SYSTEMTECHNIK

Dezentrale Systeme

MATURA 2016

STEFAN GEYER

Inhaltsverzeichnis

1 Cloud Computing und Internet of Things

1.1 Nachrichtenorientierte Middleware

1.1.1 Warteschlangensysteme

Ein Warteschlangensystem ist eine Middleware, die Kommunikation zwischen verschiedenen Teilnehmern ermöglicht. Solche Systeme bieten eine zwischenliegende Speicherkapazität an, in welcher Nachrichten persistiert werden können. Warteschlangensysteme werden üblicherweise verwendet, wenn die Nachrichtenübertragung Minuten statt Sekunden oder Millisekunden dauern darf.

Sendet ein Client eine Nachricht an einen Teilnehmer, wird diese in eine Warteschlange eingefügt. Die Nachrichten in der Warteschlange werden daraufhin zwischen mehreren Kommunikationsservern weitergeleitet und schlussendlich an das Ziel ausgeliefert. Jede Anwendung hat ihre eigene Warteschlange, an die andere Anwendungen Nachrichten senden können. Der Vorteil dieser Technologie ist, dass es egal ist, ob der Empfänger angeschaltet ist, oder nicht. Die Nachricht wird an den Empfänger zugestellt, sobald sie das nächste Mal verfügbar ist. Der Sender erhält demnach nur die Garantie, dass die Nachricht irgendwann in die Warteschlange des Empfängers eingefügt wird. Es ist allerdings unbekannt wann das sein wird, und ob der Empfänger die Nachricht liest.
Dadurch ist eine lose gekoppelte Kommunikation möglich. Anders gesagt, es ist nicht notwendig, dass der Empfänger immer aktiv ist, um mit diesem zu Kommunizieren. Ebenso ist es nicht notwendig, dass der Sender aktiv ist, wenn der Empfänger die Nachricht empfängt. Sobald eine Nachricht in der Warteschlange abgelegt wird, bleibt sie da, bis sie entfernt wird.

Prinzipiell ist es egal, welche Daten in den Nachrichten gesendet werden. Wichtig ist allerding, dass die Nachrichten korrekt adressiert sind. Normalerweise werden die Nachrichten systemweit über einen eindeutigen Namen identifiziert.

Das Warteschlangensystem bietet den Anwendungen eine Schnittstelle, über die diverse Aktionen durchgeführt werden kann. In den meisten Fällen ist sie daher sehr einfach gehalten. Es gibt die Kommandos PUT, GET, POLL und NOTIFY.
Mit dem PUT-Kommando sendet der Sender eine Nachricht an das System, welche an die Warteschlange des Empfängers angehängt wird.
Das GET-Kommando ist ein blockierender Aufruf, welcher so lange blockt, bis eine Nachricht in der Warteschlange ankommt (Der Prozess blockiert, solange die Warteschlange leer ist). Diese Nachricht wird ausgelesen, zurückgegeben und aus der Warteschlange entfernt.
Die nicht-blockierende Alternative zum GET-Befehl ist der POLL-Befehl. Der Befehl fragt die Warteschlange ab. Ist diese leer oder wird die gewünschte Nachricht nicht gefunden, fährt der Prozess einfach mit seiner Arbeit fort.
Mit dem NOTIFY-Befehl ist es möglich, dass der Prozess einen Callback-Handler benutzt, welcher immer dann aufgerufen wird, wenn die Warteschlange eine neue Nachricht erreicht. Im Callback kann ein beliebiges Verhalten implementiert werden, welches zum Beispiel die Nachricht ausliest oder löscht.

Jede Anwendung hat eine Quellwarteschlange, über welche die Nachrichten gesendet werden, und eine Zielwarteschlange, in welche eingehende Nachrichten gespeichert werden (Jede Nachricht muss eine Zielwarteschlange beinhalten, um den Empfänger zu identifizieren). Um diese Warteschlangen zu verwalten, ist meist ein Warteschlangenmanager vorhanden. Die Anwendung kommuniziert nur mit dem Manager, welcher die Warteschlangen im Hintergrund regelt.
In der Praxis sind die Warteschlangen eines Systems auf verschiedenen Rechnern (verschiedene Applikationen) verteilt. Daher ist es notwendig, dass von System eine Datenbank mit Verbindungen zwischen Warteschlangennamen und Netzwerkposition geführt wird.

Ein weiterer wichtiger Punkt bei Warteschlangensystemen sind Nachrichten-Broker. Da das Nachrichtensystem als zusammenhängendes Informationssystem arbeiten soll, ist es notwendig, das die Empfänger die Nachrichten verstehen die sie erhalten. Das bedeutet, dass das gesendete Format ein Format sein muss, dass der Empfänger versteht.
Problematisch wird es, wenn eine neue Anwendung zum System hinzukommt, die mit einem gesonderten Nachrichtenformat arbeitet und alle möglichen Empfänger eingestellt werden müssen, um das Format zu erzeugen. Ein allgemeines Nachrichtenformat wäre die logische Folge, allerdings macht das bei Warteschlangensystemen wenig Sinn, da die verschiedenen Prozesse in dem System arbeiten. Wenn die Prozesse unterschiedlich arbeiten, lässt sich schwer ein Format finden, welches sich für alle gut eignet.
Daher kommen die Nachrichten-Broker ins Spiel, deren Aufgabe es ist die Nachricht vom Senderformat in ein Format umzuwandeln, dass der Empfänger versteht. Weiters stellt der Broker ein Publish/Subscribe-Format zur Verfügung. Es können Themen (Topics) erstellt werden, die von Anwendungen abonniert werden können. Wird eine Nachricht zu dem Thema veröffentlich, wird die Nachricht an alle Abonnenten weitergeleitet. [1]

