Replikation

Konsistenzmodelle, -protokolle (Fock)
Verwaltung & Umgebung (Polydor)

Inhalt

1. Einführung

1.1. Gründe für Replikation

Replikation hat einen sehr hohen Stellenwert bei größeren Firmen, um deren Daten abzusichern und die Dienste ausfallsicher zu gestalten. Sie wird auch oft im Nachhinein eingesetzt, damit bestehende Infrastruktur/Systeme ausfallsicherer bzw. performanter werden. Replikation vermindert lange Ladezeiten/Latenzzeiten, da die Server an strategisch ausgewählten Standorten platziert werden und auf diese repliziert wird. Dabei werden auch mögliche Ausfälle miteinbezogen. Dadurch stehen auch andere Replikatserver bereit, die bei einem Ausfall (Knotenausfall) für diesen einspringen, damit die Verfügbarkeit der Inhalte/Dienste erhalten bleibt. Des Weiteren dient Replikation auch als eine Art Sicherheitskopie, da derselbe Wert, der von mindestens zwei Kopien zurückgeliefert wird, als richtig betrachtet wird. Allerdings gibt es für die Verwendung von Replikation auch Einwände. Diese werden bei der Replikation von Daten mit der Einschränkung erkauft, nachdem mehrere Kopien vorhanden sind. Zwangsläufig muss bei einer Änderung an einem dieser Replikate, bei allen anderen dieselbe Änderung durchgeführt werden. Dadurch kann es zu Konsistenzproblemen kommen, da die Replikate nicht nahe beieinanderliegen müssen. Daraus folgt, dass weite Strecken zurückgelegt werden müssen, damit diese angepasst werden können. [1] [2] [9]

1.2. Replikation als Skalierungstechnik

„Leistungsorientierte Replikation und Caching (Zwischenspeicherung) sind als Skalierungstechniken weit verbreitet. Skalierbarkeitsprobleme treten im Allgemeinen in Gestalt von Leistungseinschränkungen auf. Die Platzierung von Datenkopien in der Nähe der Prozesse, die sie verwenden, kann die Leistung dank der Verringerung der Zugriffszeiten erhöhen und auf diese Weise Skalierbarkeitsprobleme beheben.“ [1]

Dabei kann der Umstand auftreten, dass mehr Bandbreite benötigt wird, da die Kopien auf dem Laufenden gehalten werden müssen, weil diese sonst inkonsistent wären. Je mehr Änderungen auftreten, desto mehr Bandbreite wird benötigt. Dabei erschwert der mögliche Umstand, der sehr hochfrequenten Änderungsoperationen die Reduzierung der Bandbreite. [1]

Ein weitaus ernsthafteres Problem ist, dass eine Sammlung von Kopien erst dann konsistent ist, wenn alle Kopien immer identisch sind. Dies bedeutet, dass eine beliebige Leseoperation von einer Kopie, dasselbe Ergebnis wie bei allen anderen Kopien hervorruft. Daher ist es ratsam/sinnvoll, wenn eine Aktualisierung, die an einer Kopie vorgenommen wird, an allen anderen existenten Kopien ebenfalls vorgenommen wird, bevor eine weitere Operation auf diesen stattfinden kann. Dabei sollte es egal sein, von welcher Kopie die Operation ausgeht. Diese Art der Konsistenz wird bisher informell (unpräzise) als enge Konsistenz (Tight Consistency) bezeichnet, welche von einer sogenannten synchronen Replikation ermöglicht

wird. Dabei ist die Schlüsselaufgabe, eine Aktualisierung an allen Kopien in Form einer atomaren Operation, oder Transaktion vorzunehmen. Aber dies gestaltet sich doch sehr schwierig, angesichts von Kopien die sich in einem weit ausgedehnten Netzwerk befinden und die dadurch weit verstreut angeordnet sein können. Diese Operationen müssen natürlich auch sehr schnell von statten gehen. [1]

Dabei ergibt sich die Schwierigkeit aus dem Umstand, dass alle Replikate miteinander synchronisiert werden müssen und sich diese dadurch zuerst einig werden müssen, wann eine Aktualisierung tatsächlich lokal erfolgen soll. Replikate müssen sich z.B. auf eine globale Operationsreihenfolge verständigen, welche durch Lamport-Zeitstempel ermöglicht werden, oder über einen Koordinator. Die Synchronisierung über ein WAN, wenn die Replikate weit verbreitet sind, benötigt enorm viele Kommunikationsressourcen. [1]

Einerseits werden Skalierbarkeitsprobleme durch Anwendung von Replikation und Caching verringert. Andererseits wird im Allgemeinen eine globale Synchronisierung nötig, damit alle Replikate/Kopien laufend aktuell gehalten werden. Bei den meisten Fällen/Situation werden dafür die Konsistenzbeschränkungen gelockert, da dies sonst enorme Leistungseinbußen mit sich bringt. Dabei hat sich erwiesen, dass das Ausmaß einer möglichen Lockerung der Konsistenz stark von dem Zugriff- und Aktualisierungsverhalten gegenüber den replizierten Daten sowie von deren Zwecken abhängt, für die sie verwendet werden. [3]

1.3. Synchrone / Asynchrone Replikation

Zwei Arten von Replikation sind derzeit existent, die synchrone- und asynchrone Replikation. Die synchrone setzt darauf auf, dass bei der Änderungsoperation an einem Datenobjekt, nur dann erfolgreich abgeschlossen werden kann, wenn diese auch auf allen anderen Replikaten durchgeführt wurde. Damit dieses Verfahren auch Regelkonform umgesetzt wird, wird der zwei Phasen Commit benötigt. Bei diesem müssen alle an diese Änderungsoperation gebunden Server dieser zustimmen. Ansonsten wird ein Rollback durchgeführt und die Änderungsoperation als ungültig definiert. Diese Art ist Performancetechnisch nicht schnell, weil sie von allen eingebunden Servern abhängig ist und diese die Anfrage erst erhalten müssen und danach auf die Antwort gewartet werden muss (direkte Synchronisation). [3]

Die asynchrone Replikation ist, wenn zwischen der Bearbeitung der primären Daten und den Replikaten eine Latenz vorliegt. Die Daten sind nur zum Zeitpunkt der Replikation synchron. Asynchrone Replikationsverfahren können mittels Read-Only oder Update-Anywhere realisiert werden. Bei Update-Everywhere kann auf allen Datenbanken geschrieben werden und es entsteht eine Aktualisierung in beide Richtungen. Bei Read-Only können die Daten nur von der primären Datenbank auf die Replizierten repliziert werden. [3]

2. Umgebung

Das Kernproblem mit dem sich die Replikationsverwaltung auseinandersetzt ist, wo werden Replikatserver platziert, wann und von wem werden Replikate platziert. Beziehungsweise welche Mechanismen halten diese Daten dann konsistent. Dies darf unter keinen Umständen in Vergessenheit geraten und auch mit diesem Problem/dieser Thematik befasst sich die Replikationsverwaltung. [1]

2.1. Platzierung der Replikatserver

Aus einem einfachen Grund, ist die Platzierung von Replikatservern aktuell noch nicht wirklich intensiv erforscht worden. Dieser ist, dass normal die Platzierung eher eine verwaltungstechnische und kommerzielle Frage ist, als ein Optimierungsproblem. Trotzdem hat sich hierbei die Analyse der Clients und des Netzwerkes als sinnvolle Unterstützung bei der Platzierung etabliert, beziehungsweise erwiesen. [1]

Für diese Aufgabe gibt es aber auch schon diverseste Methoden, für die Berechnung der optimalen Platzierung von Replikatservern. Aber all diese haben dasselbe Problem. Wenn die besten K aus N Standorten ausgewählt werden müssen (K < N), dann läuft die Situation auf ein Optimierungsproblem hinaus, welches bekannt ist, dass es sich mathematisch nur sehr schwer berechnen lässt. Dazu bleiben nur heuristische Methoden übrig. Dabei ist es eben wichtig zu beachten, wenn K < N gilt, dass nicht alle Standorte ideal sein können. [1]

