979	388,4	42,9000%	313044,3	1,0000	128,1
8	1	0,01	0,1	85,4000%	29661	1,4891	381,5	42,8667%	26746,6	1,0000	127,5
64	1	0,8	0,1	86,2000%	161773	1,5143	391,6	42,8333%	156639,9	1,0000	128,1
16	1	0,01	0,2	86,1667%	37170	1,5044	388,9	42,8333%	34040,5	1,0000	127,8
32	4	0,01	0,2	84,1667%	19390	1,5014	379,1	42,8333%	16884	1,0000	128,5
2	1	0,4	0,01	82,6000%	18717	1,5093	374	42,8333%	16062,8	1,0000	126,6
16	1	0,8	0,01	86,5000%	83260	1,4998	389,2	42,8000%	79421,3	1,0000	127
64	1	0,4	0,01	84,6667%	319408	1,4949	379,7	42,8000%	311690	1,0000	127,3
32	1	0,1	0,01	86,5333%	44435	1,5135	392,9	42,7667%	41789	1,0000	128,3
8	1	0,8	0,01	84,7667%	43918	1,4872	378,2	42,7667%	40814,6	1,0000	127,1
4	1	0,4	0,1	83,2667%	15394	1,4680	366,7	42,7667%	12854,2	1,0000	127,6
64	1	0,8	0,2	85,0000%	114381	1,4851	378,7	42,7333%	110043,2	1,0000	126,5
8	1	0,001	0,01	86,2000%	52043	1,4571	376,8	42,7000%	48855,5	1,0000	127,4
32	4	0,4	0,2	85,2667%	18444	1,5090	386	42,7000%	15865	1,0000	128,1
8	1	0,4	0,1	84,8333%	25194	1,5088	384	42,6667%	22321,7	1,0000	127,2
64	1	0,8	0,01	86,2667%	318696	1,4818	383,5	42,6333%	310960,6	1,0000	127,4
32	2	0,001	0,01	86,3333%	45675	1,5069	390,3	42,6000%	43050	1,0000	127,8
32	4	0,4	0,01	86,3000%	44057	1,5191	393,3	42,6000%	41402	1,0000	127,8
64	1	0,1	0,1	85,7667%	163959	1,4967	385,1	42,6000%	158835,8	1,0000	126,6
32	2	0,01	0,1	86,1000%	25396	1,5048	388,7	42,5667%	22786	1,0000	127,7
64	1	0,4	0,1	85,4000%	162460	1,5039	385,3	42,5667%	157228,2	1,0000	126,4
32	4	0,1	0,1	84,7333%	24698	1,5035	382,2	42,5333%	22152	1,0000	127,6
16	1	0,4	0,01	86,9000%	84054	1,4952	389,8	42,5000%	80201,4	1,0000	126,9
32	1	0,001	0,1	84,7333%	25960	1,5020	381,8	42,5000%	23417	1,0000	127,5
8	1	0,1	0,01	86,2667%	46530	1,4923	386,2	42,3000%	43284,8	1,0000	125,8
32	2	0,8	0,01	85,9333%	43778	1,4911	384,4	42,2667%	41202	1,0000	126,8
8	1	0,8	0,1	85,6000%	24401	1,5152	389,1	42,2333%	21482	1,0000	125,9
32	4	0,4	0,1	84,8000%	24282	1,4890	378,8	42,2333%	21761	1,0000	126,7