1.1.2 JMS – Java Message Service

Mit dem Java Message Service wird eine Message Oriented Middleware in Java umgesetzt. Eine konkrete Implementierung davon ist Apache ActiveMQ, welche einerseits eine Java Bibliothek ist, und andererseits aber auch einen Message Broker zur Verfügung stellt. Der folgende Code wird benötigt, um die Verbindung herzustellen.

```
// Create a ConnectionFactory
ActiveMQConnectionFactory connectionFactory = new
      ActiveMQConnectionFactory("vm://localhost");

// Create a Connection
Connection connection = connectionFactory.createConnection();
connection.start();

// Create a Session
Session session = connection.createSession(false,
      Session.AUTO_ACKNOWLEDGE);
```

Erstellt wird eine Verbindung mit der ConnectionFactory der Bibliothek. Mithilfe der Factory wird eine Connection erstellt, welche auch direkt gestartet wird. Danach wird eine Session erstellt. Davon ausgehend, können nun die einzelnen Komponenten initialisiert werden.

Um einen Producer für eine Queue zu erstellen, ist der folgende Code erforderlich:

```
// Create the destination (Topic or Queue)
Destination destination = session.createQueue("TestQueue");

// Create a MessageProducer from the Session to the Topic or Queue
MessageProducer producer = session.createProducer(destination);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

// Create a messages
String text = "Hello world!";
TextMessage message = session.createTextMessage(text);

// Tell the producer to send the message
producer.send(message);

// Clean up
session.close();
connection.close();
```

Die Klasse Destination legt die Queue an. Mit dem MessageProducer ist es möglich Nachrichten über die Queue an eine andere Warteschlange zu senden. Daraufhin wird über den Producer eine TextMessage gesendet.

Für den entsprechenden Consumer ist dieser Code notwendig:

```
// Create the destination (Topic or Queue)
Destination destination = session.createQueue("TestQueue");

// Create a MessageConsumer from the Session to the Topic or Queue
MessageConsumer consumer = session.createConsumer(destination);

// Wait for a message
```

```
Message message = consumer.receive(1000);

// Print the result
System.out.println("Received: " + ((TextMessage) message).getText());

// Clean up
consumer.close();
session.close();
connection.close();
```

Die Queue muss denselben Namen wie die, des Senders haben. Danach wird ein entsprechender Consumer erstellt. Es wird auf eine Nachricht gewartet und das Ergebnis ausgegeben.

Das Initialisieren eines Producers für ein Topic funktioniert wie folgt:

```
Topic topic = session.createTopic("TestTopic");
MessageProducer producer = session.createProducer(topic);
```

Es werden anstatt einer Queue ein Topic und der entsprechende MessageProducer erstellt.

Natürlich gibt es auch dafür einen entsprechenden Consumer:

```
Topic topic = session.createTopic("TestTopic");
MessageConsumer consumer = session.createConsumer(topic);

// The MessageListener handles incoming Messages
MessageListener listener = new MessageListener() {
    public void onMessage(Message message) {
        System.out.println(((TextMessage) message).getText());
    }
};

consumer.setMessageListener(listener);
```

Es muss ein Topic definiert werden, das denselben Namen trägt wie jenes des Producers. Durch den MessageListener kann festgelegt werden, was mit eingehenden Nachrichten passiert.

[2]

2 Automatisierung, Regelung und Steuerung

2.1 Systemarchitekturmodell der Service-Oriented-Architecture

Die Service-Oriented-Architecture (SOA) wird oft nicht als Architektur, sondern als Paradigma (Denkmuster) bezeichnet, dessen Ziel es ist, die Flexibilität großer Systeme zu erhöhen. [3]

2.1.1 Konzepte

SOA setzt für einen vollen Funktionsumfang drei Eigenschaften voraus:

2.1.1.1 Services

„In der Softwareentwicklung geht es immer um Abstraktion. Es müssen Anforderungen des Kunden so abstrahiert werden, dass nur noch die technisch relevanten Aspekte übrig bleiben. Bei dieser Abstraktion können jedoch auch wichtige Details verloren gehen.“ - [3]

In der SOA werden die Anforderungen auf die fachlichen Aspekte eines Problems abstrahiert. Ein Service ist eine technische Repräsentation einer fachlichen Funktionalität und trägt zur Lösung eines Problems bei. [3]

2.1.1.2 Interoperabilität

Bei verteilten Systemen werden Verbindungen hergestellt, über welche miteinander kommuniziert wird. Ist das schnell, einfach und problemlos möglich, spricht man von einer „hohen Interoperabilität“. Dieses Konzept ist allerdings nur einen Basis, durch die weitere Prinzipien geschaffen werden, die es technisch einfach ermöglichen fachliche Funktionalität über verschiedene Systeme abzuwickeln. [3]

2.1.1.3 Lose Kopplung

Ein wichtiger Punkt ist, dass Systeme möglichst unabhängig voneinander sein sollten. Manchmal ist es zwar erwünscht, dass zwei Systeme so eng koppeln, dass sie quasi verschmelzen, doch in den meisten Fällen ist es wünschenswert, dass ein System so selbstständig wie möglich agiert.
Im Allgemeinen brauchen große Systeme die folgenden drei Eigenschaften:

- Flexibilität
- Skalierbarkeit
- Fehlertoleranz

Lose Kopplung ist ein Konzept um Abhängigkeiten zu minimieren. Je geringer die Abhängigkeiten zwischen Systemen sind, desto geringer sind die Auswirkungen von Veränderungen oder Fehlern. Außerdem führt lose Kopplung auch zu einer besseren Skalierbarkeit. Große Systeme funktionieren nur dann, wenn die eigentliche Business Logik so dezentral wie möglich umgesetzt wird. Zentralismus soll also möglichst vermieden werden. [3]

2.1.2 Umgesetzte Architekturstile

Mit dem Enterprise-Service-Bus, wird ein Ereignisbasierter Architekturstil umgesetzt.