2.1.1. Ansatz von Qui et al

Bei diesem Ansatz wird auf die räumliche Entfernung zwischen den Standorten und den Clients gesetzt. Dabei wird die Entfernung mit Hilfe der Bandbreite oder der Latenzzeit gemessen. Diese Lösung wählt einen Server nach dem anderen auf eine Weise aus, welche die durchschnittliche Distanz zwischen dem gewählten Server und seinen Clients verringert, indem er versucht den Abstand zu diesen zu vermindern, indem er davon ausgeht, dass bereits k Server platziert wurden (N – k Standorte sind noch verfügbar). [1] [3] [13]

2.1.2. Ansatz von Radoslav et al

Dieser Ansatz ignoriert die Standorte der Clients komplett und berücksichtigt nur die Topologie des Internets, welche sich aus autonomen Systemen zusammensetzt. Dabei beschreibt sich ein autonomes System (AS) am besten, als ein Netzwerk, in welchem alle Knoten dasselbe Routing-Protokoll ausführen. Dieses wird hierbei von einer einzelnen Organisation verwaltet. Als erstes wird das größte AS gesucht und platziert einen Server an dem Router mit der größten Anzahl von Netzwerkschnittstellen (d. h. Verknüpfungen). Diese Prozedur wird danach ebenso mit dem zweitgrößten AS, dem drittgrößten AS, usw. wiederholt. Mittels der clientunabhängigen Platzierung werden ähnliche Ergebnisse erzielt, wie mit der clientorientierten, vorausgesetzt, dass die

Clients sich gleichmäßig über das Internet verteilen (relativ zur bestehenden Topologie). Inwiefern diese Aussage korrekt ist, lässt sich nicht wirklich ermitteln, da diese noch fast nicht erforscht wurde. [1] [3] [13]

Diese beiden Methoden benötigen allerdings in der Phase der Berechnung sehr lange. Selbst für nur wenige tausende Standorte werden Dutzende Minuten benötigt. Bei dieser Dauer, kann es für so manches Unternehmen sehr ungemütlich werden, indem es doch zu möglichen Flash Crowds (plötzliche Ausbrüche von Anfragen nach einer bestimmten Website, die im Internet regelmäßig auftreten) kommen kann. In solch einer Situation muss schnell bestimmt werden, wo ein Replikatserver benötigt und platziert werden muss, damit ein einzelner für die Inhaltsplatzierung gewählt werden kann. [1]

2.1.3. Ansatz von Szymaniak et al

Dieser Ansatz entwickelte eine Methode, mit deren Hilfe sich eine Region für die Platzierung von Replikaten sehr schnell identifizieren lässt. Dabei wird unter einer Region eine Sammlung von Knoten verstanden, welche auf dieselben Inhalte zugreifen und zwischen denen nur eine geringe Latenz besteht. Bei dieser Methodik werden als erstes die anspruchsvollsten Regionen ausgewählt (Regionen mit den meisten Knoten, und einer dieser Knoten in einer dieser Regionen wird als Replikatserver ausgewählt). Damit dies funktioniert, wird der gesamte Raum in Zellen unterteilt und die k-dichtesten Zellen werden dann für die Platzierung eines Replikatservers ausgewählt. Dabei ist die Größe der Zelle von großer Bedeutung. Falls die Zellengröße zu groß gewählt wird, kann es passieren, dass sich mehrere Knotencluster in derselben Zelle befinden. In diesem Fall, hätte die auf diese zutreffende Zelle eine zu geringe Anzahl an Replikatservern erhalten. Ebenso kann eine zu kleine Wahl der Zellengröße bewirken, dass ein einzelner Cluster sich über mehrere Zellen erstreckt, sodass zu viele Replikatserver verfügbar wären. [1] [3] [13]

Dabei hat sich herauskristallisiert, dass eine angemessene Zellengröße als einfache Funktion der durchschnittlichen Entfernung zwischen zwei Knoten und der Anzahl der benötigten Replikate errechnen lässt. Weiters lässt sich beweisen, dass der Algorithmus mit dieser Zellengröße genauso gut arbeitet wie mit der Auswahl der Replikatserver des Ansatzes von Qui et al. Der hierbei entstandene Algorithmus ist ebenfalls so gut, wie der von Qui et al, nur, dass er nicht annähernd so komplex wie dieser ist. Als kleine Vergleichsmöglichkeit existiert folgendes Experiment: die 20 besten Standorte für Replikate innerhalb von 64.000 Knoten werden rund 50.000 Mal schneller berechnet, was bedeutet, die Replikatserverplatzierung kann durch/bei diesem Verfahren in Echtzeit mitverfolgt werden. [1]

3. Verwaltung

3.1. Replikation und Platzierung von Inhalten

Bei der Replikation und der Platzierung von Inhalten kann in drei verschiedene Ansätze von Replikation unterteilt werden:

- Permanente Replikate
- Server initiierte Replikate
- Client initiierte Replikate

[1] [13]

3.1.1. Permanente Replikate

Von permanenten Replikaten wird gesprochen, wenn die ursprüngliche Sammlung von Replikaten betrachtet wird, aus denen sich ein verteilter Datenspeicher zusammensetzt. Meistens ist die Anzahl dauerhafter Replikate gering. Bei einer Website kann deren Verteilung auf zwei unterschiedliche Arten erfolgen.

Für die Erste werden die Daten die die Website ausmachen auf eine limitierte Anzahl an Servern an einem Standort repliziert. Ankommende Anfragen bei diesem Standort, werden mittels Load Balancing auf die verfügbaren Server aufgeteilt, damit nicht eine Überlastung eines einzelnen oder mehrerer Server daraus resultiert. Diese Methode ist für den Client nicht spürbar, da er nichts von der Replikation mitbekommt.

Bei der Zweiten Methode wird als Spiegelung (Mirroring) bezeichnet und bei ihr wird die Website auf eine limitierte Anzahl an Servern kopiert, welche als Mirror-Websites betitelt werden. Dabei sind diese Server über das ganze Internet geografisch verteilt. Hier wird meist dem Client eine Liste bereitgestellt, bei der er sich simpel seinen Mirror auswählen kann. Diese Websites haben die geringe Anzahl an Replikaten mit clusterbasierten Websites gemeinsam.

[1] [13]

3.1.2. Server initiierte Replikate

Anders als bei der vorher behandelten Methode sind die vom Server initiierten Replikate von Temporärer Natur und daher nicht permanent existent. Diese werden erst erstellt, wenn Beispielsweise ein platzierter Webserver ohne Probleme die Anfragen abarbeiten kann, aber aus irgendeinem Grund treten für eine gewisse Zeitspanne erhöhte Zugriffsanfragen auf. In diesem Fall ist es durchaus lohnenswert, temporäre Replikate in der Region zu erstellen, aus der/denen die Flut an Anfragen auftritt. Dieses Problem der dynamisch platzierten Replikate wird auch aktiv von

Webhosting-Diensten erkannt und sie bieten dazu eine (relativ statische) Sammlung von Servern an, die über das Internet verteilt sind und den Zugriff auf die Webinhalte Dritter ermöglichen und aufrechterhalten. Damit diese Dienstleistung von den Betreibern überhaupt optimiert werden kann, müssen Dateien dynamisch auf Server repliziert werden, auf denen sie benötigt werden, um die Performance zu erhöhen.