Fortgesetzt auf der nächsten Seite

Tabelle A.1 – *Forgesetzt von der vorherigen Seite*

		Bloom Fpr		Update				Reg			
v	**b**	**Inner**	**Blatt**	**G**	**B**	**Red**	**Kvv**	**G**	**B**	**Red**	**Kvv**
32	4	0,001	0,1	85,1000%	25943	1,4888	380,1	42,2000%	23408	1,0000	126,6
16	1	0,1	0,01	85,7667%	85541	1,4808	381	42,1000%	81665,7	1,0000	125,5
32	4	0,001	0,2	85,7333%	20129	1,5152	389,7	42,0667%	17494	1,0000	126,2
32	1	0,1	0,2	85,3333%	18804	1,4992	383,8	42,0667%	16228	1,0000	126,2
32	2	0,8	0,2	84,5333%	18157	1,4838	376,3	42,0000%	15654	1,0000	126
4	1	0,001	0,1	84,2000%	24019	1,4968	378,1	42,0000%	21237,1	1,0000	125
2	1	0,001	0,1	79,6333%	28182	1,5107	360,9	42,0000%	25514,4	1,0000	124,3
32	1	0,001	0,01	86,0333%	45639	1,4983	386,7	41,9000%	43029	1,0000	125,7
32	4	0,8	0,2	84,7333%	18192	1,4941	379,8	41,8667%	15650	1,0000	125,6
8	1	0,01	0,2	85,1000%	23718	1,4810	378,1	41,8333%	20904,4	1,0000	124,8
32	1	0,4	0,01	86,2667%	43989	1,4934	386,5	41,7667%	41377	1,0000	125,3
32	4	0,8	0,1	85,5333%	24102	1,4801	379,8	41,5667%	21551	1,0000	124,7
2	1	0,01	0,01	84,3000%	28352	1,4939	377,8	41,3667%	25509,7	1,0000	122,7
4	1	0,4	0,2	80,1333%	12404	1,5104	363,1	41,3667%	9902,8	1,0000	123,6
32	2	0,4	0,1	85,4000%	24364	1,5105	387	41,2000%	21730	1,0000	123,6
4	1	0,01	0,2	79,0667%	17582	1,5046	356,9	40,5000%	15052,4	1,0000	120
32	4	0,01	0,1	87,0667%	25404	1,4912	389,5	40,4000%	22721	1,0000	121,2
2	1	0,1	0,1	80,0333%	16179	1,4935	358,6	40,2667%	13634,4	1,0000	119,1
4	1	0,001	0,2	81,5333%	20944	1,4963	366	40,2000%	18266,8	1,0000	118,9
16	1	0,4	0,2	86,6000%	32993	1,5027	390,4	39,3667%	29784,9	1,0000	117,4
4	1	0,1	0,2	78,9667%	14330	1,5146	358,8	39,3667%	11793,9	1,0000	117,7
2	1	0,01	0,1	79,0000%	22121	1,4928	353,8	38,6667%	19512,6	1,0000	114,8
2	1	0,4	0,1	77,5333%	12484	1,5039	349,8	37,3000%	9986,9	1,0000	110,4
2	1	0,1	0,2	71,8000%	14020	1,4995	323	37,1333%	11763,1	1,0000	110
2	1	0,01	0,2	71,6333%	19991	1,4942	321,1	36,3333%	17657,2	1,0000	108,4
2	1	0,001	0,2	72,0333%	26050	1,5178	328	34,6333%	23532,9	1,0000	102,5
2	1	0,4	0,2	68,6000%	10233	1,4820	305	33,7667%	8105,4	1,0000	99,9
4	1	0,8	0,1	68,5000%	13817	1,4964	307,5	33,5333%	11616,3	1,0000	99,5
4	1	0,8	0,01	70,5333%	23720	1,4957	316,5	33,2000%	21295,4	1,0000	99
4	1	0,8	0,2	64,3000%	10692	1,5054	290,4	30,5667%	8604,1	1,0000	90,8
2	8	0,8	0,01	21,1333%	8728	8,7461	554,5	30,3333%	11874,8	9,6011	873,7
2	8	0,8	0,2	16,9333%	6092	8,8661	450,4	27,5000%	9377,9	9,4655	780,9
2	4	0,8	0,01	22,9000%	8760	4,8079	330,3	26,9667%	11470,9	7,6341	617,6
2	8	0,8	0,1	27,6333%	9251	8,7986	729,4	24,1333%	8859,3	9,5691	692,8
2	4	0,8	0,1	11,9000%	4900	4,9132	175,4	20,4333%	7643,3	7,4715	458
2	2	0,8	0,01	13,6000%	9321	3,1667	129,2	17,5000%	9703	3,3733	177,1
2	2	0,8	0,2	16,2333%	4946	3,1191	151,9	12,6667%	4647,3	3,3105	125,8
2	2	0,8	0,1	9,8667%	5438	3,1959	94,6	9,6667%	5445,3	3,4621	100,4
2	4	0,8	0,2	12,0333%	4205	4,8587	175,4	7,8000%	4115,3	7,3846	172,8
2	1	0,8	0,01	0,0000%	13174	0,0000	0	5,1667%	13174,6	1,0000	15,4
2	1	0,8	0,1	0,0000%	7183	0,0000	0	3,9667%	7219,5	1,0000	11,8
2	1	0,8	0,2	0,0000%	5380	0,0000	0	3,7333%	5430,9	1,0000	11,1