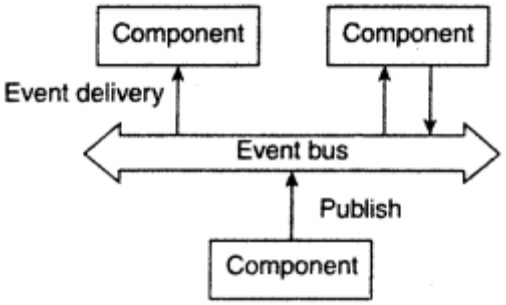

Abbildung 1: Ereignisbasierter Architekturstil

Der ESB wird in Punkt 4.1.2 genauer beschrieben.

3 Security, Safety, Availability

3.1 Algorithmen der Lastenverteilung

Es gibt viele verschiedene Methoden eine Serverseitige Lastenverteilung durchzuführen. Die bekanntesten sind dabei Round-Robin, Weighted Distribution, Least Connections und Response Time.

3.1.1 Round-Robin

Die verschiedenen Verbindungen werden vom Lastenverteiler der Reihe nach verteilt. Bei einem Beispiel mit drei Servern und vier offenen Verbindungen erhält Server 1 zwei Verbindungen und Server 2 und 3 jeweils eine. Dieses Prinzip ist die einfachste Lastverteilungs-Methode, jedoch kann keine gleichmäßige Verteilung garantiert werden, da jede Verbindung durchaus unterschiedlich lang geöffnet bleiben kann. Daraus folgt, dass die Server unterschiedlich viele Verbindungen nebenläufig abarbeiten müssen, was zu einer schlechten Lastenverteilung führen kann.

Da der Round-Robin Algorithmus sehr simpel ist, verbraucht er nur wenige Ressourcen des Lastenverteilers. Deshalb kommt er meistens in Situationen zum Einsatz, wo die Lastenverteilung sehr rechenaufwändig ist. Wenn man ein Beispiel mit 1000 Servern betrachtet, ist es für den Lastenverteiler sehr aufwändig festzustellen, welcher der Server für die nächste Verbindung am besten geeignet ist.

Ein weiterer Einsatzbereich von Round-Robin sind Situationen in denen jeder Request oder jede Verbindung ungefähr gleich viele Ressourcen beansprucht und in etwa gleich lange offen bleibt. [4]

3.1.2 Weighted Distribution

Oft kommt es vor, dass nicht alle Server in einem System hardwaretechnisch gleichstark ausgestattet sind. Mit dem Weighted-Distribution-Algorithmus kann die Leistung einzelner Server in Relation zueinander gestellt werden. Der Serveradministrator legt am Lastenverteiler eine Gewichtung für alle

Server fest (zum Beispiel zwischen 1 und 5). In einem Beispiel, in dem Server 1 eine Gewichtung von 5 und Server 2 eine Gewichtung von 1 hat, bekommt Server 5 demnach auch die fünffache Menge an Anfragen zugewiesen.

Oft wird dieser Algorithmus mit anderen Methoden wie Round-Robin oder Least-Connections kombiniert. Unter der Annahme, dass bei dem oben genannten Beispiel ein Weighted-Least-Connections Algorithmus verwendet wird, würde der Lastenverteiler 5 offene Verbindungen am stärkeren Server wie eine offene Verbindung am schwächeren Server behandeln. Wird im Beispiel Weighted-Distribution mit Round-Robin kombiniert, sendet der Lastenverteiler 5 Requests an den stärkeren Server für jeden Request den er an den schwächeren Server sendet.

Unter Verwendung des Weighted-Distribution Algorithmus ist es auch problemlos möglich neue Server zu einer Serverfarm mit etwas älteren Servern hinzuzufügen. Allerdings ist es nicht unbedingt einfach das relative Gewicht, dass jedem Server zugewiesen wird, festzustellen. Einerseits fließen Hardwarekomponenten wie Prozessorgeschwindigkeit, die Größe des Arbeitsspeichers oder die Anzahl an Netzwerkkarten, aber andererseits auch die Art der Applikation in die Berechnung ein. Eine Anwendung die grafische Berechnungen voraussetzt (z.B. CGI-Script) ist sehr rechenintensiv während das Bereitstellen von statischem Content das I/O-Subsystem auslastet. [4]

3.1.3 Least Connections

Wie der Name schon sagt wird bei der Least-Connections-Methode der neue Request an den Server weitergeleitet, der die wenigsten offenen Verbindungen besitzt. Um diese Methode umsetzen zu können, ist es notwendig, dass der Lastenverteiler darüber informiert bleibt wie viele Verbindungen auf jedem Server offen sind. Diese Methode ist sehr beliebt und wird in vielen Anwendungsfällen benutzt wie zum Beilspiel im Web oder für DNS. [4]

3.1.4 Response Time

Durch die Reaktionszeit einer Anwendung kann auf deren Performance geschlossen werden. Dieses Konzept wird oft im Web angewandt: Viele Webseiten werden danach bewertet, wie schnell sie Anfragen beantworten und den Content bereitstellen können. Nachdem die Response-Time oft eine wichtige Rolle spielt, wird sie auch für die Lastenverteilung eingesetzt. Es stellt sich nun die Frage wie man diese Zeit am einfachsten messen kann. Ist die Applikation eine Web-Applikation welche das HTTP-Protokoll verwendet, so kann die Zeit zwischen dem Senden des Requests und dem Eintreffen der Response gemessen werden. Bei einer Anwendung, die über das TCP-Protokoll kommuniziert, wird die Zeit zwischen dem Senden einer TCP SYN (Synchronize) Nachricht und dem Empfangen der entsprechenden TCP SYN ACK (Synchronize Acknowledge) Nachricht gemessen. [4]