Für diesen Fall existieren natürlich auch schon Ansätze. Einer davon ist der Rabinovich et al Ansatz, welcher speziell für das Einsatzgebiet auf Webseiten optimiert wurde. Daher geht er davon aus, dass Aktualisierungen/Schreibzugriffe seltener erfolgen und meist Lesezugriffe auftreten. Dieser Algorithmus baut auf dem Prinzip auf, indem er die Dateien als eine Dateneinheit einsetzt. Er berücksichtig zwei Schwierigkeiten der dynamischen Replikation:

1. Belastung eines Servers verringern
2. Bestimmte Dateien können von einem auf den anderen Server übertragen/verschoben oder repliziert werden (naheliegende Server, die sich in der Nähe von Clients befinden, die eine hohe Frequenz an Anfragen auf die bestimmte Datei haben)

Jeder Server überwacht die Zugriffszahlen jeder Datei und logt dabei mit, woher diese Zugriffe erfolgen. Es wird auch angenommen, dass für einen gegebenen Client C jeder Server ermitteln kann, welcher der Server des Webhosting-Dienstes am nächsten zum Client C liegt (Informationsgewinn aus zum Beispiel Routing-Datenbanken). Wenn hierbei für zwei Clients derselbe Server für sie am nächsten ist, dann wird der Zugriff dieser Clients auf dieselbe Datei vom Server zusammengefasst und protokolliert. Wenn die Anzahl der Anfragen nach einer bestimmten Datei auf dem Server unter einem vorher festgelegten Löschungswert liegt, kann die Datei vom Server gelöscht werden. Durch diese Vorgehensweise kann sich als Folge eine erhöhte Arbeitsbelastung anderer Server ergeben, da nun eine verringerte Anzahl an Replikaten für diese Datei verfügbar ist. Als wichtigster Punkt bei der Löschung, kontrolliert eine Maßnahme, dass sichergestellt ist, dass zumindest eine Kopie jeder Datei existent bleibt.

Dieses Verfahren gibt es auch in umgekehrter Funktionsweise. Dabei wird ein Replikationsgrenzwert festgelegt, der immer höher als der Löschungsgrenzwert gewählt werden muss. Der Replikationsgrenzwert gibt Aufschluss darüber, ob die Anzahl der Anfragen nach einer bestimmten Datei so hoch sind, dass es sich lohnt, diese auf einen anderen Server zu replizieren. Wenn der Wert der Anfragen zwischen diesen beiden liegt, dann wird der Datei höchstens eine Migration erlaubt. In dieser Situation wird der nächstgelegene Server doppelt so viele Anfragen erhalten, wie der Server mit der Kopie. Beim Eintritt dieses Ereignisses, wird der Server mit der Datei versuchen, diese auf den Server mit den vielen Anforderungen zu migrieren. Allerdings kann es auch passieren, dass der Server auf den versucht wird, die Datei zu migrieren,

schon eine zu hohe Auslastung hat. Hierbei kann die Datei nur repliziert werden, wenn die Anfragen den Replikationsgrenzwert überschreiten.

[1] [13]

3.1.3. Client initiierte Replikate

Sie stellt mitunter eine der wichtigsten Varianten dar, und wird auch als clientseitiger Cache bezeichnet. Beim Cache handelt es sich um eine lokale Speichervorrichtung am Client, die eine Kopie der gerade angeforderten Daten vorübergehend speichert und der Client kann diese verwenden. Die Verwaltung des Cache erfolgt nur durch den Client, welcher auch die vollständige Kontrolle über seinen Cache besitzt, damit ist er auch für die Aufrechterhaltung der Konsistenz seines Cache verantwortlich. Er kann sich aber vom Datenspeicher, von dem er die Datei angefordert hat, versichern lassen, dass er ihn informiert, sobald die im Cache gesicherte Datei veraltet ist. Die Dateien im Cache dienen nur einem Zweck, der Verkürzung von Zugriffszeiten auf Daten. Falls der Client beliebige Daten anfragt, verbindet er sich zur nächstgelegenen Kopie des Datenspeichers und fordert von ihm die gewünschten Daten an. Wenn hauptsächlich Lesezugriffe darauf auftreten, dann kann die Performance erhöht werden, indem es dem Client gestattet wird, dass er die angeforderten Daten in einem nahe gelegenen Cache speichern darf. Diese Art von Cache kann sich entweder auf dem Client selbst befinden, oder auf einem eigens dafür eingerichteten Computer, der im selben LAN wie der Client ansässig ist. Für den Erfolg dieses Verfahrens ist ein Punkt von äußerster Bedeutung, und zwar, dass die gecachten Daten in der Zwischenzeit nicht verändert werden. Daten im Cache werden nur für eine begrenzte Zeitdauer in diesem gehalten. Dadurch wird vermieden, dass sich extrem veraltete Daten im Cache befinden können, oder auch, um Platz für neue Daten zu schaffen. Bei jeder Anfrage zu einer Datei im Cache, tritt ein sogenannter Cache-Treffer (Cache-Hit) auf und hierbei wird die Anzahl der Cache-Treffer erhöht. Datencaches können auch von mehreren nahegelegenen Clients verwendet werden. Die Caches von einem Client werden normalerweise lokal am Rechner oder auf einem Rechner innerhalb desselben LAN erstellt. Auch hier gibt es noch weitere Anwendungsgebiete. Dabei kann ein gemeinsam verwendeter Cache installiert werden, der zwischen Abteilungen einer Organisation platziert sein kann, oder einen Cache innerhalb eines WANs.

[1] [13]

3.2. Verteilung von Inhalten

Für die Verteilung von Inhalten ist die Replikationsverwaltung auch verantwortlich. Dabei regelt sie die Weiterleitung von (aktualisierten) Inhalten an die beteiligten Replikatserver. Für diese gilt es natürlich, dass bestimmte Abwägungen dafür getroffen werden müssen. [1]

3.2.1. Zustand und Operationen

Für diesen Punkt existieren drei wesentliche Möglichkeiten:

1. Weiterleiten einer Benachrichtigung über Aktualisierung
2. Übertragen der Daten von einer Kopie zur nächsten
3. Weiterleiten der Aktualisierungsoption an andere Kopien

Bei der Aktualisierung durch eine weitergeleitete Benachrichtigung, kommen Invalidierungsprotokolle zur Anwendung. Mit dieser Art von Protokollen werden die anderen existenten Kopien darüber informiert, dass eine Aktualisierung stattgefunden hat und die auf ihnen vorhandenen Daten nicht mehr länger aktuell/gültig sind. In diesen Fällen kann bei der Invalidierung angegeben werden, dass nur ein bestimmter Teil des Datenspeichers aktualisiert werden muss und dadurch nicht die ganze Kopie als ungültig erklärt wird, sondern nur ein Teil von ihr. Der bei dieser Methodik entscheidende Punkt besteht darin, dass nichts anderes als eine simple Benachrichtigung über dies weitergeleitet wird. Im Weiteren vorgehen, muss bei jeder Operation die für eine für ungültig erklärte Kopie angefordert wird, diese aktualisiert werden, bevor die Möglichkeit der Durchführung der Operation an ihr für gültig erklärt werden kann. Diese Art von Protokollen zahlt sich am meisten aus, wenn sehr viele Aktualisierungsoperationen im Vergleich zu Lesezugriffe auftreten. Dazu ist ein Beispiel über einen Datenspeicher bei der Erläuterung von Vorteil. Wenn mehrere Aktualisierungen eines Replikates auftreten, aber dazwischen keine Lesezugriffe bei den anderen Replikaten auftreten, wäre es unnötig, wenn die Aktualisierung sofort an alle Replikate weitergeleitet wird. Diese wird dann sowieso von der/den nachfolgenden Aktualisierungen überschrieben, aber stattdessen wäre es sinnvoller gewesen, wenn eine Benachrichtigung übermittelt werden würde, dass die Daten verändert wurden.

Die Übertragung der veränderten Daten an die Replikate ist die zweite Alternative. Diese Methodik stützt sich auf ein relativ großes Verhältnis gegenüber Lese- und Schreibzugriffen und daher auf das komplette Gegenteil, als die vorherige Methode. Dabei besteht eine hohe Wahrscheinlichkeit, dass eine Aktualisierung effektiv ist, wenn die veränderten Daten vor der nächsten Aktualisierung gelesen werden. Weiters ist es möglich, anstatt der Weiterleitung der veränderten Daten, die Veränderungen zu protokollieren und nur diese Protokolle zu übertragen, um Bandbreite zu sparen. Darüber hinaus werden auch oft mehrere Veränderungen in eine einzige Nachricht zusammengefasst, um den kommunikativen Mehraufwand zu reduzieren.