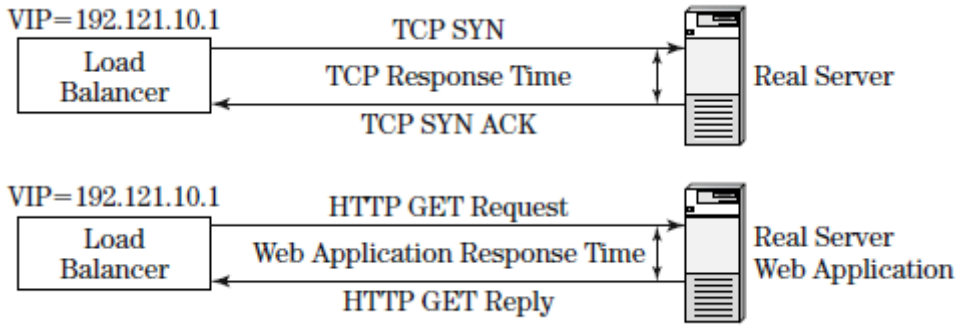

Abbildung 2: Messung der Response-Time [4]

Damit eine gute Lastenverteilung durchgeführt werden kann, muss die Response-Time über einen längeren Zeitraum gemessen werden, wobei die Messungen immer sehr aktuell sein sollten. Das ist notwendig, da die Zeiten von vor einer Stunde nicht mehr den aktuellen Zeiten entsprechen müssen. Allerdings haben viele Lastenverteiler die Funktion, basierend auf vergangenen und aktuellen Messungen, eine Vorhersage zu treffen welcher Server als Nächstes am Schnellsten antworten wird.

Da diese Methode sehr aufwändig ist, wird sie oft nur in Kombination mit einem anderen Algorithmus verwendet. [5]

3.2 Session-Persistence und das Megaproxy Problem

Das Megaproxy Problem ist ein Problem das bei der Session-Persistence von Lastenverteilern auftritt.

3.2.1 Session-Persistence Kurzfassung

Session-Persistence beschreibt den Vorgang der Identifikation eines Nutzers durch die IP-Adresse. Dadurch kann der Lastenverteiler dem Nutzer immer denselben Server zuteilen. Warum das notwendig ist kann anhand eines e-Commerce Einkaufswagen erklärt werden: Ein Nutzer möchte auf der Webseite etwas kaufen und wird vom Lastenverteiler auf einen Server weitergeleitet. Dort legt der Nutzer ein Produkt in den Einkaufswagen. Danach möchte der Nutzer noch ein weiteres Produkt kaufen und klickt auf den Link des Produktes. Der Lastenverteiler weist dem Nutzer diesmal allerdings einen anderen Server zu (z.B. wegen Round-Robin). Der Nutzer legt auch dieses Produkt in den Einkaufswagen und klickt wieder auf einen Link um den Einkaufswagen anzusehen und die Bezahlung der Produkte durchzuführen. Auch hier wird der Nutzer vom Lastenverteiler auf einen neuen Server umgeleitet. Auf diesem Server wurde das Legen in den Einkaufswagen nicht durchgeführt und daher findet der Nutzer seine gewählten Produkte nicht vor. [6]

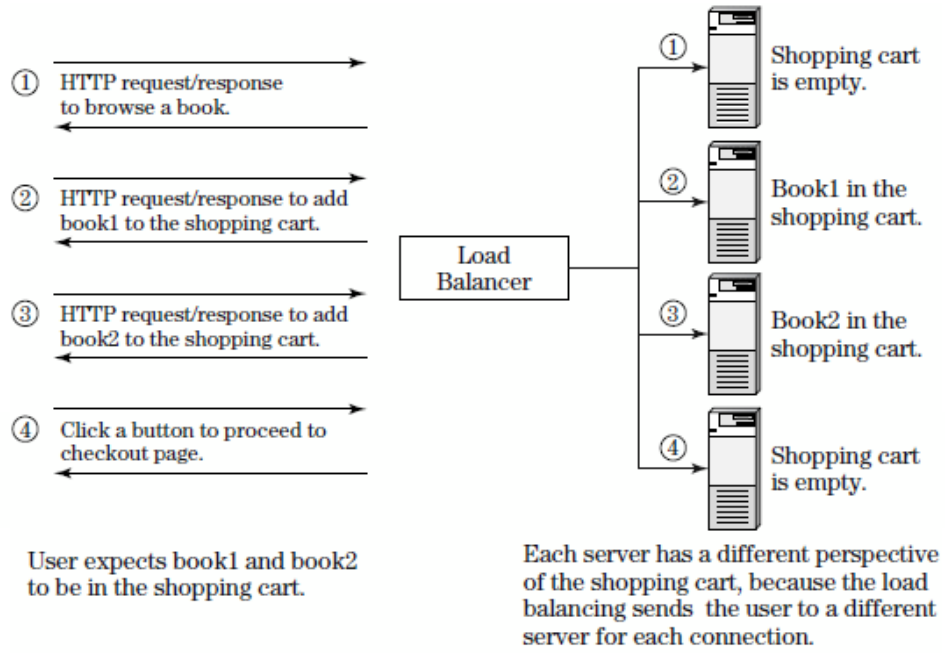

Abbildung 3: Grafische Darstellung des Megaproxy-Problems [6]