Für den dritten Ansatz, wird jedem Replikat nur mitgeteilt, welche Änderungsoperationen es vornehmen muss und hierbei werden absolut keine Datenänderungen übertragen. Diese Herangehensweise setzt voraus, dass jedes Replikat von einem Prozess repräsentiert wird, welcher fähig ist, entsprechende Operationen auszuführen. Bei dieser Art besteht der Hauptvorteil darin, dass Aktualisierungen oft mit minimalstem Bandbreitenverbrauch weitergeleitet werden. Dabei ist aber auch vorausgesetzt, dass die Operation nicht allzu umfangreich ist. Des Weiteren sind der Komplexität der Operationen kaum Grenzen gesetzt. Dadurch wird es erlaubt, fast beliebig komplexe Operationen weiterzuleiten und dadurch kann es unter Umständen auftreten, dass manchen Replikaten mehr Rechenleistung abverlangt wird. Besonders stark tritt dieser Fall auf, wenn die Operationen zunehmend komplexer veranlagt sind.

[1]

3.2.2. Pull- und Push Protokolle

Eine weitere Gestaltungsfrage ist, ob die Aktualisierungen mittels Pull- oder Push-Prinzip übertragen werden. Die Push basierte Methode wird auch als serverbasiertes Protokoll bezeichnet. Dabei werden Aktualisierungen an die Replikate weitergeleitet, ohne dass diese um die Aktualisierung gebeten haben. Push-basierte Ansätze treten daher oft zwischen dauerhaften und vom Server initiierten Replikaten auf, aber diese werden auch bei Aktualisierungen von Clientcaches benutzt. Im Allgemeinen wird diese Methode verwendet, wenn ein hoher Konsistenzgrad erhalten werden muss (wenn sie ident bleiben müssen).

Bei einem Pull-basierten Ansatz, fordert ein Server oder ein Client einen anderen Server auf, dass er ihm alle aktuellen vorhandenen Aktualisierungen zuschickt. Pull-basierte Protokolle werden auch als clientbasierte Protokolle bezeichnet. Diese Strategie wird oft bei Webcaches angewandt. Darin besteht zu beginn die Aufgabe, zu überprüfen, welche im Cache existierenden Dateien noch nicht auf dem aktuellsten Stand sind. Wenn ein Cache eine Anfrage bezüglich einer Datei erhält, fragt dieser zuerst beim ursprünglichen Webserver nach, ob die Datei noch aktuell ist, oder ob diese in der Zwischenzeit verändert wurde. Falls eine Änderung auftrat, wird diese zuerst in den Cache geladen und erst dann dem anfragenden Client zurückgegeben. Dieser Ansatz ist besonders effektiv, wenn das Verhältnis von Lesezugriffen zu Aktualisierungen sehr gering ist. Der größte Nachteil im Bezug zu Pull-basierten Ansätzen ist, dass im Falle eines ausbleibendem Cache-Treffers sich die Antwortzeit verlängert.

Der größte Unterschied zwischen dem Pull- und Push-Prinzip ist, dass die Clients beim Pull die Clients ständig überprüfen müssen, ob sich nicht etwas geändert hat. Beim Push müssen ständig alle Clients überwacht werden, damit diese im Falle einer Änderung sofort benachrichtig werden.

Des Weiteren existieren auch hybride Prinzipien, welche sowohl auf das Push-, als auch auf das Pull-Prinzip setzen. Diese Art wird „Leasen“ genannt und funktioniert folgendermaßen. Indem der Server den Clients zusichert, dass er diese für einen bestimmten Zeitrahmen mittels Push verständigt, wenn Aktualisierungen auftreten. Dabei ist diese Zeitspanne der Verständigung variabel Einstellbar. Auch nach Ablauf dieser, ist es möglich wieder so eine Zeitspanne anzufordern. Des Weiteren ist es auch möglich, dass Kriterien für Dateien welche sich in „Lease“ befinden einzubinden. Dabei wird die Push-Dauer für eine Datei, welche sich nur selten ändert höher sein, als eine die sich häufig ändert. Durch „Leases“ ist es dem Server auch möglich, sich selbst zu regulieren, wenn ihm auffällt, dass er langsam zu einer Überlastung gelangt. Dann kann er die Leasedauer verringern und dadurch die aufwändigen Push Benachrichtigungen vermindern.

[1]

3.2.3. Unicasting und Multicasting

Ob Unicasting oder Multicasting verwendet werden soll, ähnelt sehr der Entscheidung, ob Aktualisierungen mittels Push- oder Pull-Prinzip übermittelt werden sollen. Wenn ein Server der zu einem Datenspeicher gehört, seine Aktualisierung an N andere Server schickt, erfolgt dies mittels der Unicast-Kommunikation. Dies wird realisiert, indem er N einzelne Nachrichten an jeden Server verschickt. Dabei erhält jeder Server eine eigene Nachricht. Beim Multicasting regelt dies das zugrundeliegende Netzwerk, indem es die Nachricht effizient an mehrere Empfänger zustellt. Bei vielen Situationen ist es kostengünstiger, von vorhandenen Multicasting-Einrichtungen Gebrauch zu machen. Ein extremer Fall tritt ein, wenn sich alle Replikate innerhalb desselben LAN befinden. Dadurch ist es möglich, Broadcasting auf Hardwareebene durchzuführen. In diesem Fall ist das Broadcasting oder Multicasting nicht mit höheren Kosten verbunden, als eine einzelne Nachricht von A nach B. Aktualisierungen mittels Unicasting wären hierbei weniger effizient. Multicasting ist gut mit einem Push-basierten Ansatz zur Weiterleitung von Aktualisierungen verknüpfbar. Wenn beide Verfahren sorgfältig aufeinander abgestimmt sind, dann muss ein Server, der entschieden hat, seine Aktualisierungen an eine Reihe von Servern zu übertragen, diese nur an eine einzelne Multicasting Gruppe zum Versand zu verwenden. Im Gegensatz zu einem Pull-basierten Ansatz, bittet im Allgemeinen nur ein einziger Client oder Server um eine Aktualisierung seiner Kopie. [1] [3]

4. Konsistenzmodell

Was ist überhaupt ein Konsistenzmodell?

Ein Konsistenzmodell kann man mit einem Vertrag vergleichen, welchen die Prozesse mit dem Datenspeicher abschließen. Der Vertrag verspricht, dass der Speicher solange einwandfrei funktionieren wird, solange die Prozesse sich an die bestimmten Regeln halten. [10]

4.1. Datenzentrierte Konsistenzmodelle

Normalerweise wird bei einer Leseoperation, von einem Prozess erwartet, dass er den aktuellsten Wert zurückerhält, also das Resultat des letzten Schreibvorganges. Jedoch gestaltet sich dies als durchaus schwierig, da es keine globale Uhr gibt, mit der man festlegen kann, welcher Schreibvorgang der letzte beziehungsweise aktuellste ist.

Um dieses Problem zu beheben gibt es viele verschiedene Konsistenzmodelle.
Prinzipiell schränkt jedes Modell die Werte ein, welche eine Leseoperation auf einen Dateieintrag zurückgeben kann. Es liegt einem nah, dass die leichtesten Konzepte die sind, welche umfangreichere und strengere Einschränkungen haben als die eher schwierigeren Konzepte, welche lockere Einschränkungen haben und daher eher umständlicher zum Benutzen sind. So wie auch im echten Leben, sind die komplexeren Varianten meist die effektiveren. [1]

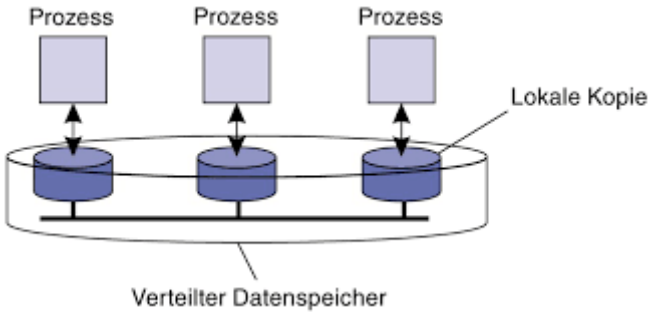

Abbildung 1; Der allgemeine Aufbau eines logischen Datenspeichers, physisch verteilt und repliziert für mehrere Prozesse [1]

4.2. Stufenlose Konsistenz

Vorweg, es gibt keine Optimallösung für die Datenreplikation. Konsistenzprobleme sind ständige Wegbegleiter, welche sich nicht mit einer Allgemeinlösung beheben lassen.
Es besteht ein Indirektes Verhältnis bezüglich Konsistenz und Performance. Wenn man die Konsistenzanforderungen lockert wird das System klarerweise schneller und wenn man sie strenger festlegt, leidet die Performance.