Wird der Client durch seine IP-Adresse identifiziert, kann der Lastenverteiler den Client immer auf den Server weiterleiten, auf den er auch die vorherigen Aktionen getätigt hat. Das hat zur Folge, dass der Nutzer immer alle gewählten Produkte im Einkaufswagen vorfindet. [3]

3.2.2 Das Megaproxy-Problem

Das Megaproxy-Problem tritt auf, wenn die IP-Adresse des Clients nicht eindeutig ermittelbar ist. Verursacht wird dieser Zustand dadurch dass der Client einen Proxy-Server verwendet, welcher alle Requests mit seiner eigenen IP weiterleitet. Proxys werden oft in Firmen oder von ISP verwendet, um die Identität der Nutzer zu verschleiern. Daraus ergeben sich zwei Probleme: Der Lastenverteiler hält alle Clients die diesen Proxy verwenden für dasselbe Endgerät und leitet diese Aufgrund der Session-

Persistence auf denselben Server weiter. Außerdem kann es sein dass ein Netzwerk mehrere Proxy-Server hat (daher der Name Megaproxy), welche vom Lastenverteiler auf verschiedene Server weitergeleitet werden. Um diese Probleme zu vermeiden darf die Identifikation nicht mehr auf IP-Basis passieren.

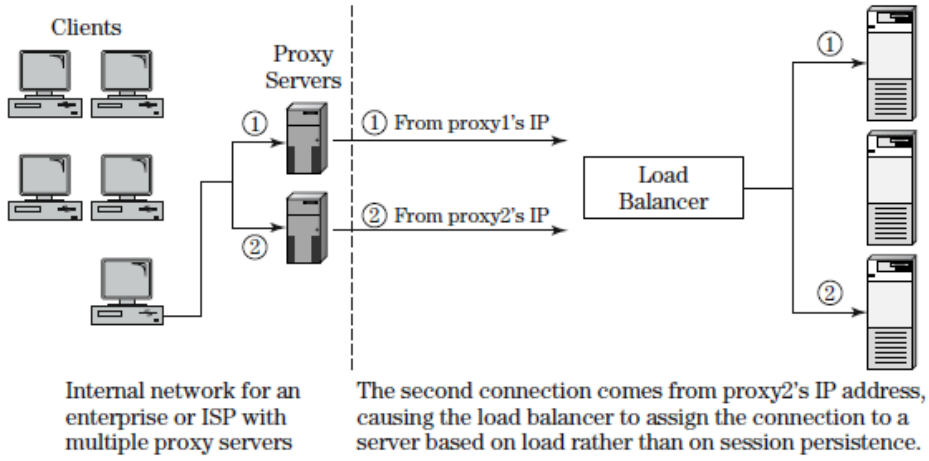

Abbildung 4: Das Problem mit mehreren Proxy-Servern [7]

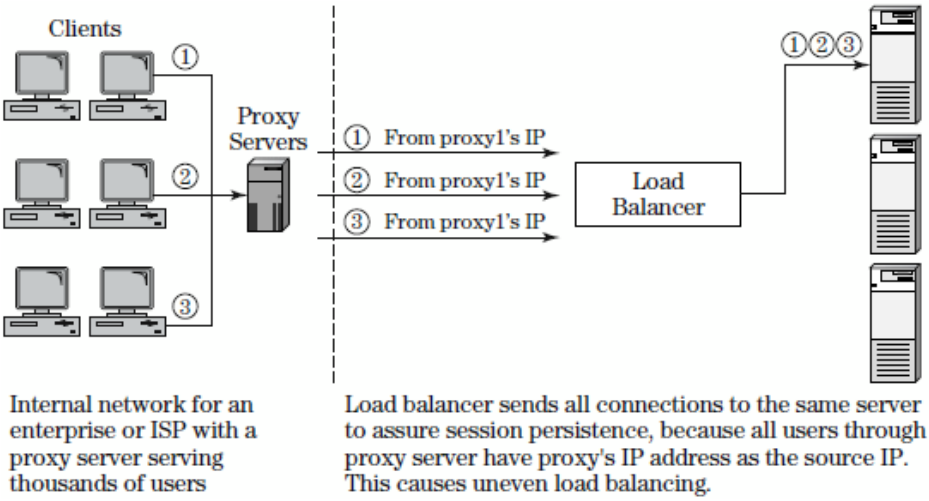

Abbildung 5: Der Lastenverteiler hält die Clients hinter dem Proxy für ein Endgerät [7]

Eine Lösung für das Problem bringt die Session-Persistence-Methode „Virtual Source“ mit sich. Damit ist es möglich die IP-Adresse aller Proxy-Server zu identifizieren und zu gruppieren. Die IP-Adressen aller Proxys werden zu einer virtuellen IP zusammengefasst. Dadurch ist es dem Lastenverteiler möglich weiterhin eine Session-Persistence zu garantieren.
Die einzelnen Clients können durch das Senden von zusätzlichen Daten (z.B. Cookies mit einer ID) identifiziert werden. [7]

3.3 High Performance Computing und Hochverfügbarkeit

3.3.1 High Performance Computing

Unter High Performance Computing – kurz HPC – versteht man die Nutzung von Parallelverarbeitung um komplexe Anwendungen effizient und schnell abarbeiten zu können. Diese Bezeichnung wird für Systeme verwendet, die im Bereich von über einem Teraflop (1012 Flops) arbeiten. Oft wird der Begriff HPC mit dem Begriff Supercomputer verglichen, allerdings ist ein Supercomputer ein System, welches mit der höchst möglichen Operationsrate im Bereich der Computer arbeitet. Manche Supercomputer können bereits mit mehr als einem Petaflop (1015 Flops) arbeiten.