„Yu und Vahdat [11] wählen einen allgemeinen Ansatz, indem sie drei unabhängige Achsen zur Definition von Inkonsistenzen unterscheiden: Abweichung zwischen den Replikaten hinsichtlich der numerischen Werte, Abweichung zwischen den Replikaten hinsichtlich des Veralterungsgrades von Daten sowie Abweichung hinsichtlich der Reiheinfolge von Aktualsierungsoperationen.“ [1]

- Abweichung zwischen den Replikaten hinsichtlich der numerischen Werte
    - Es gibt zwei Möglichkeiten um die numerische Abweichung festzulegen.
    - Beispiel: Börsenkurs
      Bei der **absoluten numerischen Abweichung** wird angegeben, dass zwei Kopien sich nicht mehr als 0.2€ unterscheiden dürfen.
      Bei einer **relativen numerischen Abweichung** wird wiederum angegeben, dass der Unterschied zwischen zwei Kopien nicht größer als 0.5% sein darf.
- Abweichung zwischen den Replikaten hinsichtlich des Veralterungsgrades von Daten
    - *„Abweichungen im Veralterungsgrad beziehen sich auf den Zeitpunkt der letzten Aktualisierung eines Replikates. Einige Anwendungen können tolerieren, dass ein Replikat alte Daten liefert, solange sie nicht zu alt sind.*“ [1]
    - Beispiel: Wetterbericht
      Wetterberichte bleiben meistens über einen längeren Zeitraum akkurat. Das bedeutet, dass es relativ gleichgültig ist, ob der Wert einige Stunden alt ist, da er sich kaum bis geringfügig zum aktuellen Wert unterscheidet.
- Abweichung hinsichtlich der Reihenfolge von Aktualsierungsoperationen
    - Diese Aktualisierungen können so verstanden werden, dass sie auf eine lokale Kopie nur vorläufig angewendet werden, um dann auf die globale Zustimmung aller Replikate zu warten. Infolgedessen müssen einzelne Aktualisierungen unter Umständen zurückgenommen und in einer anderen Reihenfolge erneut angewendet werden, bevor sie dauerhaft werden können. [1]

4.3. Konsistenzeinheit (Consistency Unit)

Die Consistency Unit, welche als Conit abgekürzt wird, ist eine von Yu und Vahdat definierte Konsistenzeinheit, für die Definierung von Inkonsistenzen. Ein Conit gibt die Einheit an, in der die Konsistenz gemessen werden soll. [1]

4.4. Konsistente Anordnung von Operationen

Da bei der parallelen und verteilten Datenverarbeitung mehrere Prozesse Ressourcen gemeinsam zur selben Zeit verwenden, bemühte man sich die Semantik gleichzeitiger Zugriffe auf replizierte Ressourcen darzustellen. Aus dieser Bemühung heraus ist ein wichtiges Konsistenzmodell entstanden, welches weiterhin Verwendung findet. Im folgenden Abschnitt wird die sogenannte sequenzielle Konsistenz genauer erläutert. Außerdem wird mit der kausalen Konsistenz noch eine Variante dargelegt, welche eine schwächere Variante desselben Prinzips ist. [1]

4.4.1. Sequenzielle Konsistenz

„Die Resultate jeder beliebigen Ausführung sind dieselben wie in dem Fall, dass die von allen Prozessen am Datenspeicher vorgenommene (Lese- und Schreib-) Operationen in einer bestimmten sequenziellen Anordnung erfolgt sind und die Operationen jedes einzelnen Prozesses in dieser Sequenz in der von seinem Programm vorgegebenen Reihenfolge erscheinen.“ [1]

Das bedeutet so viel wie, dass jede Reihenfolge von Lese- und Schreiboperationen diverser Prozesse auf ein Datenelement zulässig ist, solange alle Prozesse, welche dieses Datenelement benutzen, die selbe Reihenfolge beobachten.
Hier muss beachtet werden, dass nichts über den Zeitlichen Ablauf gesagt wurde, daraus schließen wird, dass es sich hier nicht immer um die zeitliche Reihenfolge handelt.

Für ein besseres Verständnis folgen nun zwei Beispiele.
Damit man die folgenden Diagramme verstehen kann müssen noch ein paar Rahmenbedingungen erwähnt werden. Prinzipiell stellen die Diagramme dar, wie verschiedene Prozesse verschiedene Operation zu unterschiedlichen Zeiten durchführen. Die Zeit verläuft über die horizontale Achse von links nach rechts.
Das Symbol W(x)a bedeutet, dass eine Schreiboperation (W) auf ein Datenelement (x) durchgeführt wird und diese Variable wird einem Wert (a) gleichgestellt. Zudem gibt es auch noch einen Lesezugriff, welcher mit R symbolisiert wird. [1]

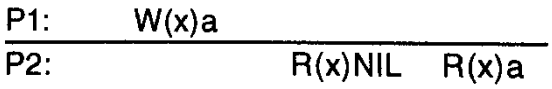

Abbildung 2; Verhaltensweise zweier Prozesse, die dassselbe Datenelemnt bearbeiten. Die horizontale Achse gibt den Zetverlauf an. [1]

In der Abbildung 2 führt P1 eine Schreiboperation auf das Datenelement x durch und befüllt sie mit dem Wert a.
Hierbei muss jedoch beachtet werden, dass P1 die Operation zunächst auf einer lokalen Kopie durchführt und erst im Anschluss an die anderen lokalen Kopien weiterleitet.
In diesem Beispiel liest P2 nun das Datenelement x aus und bekommt den Wert NIL zurück. Wir gehen davon aus, dass jedes Datenelement ursprünglich den Wert NIL hat. Einige Zeit später liest P2 nochmal das Datenelement x und bekommt den Wert a zurück.
Daraus schließen wir, dass es eine gewisse Zeit gedauert hat, bis das Datenelement bei der Kopie von P2 den aktualisiert wurde. Jedoch ist das völlig in Ordnung. [1]

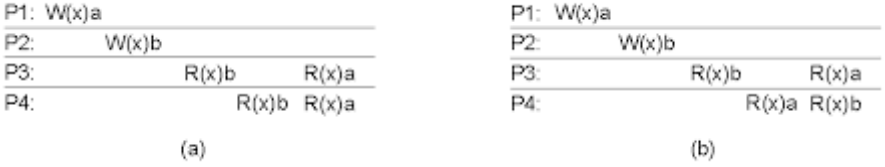

Abbildung 3; (a) ein sequenziell konsistenter Datenspeicher; (b) ein Datenspeicher, der nicht sequenziell konsistent ist [1]

In der Abbildung 3 werden zwei Diagramme gezeigt, a und b. Hier in diesem Fall ist a ein sequenziell konsistenter Datenspeicher und b ist nicht sequenziell konsistent.

In diesem Beispiel wird noch einmal verdeutlicht, dass Zeit keine Rolle spielt. Hier in dem „Diagramm a“, verarbeiten vier Prozesse (P1, P2, P3 und P4) das selbe Datenelement x. Zuerst führt P1 eine Schreiboperation durch mit dem Wert „a“ und danach führt P2 auch eine Schreiboperation durch mit dem Wert „b“. Nun lesen P3 und P4 zuerst den Wert „b“ und dann erst den Wert „a“.
Daraus folgt, dass die Schreiboperation von P2 schneller an die Replikate weitergeleitet wurde als die vom P1.
Beim „Diagramm b“ wird die sequenzielle Konsistenz verletzt, da nicht alle Prozesse dieselbe Reiheinfolge von Schreiboperationen sehen. Hier wird klarerweise die Reiheinfolge missachtet, da P3 zuerst „b“ und dann „a“ ausliest und P4 zuerst „a“ und dann „b“ ausliest. [1]