HPC wird oft für wissenschaftliche Forschungen und von akademischen Institutionen eingesetzt. Außerdem setzten manche Regierungen HPC für komplexe Anwendungen oder Berechnungen ein. Des Weiteren ist es wahrscheinlich, dass sich Firmen für die starke Prozessorleistung und -geschwindigkeit interessieren. Mögliche Anwendungsfelder wären dabei die Transaktionsverarbeitung und das Data-Warehouse. [8]

3.3.2 Hochverfügbarkeit

Unter Verfügbarkeit versteht man ein System, dass ein Service bis zu einem speziellen Grad bereitstellen kann. In der Welt der Informatik versteht man unter Verfügbarkeit die Zeitspanne in der ein Serviceverfügbar ist (z.B. 16 Stunden pro Tag, Sechs Tage pro Woche). Jeder Verlust des Services, egal ob geplant oder nicht, wird als Ausfall bezeichnet. Die Dauer eines solchen Ausfalls wird als Downtime bezeichnet und in Minuten oder Stunden gemessen.

Ein System kann als hochverfügbar bezeichnet werden, wenn es darauf konzipiert ist, eine geringe Fehleranzahl und damit verbunden eine geringe Downtime hat.
Wir Menschen erwarten uns hochverfügbare Systeme, wenn unser Leben, unsere Gesundheit und Wohlbefinden davon abhängen. Ein Beispiel dafür ist die Energieversorgung. Jeder Ausfall ist inakzeptabel, da viele unserer Geräte, wie beispielsweise die Beleuchtung oder der Kühlschrank, davon abhängen.

High-Available-Computing nutzt Computer, welche darauf programmiert sind so wenig geplante und ungeplante Downtime wie möglich zu haben, um ein Service bereitzustellen. Konkret könnte das bereitgestellte Service eine Datenbank sein. Wenn beispielsweise eine Bank die Datenbank verwendet um Kontostände zu speichern und diese bei einem Geldtransfer anpassen möchte, muss die Datenbank unbedingt jederzeit, in der Nacht und am Tag und an jedem Tag der Woche, verfügbar sein. [9]

4 Authentication, Authorization, Accounting

4.1 Die Service-Oriented-Architecture

4.1.1 SOA-Triangle

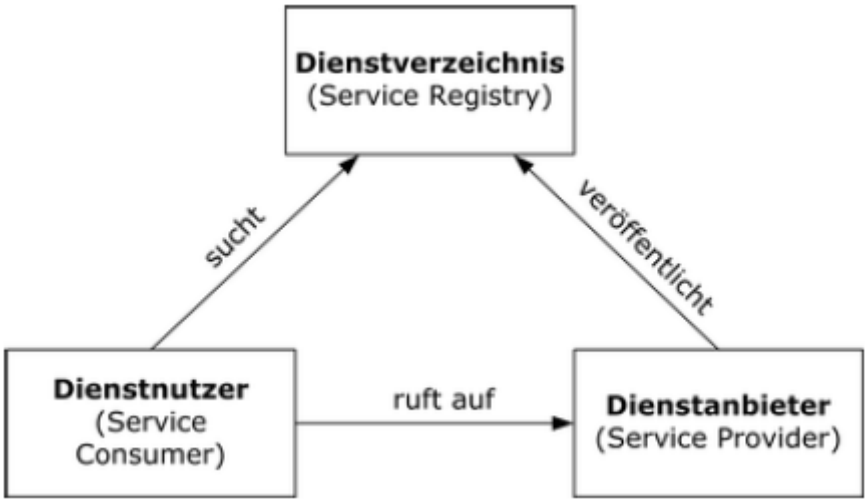

Abbildung 6: Abbildung des SOA-Triangle

4.1.1.1 Service-Consumer

Der Service-Consumer ist der Teil des SOA-Triangles, der das bereitgestellte Service aufruft und nutzt. [3]

4.1.1.2 Service-Provider

Der Service-Provider ist jener Teil des SOA-Triangle der die Business Logik eines Services so implementiert, dass es von anderen Systemen nutzbar ist. Um das zu vereinfachen gibt es den WSDL-Standard (Web Service Definition Language), womit das Service grob beschrieben wird. Die folgenden drei Eigenschaften werden jedenfalls beschrieben:

- Signatur (Name und Parameter)
- Binding (Protokoll für den Aufruf)
- Location (Adresse der Applikation)

[3]

4.1.1.3 Service-Registry

Die Registry ist eine Übersicht mit Informationen über die Verfügbaren Services. Weiters ist es durch die Registry möglich mit den Services zu kommunizieren. Das Ziel ist es schnelle und zuverlässige Kommunikation möglichst automatisiert zu bieten.
Um die Verwaltung von Services einfacher zu machen, gibt es den Standard UDDI zur Verwaltung von Services. Er beschreibt wie man als Provider neue Services registriert und als Customer bestehende Services findet. [3]

4.1.1.4 Ablauf

1. Der Service Provider veröffentlicht ein neues Service indem er es bei der Registry registriert.
2. Möchte ein Consumer auf ein Service zugreifen, fragt er bei der Registry nach, unter welcher Adresse das Service erreichbar ist.
3. Die Registry antwortet mit der Adresse und der Consumer kann direkt mit dem Provider kommunizieren.

4.1.2 Der Enterprise-Service-Bus

Der ESB ist ein wichtiger Bestandteil der Service-Oriented-Architecture. Er sorgt dafür, dass Services einfach systemübergreifend aufgerufen werden können. Der ESB ist für die Kommunikation zwischen Nutzern und Service-Providern verantwortlich. Konkret bietet der ESB den Nutzern Userinterfaces an, und ermöglicht ihnen so den Zugriff auf Applikationen und Services.

4.1.2.1 Aufgaben

Es ist nicht genau definiert, was ein ESB können muss, aber einige Features sind eigentlich immer vorhanden. Daher kann man sie als Kernaufgaben bezeichnen.

- Herstellen der Konnektivität
  Der ESB stellt sicher, dass die Verbindung zwischen Provider und Consumer hergestellt wird.
- Interoperabilität
  Nicht alle Systeme verstehen dieselbe Programmiersprache. Daher müssen verschiedene Plattformen und Programmiersprachen integriert werden. Normalerweise wird dafür ein Zwischenformat wie SOAP definiert, auf welches dann alle APIs und Programmiersprachen gemappt werden.
- Routing von Anfragen
  Routing bedeutet die Weiterleitung eines Service-Aufrufes eines Nutzers an einen Provider und die entsprechende Rückmeldung dessen an den Kunden.

4.2 Der Representational State Transfer

Ein RESTful Service ist eine Schnittstelle, die über das HTTP-Protokoll mit anderen Komponenten kommuniziert. HTTP-Requests werden an eine URL gesendet. Über diese URL kann identifiziert werden, welche Funktion man damit aufruft. Die Antwort des Servers ist strukturiert und meist in einem Übertragungsstandard wie XML oder JSON definiert. Die Übertragungsstandards werden im Punkt 4.3 genauer beschrieben. RESTful Services werden oft zur Authentifizierung oder einfach zum Datenaustausch mit einem Server verwendet.

4.2.1 Kommunikation über das Web

Damit ein Server auf HTTP-Anfragen der Nutzer reagieren kann, ist ein Webserver notwendig. Generell wird ein RESTful Service in Form eines Mehrschichtenmodells umgesetzt. Das bedeutet, dass mindestens ein Anwendungsserver und, in den meisten Fällen, ein Datenbankserver zum Einsatz kommen. Die Anwendung für den Anwendungsserver kann beispielsweise in Java geschrieben werden. Dabei kann zum Beispiel Java EE (JAX-RS) oder das Spring-Framework verwendet werden.

4.2.2 HTTP als Übertragungsprotokoll

Wie bereits erwähnt wird bei REST das HTTP-Protokoll verwendet. Dieses bietet verschiedene Methoden. Jede Methode hat demnach eine andere Funktion im RESTful Service. Konkret wird das CREATE-READ-UPDATE-DELETE-Prinzip, kurz CRUD, relationaler Datenbanksysteme übernommen. Jeder CRUD-Operation wurde eine entsprechende HTTP-Methode zugewiesen.

POST ist das Äquivalent zu CREATE im CRUD Prinzip. Mit POST sollen somit neue Ressourcen angelegt werden. Als passenden Response-Code sollte man "201 Created" zurückgeben, falls die Aktion erfolgreich war. Die POST-Methode wird allerdings auch oft verwendet, um Aktionen durchzuführen, für die es keine eigenen Methoden gibt.

GET ist das Äquivalent zu READ im CRUD Prinzip. Mit GET sollen bestehende Informationen ausgelesen werden und es soll keine Änderung der Informationen erfolgen. Mittels GET-Parameter kann die Suche nach angegebenen Kriterien eingeschränkt werden. GET liefert bei mehrmaligen Aufrufen immer dasselbe Ergebnis (vorausgesetzt, die Daten haben sich nicht geändert).

PUT ist das Äquivalent zu UPDATE im CRUD Prinzip. Mit PUT sollen bestehende Ressourcen aktualisiert werden, falls diese existieren. Falls diese nicht existieren, wird in manchen Fällen auch eine neue Ressource angelegt. Meist wird allerdings ein entsprechender Fehlercode zurückgegeben.

DELETE ist das Äquivalent zu DELETE im CRUD Prinzip. Mit DELETE sollen bestehende Ressourcen gelöscht werden. Falls die Ressource nicht existiert, wird meist ein entsprechender Fehlercode zurückgegeben.

4.2.3 Ansprechen von konkreten Ressourcen durch URLs

Bei einem HTTP-Request muss neben der verwendeten Methode auch eine Ziel-URL definiert werden. Beim Erstellen eines RESTful Services sollte darauf geachtet werden, dass die URLs auch richtig definiert werden.
Eine Ressource benötigt zwei Basis URLs. Die erste repräsentiert die Collection, also die Gruppe, in der sich die Elemente befinden. Hierbei sollten Nomen im Plural als Gruppenname verwendet werden. Außerdem ist es wichtig sich auf eine konkrete Gruppe zu beziehen, (z.B. nicht "animals" sondern "dogs") da sich die Gruppennamen sonst eventuell mit zukünftigen Namen überschneiden. (z.B. eine Gruppe mit dem Namen "cats")
Die zweite identifiziert das anzusprechende Element. Die Identifizierung kann über einen Zahlenwert, ein Datum, einen Namen oder vieles anderes erfolgen. Wichtig ist nur, dass der Identifier eindeutig und verständlich ist. Das könnte beispielsweise wie folgt aussehen.

```
http://api.funnyanimalpics.com/dogs/47
```

Hierbei repräsentiert "dogs" die Gruppe und "47" das gewünschte Element.