4.4.2. Kausale Konsistenz

„Bei der kausalen Konsistenz muss, im Gegensatz zur sequentiellen Konsistenz, nur für kausal voneinander abhängige Operationen eine globale Reihenfolge gefunden werden. Zwei Operationen sind kausal voneinander abhängig, wenn sie sich gegenseitig beeinflussen. Jede von demselben Prozess ausgeführte Operation ist potentiell kausal abhängig von den zuvor von diesem Prozess ausgeführten Operationen. Außerdem ist eine Leseoperation, die den von einem anderen Prozess geschriebenen Wert liest, von dieser Schreiboperation kausal abhängig.
Liest ein Prozess P2 beispielsweise zunächst das Objekt x, welches zuvor von einem anderen Prozess P1 geschrieben wurde, und schreibt anschließend y, so ist die Schreiboperation von P2 kausal von der Schreiboperation von P1 abhängig, da die Berechnung von y auf dem gelesenen Wert von x basieren könnte.
Wenn aber zwei Prozesse ohne zuvor ein Objekt gelesen zu haben ein Objekt verändern, so sind diese Operationen nicht kausal voneinander abhängig, sondern konkurrieren.
Konkurrierende Operationen dürfen bei kausaler Konsistenz auf verschiedenen Replikaten in unterschiedlichen Reihenfolgen verarbeitet werden.“ [10]

Für ein besseres Verständnis folgt nun ein Beispiel.
Bei dem folgendem Diagramm gilt dasselbe wie bei den Diagrammen der sequenziellen Konsistenz.

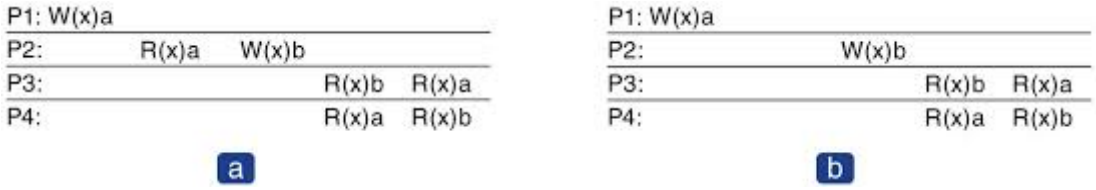

Abbildung 4; (a) ein Regelverstoß in einem kausal konsistenten Speicher; (b) eine korrekte Ereignissequenz in einem kausal konsistenten Speicher [1]

In diesem Beispiel der Abbildung 4(a) wird ein Regelverstoß in einem kausal konsistenten Speicher veranschaulicht. Hier hängt nämlich W(x)b vom P2 potenziell von W(x)a vom P1 ab, da b möglicherweise das Ergebnis einer Verarbeitung des R(x)a gelesenen Wertes ist. Da die beiden Schreibvorgänge in kausalem Bezug stehen, müssen alle Prozesse sie in derselben Reiheinfolge wahrnehmen. Da die Reiheinfolgen von P3 und P4 nicht übereinstimmen, ist Abbildung 4(a) inkorrekt.

In Abbildung 4(b) wurde der Leservorgang hingegen entfernt, sodass die Schreibvorgänge von P1 und P2 parallel sind. Die Abbildung 4(b) ist korrekt, da ein kausal Konsistenter Speicher nicht erfordert, dass bei parallelen Schreibvorgängen die globale Anordnung nicht gleich sein muss. [1]

Um einen kausalen Konsistenten Speicher umzusetzen, muss eine Methode implementiert werden, welche überwacht, welcher Prozess, welche Schreibvorgänge gesehen hat. [1]

4.5. Clientzentrierte Konsistenzmodelle

Bei verteilten Systemen ist es erforderlich, gleichzeitig gemeinsam verwendete Daten zu benutzen, jedoch muss stets die sequenzielle Konsistenz aufrecht erhalten bleiben. Aus Leistungsgründen kann sequenzielle Konsistenz möglicherweise jedoch nur garantiert werden, wenn Prozesse Synchronisationsmechanismen wie z.B. Transaktionen oder Sperren verwenden. [1]

4.5.1. Monotones Lesen und Schreiben

4.5.1.1. Lesen

Von einer Datenbank wird meistens erwartet, dass die Aktualität der Daten zunimmt. Es würde von der Annahme abweichen, wenn beim erneuten Lesen eines Datums ein älterer Wert, als der zuvor gelesene, zurückgegeben wird. [1]

Definition: Konsistenz für monotones Lesen
„Wenn ein Prozess den Wert eines Datenelements x liest, gibt jede anschließende Leseoperation dieses Prozesses auf x stets denselben oder einen aktuellen Wert zurück." [1]

Beispiel:
Nimmt man für dieses Beispiel eine replizierte E-Mail-Datenbank her, wobei ein Benutzer von einem Replikat eine Liste aller E-Mails auslesen kann. Während der Verwendung dieser Liste muss aufgrund der Mobilität des Anwenders das Replikat gewechselt werden. Wählt der Benutzer nun eine E-Mail zur Anzeige aus, so garantiert das monotone Lesen, dass die E-Mail tatsächlich auch auf dem neuen Replikat bereits empfangen wurde und der Benutzer nicht stattdessen eine Fehlermeldung erhält. [10]

4.5.1.2. Schreiben

Sehr oft ist es wichtig, dass die Reihenfolge der Schreiboperationen richtig an alle Kopien des Datenspeichers weitergeleitet werden. Falls dies der Fall ist wird dies mit der Eigenschaft „Konsistenz für monotones Schreiben" zum Ausdruck gebracht.

Definition: Konsistenz für monotones Schreiben
„Eine Schreiboperation eines Prozesses an einem Datenelement x wird abgeschlossen, bevor eine folgende Schreiboperation auf x durch denselben Prozess erfolgen kann" [1]

Beispiel:
In diesem Beispiel arbeitet ein Benutzer mit einem Textverarbeitungsprogram und sichert damit regelmäßig sein bearbeitetes Dokument. In diesem Fall garantiert das monotone Schreiben, dass auf jedem Replikat keine neuere Sicherung durch eine ältere überschrieben wird. In diesem Fall könnte das monotone Schreiben aber auch durch Schreiben nach dem Lesen realisiert werden, wenn die Textverarbeitung vor dem Erzeugen einer neuen Version das Dokument explizit liest. [10]

4.5.2. RYW & WFR – Konsistenz

4.5.2.1. Read Your Writes

Die „Read Your Writes"- Konsistenz (*also „lese dein Geschriebenes"*) ist eng verwandt mit den monotonen Lesevorgängen. Ein Datenspeicher verfügt über diese Eigenschaft, wenn folgenden Bedingung gilt.
„*Die Folge einer Schreiboperation eines Prozesses auf das Datenelement x wird für eine anschließende Leseoperation auf x durch denselben Prozess stets sichtbar sein.*" [1]

Die Abwesenheit dieser Konsistenz lässt sich manchmal erfahren, wenn man sein Passwort aktualisieren will. Nehmen wir zum Beispiel eine Digitale Bibliothek. Um diese im Web zu benutzen muss man sich mit Benutzername und Passwort anmelden. Bei der Aktualisierung seines Passwortes kann es dazu führen, dass man einige Minuten warten muss bis sein neues Passwort wirksam wird. Die Verzögerung kann dadurch verursacht werden, dass ein eigener Server für die Verwaltung der Passwörter zum Einsatz kommt und es einige Zeit in Anspruch nehmen kann, das Passwort an die unterschiedlichen Server weiterzuleiten, aus denen sich die Bibliothek zusammensetzt. [1]

4.5.2.2. Write Follow Reads

Write Follow Reads (WFR) Konsistenz, beschreibt eine Schreiboperation durch einen Prozess auf einem Datenelement x, die einer vorhergehenden Leseoperation für x folgt, findet garantiert auf demselben oder einem neueren Wert von x statt, der gelesen wurde. [10]

Kurz zusammengefasst: „*Jede nachfolgende Schreiboperation eines Prozesses auf ein Datenelement x erfolgt auf eine einer Kopie von x, die auf dem Stand des Wertes ist, der kurz vorher von dem Prozess gelesen wurde.*" [1]

Anwendungsfall: Wenn Benutzer einer Netzwerk-Newsgroup angehörig sind, wird denen gewährleistet, dass bevor sie das Posting einer Reaktion zu einem Artikel sehen können, sehen sie auf jeden Fall zuerst den Originalartikel. Sozusagen wird das Posting nicht auf ein Replikat gespeichert, solange nicht auch der Originalartikel auf dem Replikat vorhanden ist. [1]

5. Konsistenzprotokolle

Ein Konsistenzprotokoll ist im Prinzip eine Implementierung eines Konsistenzmodelles.

5.1. Urbildbasierte Protokolle

In der Praxis werden meistens Konsistenzmodelle verwendet die relativ leicht sind. Das gilt auch für die Modelle, welche für die Beschränkung von Abweichungen der Veralterung und in geringerem Maße auch die für die Beschränkung von numerischen Abweichungen sind.
Konsistenzmodelle werden ignoriert sobald sie zu schwer verständlich für die Anwendungsentwickler erscheinen, obwohl sie die Leistung verbessern könnten.

„*Bei der sequenziellen Konsistenz setzen sich urbildbasierte Protokolle durch. In diesen Protokollen verfügen alle Datenelement x im Datenspeicher über ein ihm zugeordnetes Urbild, das für die Koordination von Schreiboperationen auf x zuständig ist. Dabei kann unterschieden werden, ob das Urbild auf einem entfernten Server fixiert bleibt oder ob Schreiboperationen lokal erfolgen können, nachdem das Urbild zu dem Prozess verschoben wurde, von dem die Operation ausgeht.*" [1]

Nun folgen nähere Erläuterungen von den Klassen der Protokolle.

5.1.1. Protokolle für entfernte Schreibvorgänge (Primary Backup Protocoll)

Der Prozess, welcher eine Schreiboperation an dem Datenelement x vornehmen will, leitet die Operation an den für x primären Server weiter. Daraufhin aktualisiert dieser seine lokale Kopie von x und leitet sie anschließend an die Backupserver weiter. Die Backupserver führen auch eine Aktualisierung durch. Sobald sie mit der Aktualisierung fertig sind schicken sie eine Bestätigungsmeldung an den primären Server zurück. Sobald alle ihre lokalen Kopien aktualisiert haben, gibt der primäre Server eine Bestätigung an den Ausgangsprozess zurück. [1]

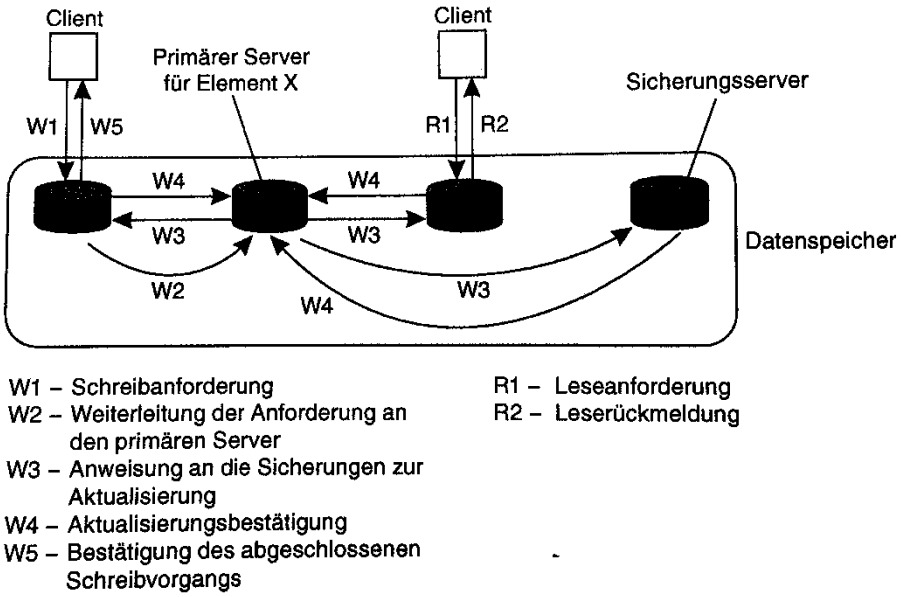

Abbildung 5; Prinzipielle Funktionsweise eines Urbildsicherungsprotokolls [1]

Hier gibt es jedoch ein Leistungsproblem, da es relativ lange dauern kann, dass eine Aktualisierungsbestätigung gesendet wird. Währenddessen kann der Prozess seine Arbeit nicht fortsetzen. „*Eine Aktualisierung wird als sperrende Operation implementiert.*“ [1]

Eine Alternative wäre ein nicht sperrender Ansatz. Jedoch ist bei dieser Option das Hauptproblem die Fehlertoleranz.

5.1.2. Protokolle für lokale Schreibvorgänge

Wie im vorherigen Fall macht ein Prozess das primäre Datenelement ausfindig. Jedoch wird in diesem Protokoll das Datenelement zu seinem eigenen Standort verschoben. Daraus ergibt sich der Vorteil, dass mehrere Schreiboperation lokal erfolgen können, während lesende Prozesse weiterhin auf ihre lokalen Kopien zugreifen können. [1]

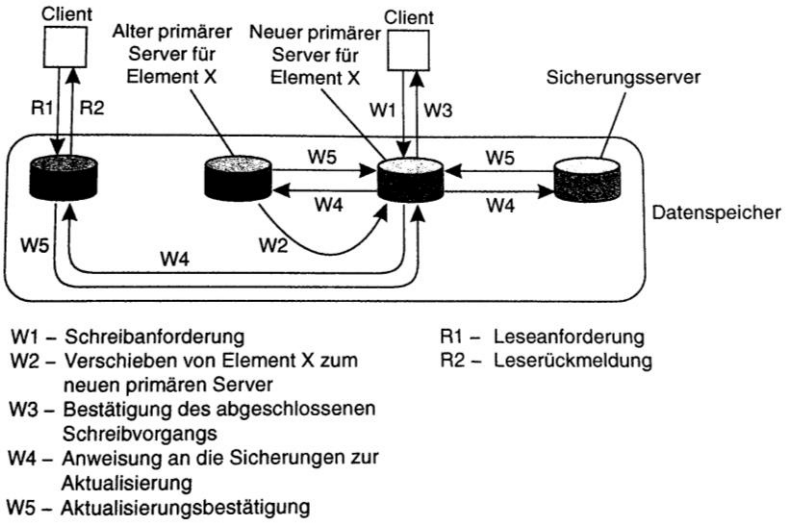

Abbildung 6; Ein Urbildsicherungsprotokoll, in dem das Urbild zu dem Prozess migriert, der eine Aktualisierung vornehmen will [1]

Dieses Protokoll kann auch auf tragbare Computer angewendet werden. Hier wird der Computer zum Primären Server für alle Datenelemente, die er aktualisieren will. Im getrennten Zustand können diese weiterhin bearbeitet werden, während andere Prozesse Leseoperationen aber keine Schreiboperationen ausüben können. Wenn die Verbindung wiederhergestellt wird, werden die Aktualisierungen am Urbild an die Backups weitergeleitet, damit wieder ein konsistenter Zustand herrscht. [1]

5.2. Protokoll für replizierte Schreibvorgänge

Bei dem Protokoll für replizierte Schreibvorgänge können Schreibvorgänge auf mehreren Replikaten anstatt nur auf einem einzigen ausgeführt werden, wie im Fall der urbildbasierten Replikate. Hier kann zwischen aktiver Replikation, bei der eine Operation an alle Replikate weitergeleitet wird, und Konsistenzprotokoll, welches auf mehrheitliche Abstimmung basiert, unterschieden werden. [1]

5.2.1. Aktive Replikation

Bei der aktiven Replikation hat jedes Replikat ein Prozess, welcher die Aktualsierungsoperationen vornimmt. Eine Schreiboperation wird dann nicht nur an ein Replikat geschickt, sondern an alle. Eine Schwierigkeit der aktiven Replikation liegt darin, dass die Operationen überall in derselben Reihenfolge ausgeführt werden müssen. Folglich wird ein auf einer absoluten Ordnung basierender Multicast-Mechanismus benötigt. Eine Möglichkeit den Mechanismus zu implementieren wäre ein zentraler Koordinator, oder auch Sequenzierer genannt, der eine absolute Reihenfolge festlegt. Die Operationen werden mit einer laufenden Nummer bestimmende Reihenfolge ausgeführt. [1]

5.2.2. Quorumgestützte Protokolle

Es gibt zwei Herrschaften, welche dieses Protokoll sehr stark geprägt haben. Der Herr Thomas (1979), welcher die Idee mit der Abstimmung hatte, und der Herr Gifford (1979), welcher die Idee verallgemeinert hat.
Der Grundgedanke dieses Protokolls ist, dass Clients erst die Erlaubnis mehrerer Server einholen müssen, bevor sie ein repliziertes Datenelement lesen oder schreiben dürfen.

Nun folgt ein Beispiel für ein besseres Verständnis.
Wir gehen von einem verteilten Dateisystem aus, bei dem eine Datei auf N Servern repliziert wird. Nun wird eine Regel aufgestellt, dass ein Client, welcher eine Aktualisierung durchführen will, muss bevor er die Operation durchführt, die Abstimmung von mindestens der Hälfte plus einem Server einholen. Sobald er die Berechtigung hat, wird die Datei geändert und die neue Datei mit einer neuen Versionsnummer versehen. Mithilfe dieser Nummer lässt sich die Dateiversion erkennen, die für alle replizierte Kopien identisch ist.
Für eine Leseoperation braucht der Client auch eine Berechtigung von der Hälfte aller Server plus einem. Stimmen dann alle Versionsnummern überein, muss es sich um die aktuellste Version handeln. [1]

Damit es eindeutig klar ist, dass diese Version immer zum Aktuellsten Wert führt wird noch ein weiteres Beispiel erläutert.

Es gibt fünf Server und ein Client stellt fest, dass drei von den fünf Servern die Version 8 haben. Dadurch ist es ausgeschlossen, dass die anderen beiden Server die Version 9 haben. Schließlich ist für eine erfolgreiche Aktualisierung von Version 8 auf Version 9 unabdingbar, dass drei Server ihr Zustimmen und nicht nur zwei. [1]

Gifford hat es tatsächlich geschafft, ein allgemeineres Konzept zu schaffen, welches nun genauer erläutert wird.
Ein Client, welcher eine Datei lesen will, von der N Replikate existieren, muss ein Lesequorum versammeln. Dieses Lesequorum ist eine willkürliche Zusammenstellung beliebig ausgewählter N_R Server, oder mehr.
Ähnlich ist zur Veränderung einer Datei ein Schreibquorum von mindestens N_W Servern nötig.
Die Werte von N_R und N_W unterliegen folgenden beiden Einschränkungen:

1. $N_R + N_W > N$
2. $N_W > N/2$

Einschränkung Nummer 1 dient zur Vermeidung von Lese-Schreib-Konflikten, während die zweite Schreib-Schreib-Konflikte verhindert. Erst wenn eine entsprechende Anzahl von Servern zugestimmt haben, kann eine Datei gelesen beziehungsweise geändert werden. [1]

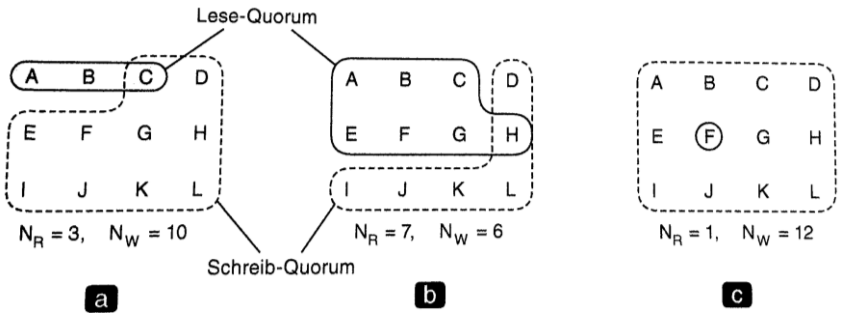

Abbildung 7; Drei Beispiele für den Abstimmungsalgorithmus: (a) eine korrekte Vorgabe für den Lese- und den Schreibvorgangssatz; (b) eine Wahl, die zu Schreib-Schreib-Konflikten führen kann; (c) eine korrekte Wahl, die als ROWA (Read-One, Write-All) bekannt ist [1]

In der Abbildung # (a) sehen wir den Algorithmus mit den Werten $N_R = 3$ und $N_W = 10$. Wenn wir davon ausgehen, dass das Schreibquorum die zehn Server von C bis L umfasst, bekommen all diese Server die neuste Version sowie die neuste Versionsnummer. Dadurch, dass das Schreibquorum die Server A bis C umfasst, hat es einen Server dabei, welcher auf dem neusten Stand ist. Mithilfe der Versionsnummer kann dann ermittelt werden, welcher Server am aktuellsten ist.
In der selben Abbildung # beim Beispiel (b) kann es zu einem Schreib-Schreib-Konflikt kommen da die Bedingung $N_W > N$ nicht eingehalten wurde. Nämlich würde ein Client eine Schreiboperation an den Servern {A, B, C, E, F, G} durchführen und ein andere eine auf den Servern {D, H, L, I, J, K, L}, dann ist es offensichtlich, dass es zu Schwierigkeiten kommt. Dadurch würden beide Operationen akzeptiert werden und es würde nicht erkannt werden, dass sie einander widersprechen.
In dem Beispiel der Abbildung # (c) kommt ein spezielles Verfahren zur Anwendung, welches im Allgemeinen als „ROWA Read-One, Write All“ bekannt ist.
Hier entspricht N_R dem Wert eins und N_W dem Wert aller Server. Mithilfe dieses Verfahrens wird eine sehr hohe Lesegeschwindigkeit garantiert, jedoch wird diese damit erkauft, dass Aktualisierungen alle Kopien beinhaltet müssen. [1]

Abbildungsverzeichnis

Literatur

[1] A. S. Tannenbaum und M. van Steen, Verteilte Systeme – Prinzipien und Paradigmen. Pearson Deutschland GmbH, November 2007, vol. 2.

[2] B. Charron-Bost, F. Pedone und A. S. (Eds.), Replication – Theory and Practice, G. Goos, J. Hartmanis, und J. van Leeuwen, Eds. Springer-Verlag Berlin Heidelberg, 2010.

[3] M. Moser. (2006, Mai) Platzierung von Replikaten in Verteilten Systemen. Humboldt Universität zu Berlin. [Online]. Verfügbar unter: opus4.kobv.de/opus4-zib/files/999/MoserDiplom.ps

[4] M. Wiesmann, F. Pedone, A. Schiper, Understanding Replication in Databases and Distributed Systems, Swiss Federal Institute of Technology, 2000.

[5] A. Schill und T. Springer, Verteilte Systeme - Grundlagen und Basistechnologien, Springer Vieweg Verlag, 2012.

[6] Göschka. (2009, Februar) Verteilte systeme. TU Wien. [Online]. Verfügbar unter: https://vowi.fsinf.at/images/4/46/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Fragenkatalog_v2_WS08.pdf

[7] Rahm E., Saake G. und Sattle K., Verteiltes und Paralleles Datenmanagement - Von verteilten Datenbanken zu Big Data und Cloud, Springer Vieweg Verlag, 2015.

[8] C. Redl. (2007) Verteilte systeme. [Online]. Verfügbar unter: http://web.student.tuwien.ac.at/~e0525250/uni/DSZusammenfassung.pdf

[9] Gunter Saake, Replikation – Uni Magdeburg [Online], Verfügbar unter: http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/tv/ws2015/slides/teil_8.pdf

[10] K. Hamann. (2009, Februar) Parallele Ausführung von Prozessen auf mobilen Geräten. Universität Hamburg. [Online]. Verfügbar unter: http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/518/Diplomarbeit.pdf

[11] Yu, H. und Vahdat, A., Design and Evaluation of Conit-Based Continuous Consistency Model for Replicated Services, ACM Trans. Comp. Syst., (20)3:239-282, 2002.

[12]R.Burger., (2014) Desaster Recovery und Replikation [Online], Verfügbar unter: http://burger-ag.de/replikation.whtml

[13]Oliver Haase, Replikate & Konsistenz II [Online], Verfügbar unter: http://www-home.htwg-konstanz.de/~haase/lehre/versy/slides/v8ReplikationKonsistenz2.pdf