4.3 Übertragungsstandards

4.3.1 JSON

Die JavaScript Object Notation ist ein Format zum Datenaustausch und ein Teilgebiet von JavaScript. Demnach besitzt JSON keine Features die in JavaScript nicht ohnehin schon vorhanden sind. Wie bereits erwähnt wird JSON oft zum Datenaustausch zwischen mehreren Systemen genutzt und kommt auch oft in REST-Schnittstellen zum Einsatz.
Beim Senden eines JSON Strings gibt es keine Grenzen in Bezug auf die Größe und Kapazität. Jedoch kann es vorkommen, dass ein Server, welcher den JSON Code empfängt, Limits verwendet, um eine zu hohe Rechenlast zu verhindern. [10]

4.3.1.1 Syntax

4.3.1.1.1 Objekte

Ein Objekt ist eine ungeordnete Liste mit Key/Value Paaren. Objekte werden mit "{" gestartet und enden mit "}". Key und Value werden durch einen Doppelpunkt getrennt. Jedes Paar wird durch ein Komma von einem anderen getrennt. [11]

4.3.1.1.2 Arrays

Ein Array ist eine geordnete Liste an Werten. Das können normale Werte, Objekte oder weitere ineinander verschachtelte Arrays sein. Arrays werden mit "[" gestartet und enden mit "]". [11]

4.3.1.1.3 Datentypen

JSON verfügt über Strings (in doppelten Anführungszeichen), Zahlenwerte, Objekte, Arrays, Wahrheitswerte und Null als Datentypen. [11]

4.3.1.1.4 Strings

Ein String kann aus keinem oder mehr Unicode Zeichen bestehen. Dieser muss an beiden Enden mit einem doppelten Anführungszeichen versehen sein. Wird das Zeichen "\" verwendet, so wird das darauffolgende Zeichen maskiert. Abbildung 4 enthält eine Liste der Zeichen, die maskiert werden können. [11]

4.3.1.1.5 Binärdaten

Da JSON ein textuelles Übertragungsformat ist, können Binärdaten oder BLOBs nicht ohne weiteres übertragen werden. Jedoch ist es mithilfe von Base64 möglich, eben solche Binärdaten in Text umzuwandeln, diesen zu senden und auf der Seite des Empfängers wieder in das Binärformat zu konvertieren. [11]

4.3.2 XML

Obwohl mittlerweile JSON das am meisten verwendete Repräsentationsformat ist, kann die Nutzung von XML dennoch einige Vorteile mit sich bringen. Einerseits kann es sein, dass ältere Systeme eher XML als JSON unterstützen. Andererseits bringt XML eine standardisiere Schemasprache und die Möglichkeit, ein Dokument gegen diese zu validieren, mit sich. Für XML wurde der Medientyp "application/xml" definiert. Dieser sagt allerdings nichts über das verwendete Vokabular aus, sondern nur, dass das XML Dokument syntaktisch korrekt ist. Für ein festgelegtes Vokabular können eigene Medientypen definiert werden. Diese können über ein "application/<name>+xml" erstellt werden. [12]

5 Desaster Recovery

Eine klassische Webanwendung besteht aus mehreren Schichten. Oft sind diese in den Applikationsserver, welcher einerseits die grafische Ausgabe aber andererseits auch die Geschäftslogik bearbeitet, und den Datenbankserver, welcher die benötigten Informationen persistiert, aufgeteilt.
Es kann durchaus passieren, dass einer oder mehrere Komponenten von Zeit zu Zeit ausfallen. Ist das der Fall muss selbstverständlich darauf reagiert werden. Im folgenden Abschnitt werden mögliche Lösungsansätze für einen Systemausfall beschrieben.

5.1 Ansatz für Aktiv-Passiv-Replikation

Für den Applikationsserver könnte beispielsweise eine Aktiv-Passiv-Replikation eingesetzt werden. Läuft der Server A, behandelt er alle Anfragen. Sollte er aber ausfallen, übernimmt Server B bis A wiederhergestellt ist. Der Datenzugriff stellt in diesem Fall auch kein Problem dar, da der Datenbankserver ausgelagert ist.
Da es bei diesem Ansatz nur notwendig ist auf einem System schreiben zu können, sollte für als Konsistenzprotokoll ein urbildbasierendes Protokoll verwendet werden. Da nur zwei Server vorhanden sind, könnte ein Protokoll für entfernte Schreibvorgänge zum Einsatz kommen. Konkret übernimmt Server A die Rolle des festgelegten Einzelservers (Primärserver), und leitet alle Änderungen an Server B weiter.

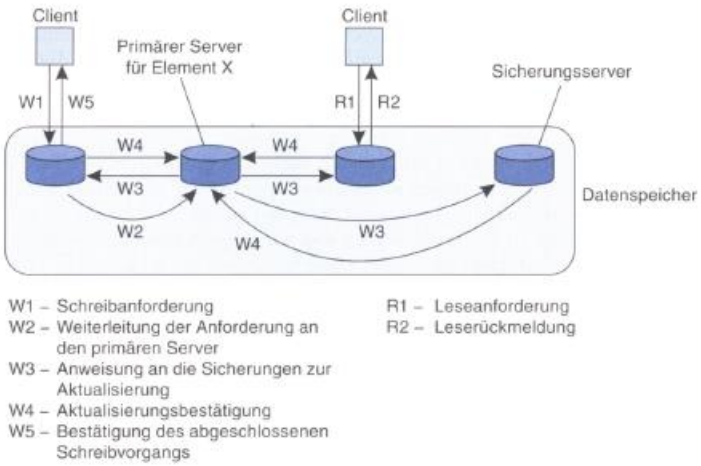

Abbildung 7: Ablauf eines Protokolls für entfernte Schreibvorgänge

5.2 Ansatz für inhaltsbewusste Replikate

Ein weiter Ansatz, der allerdings bereits im Setup zu berücksichtigen wäre ist, einige aktive (inhaltsbewusste) Replikate in einem Server-Cluster aufzusetzen. Die Anfragen würden dann, beispielsweise von einem Front-End an die jeweiligen Webserver weitergeleitet werden. Diese Methode würde zusätzlich einige angenehme Nebeneffekte wie die Möglichkeit eines Cacheeinsatzes mit sich bringen. Durch das schnellere auslesen wäre eine schnellere Antwort möglich. [13]

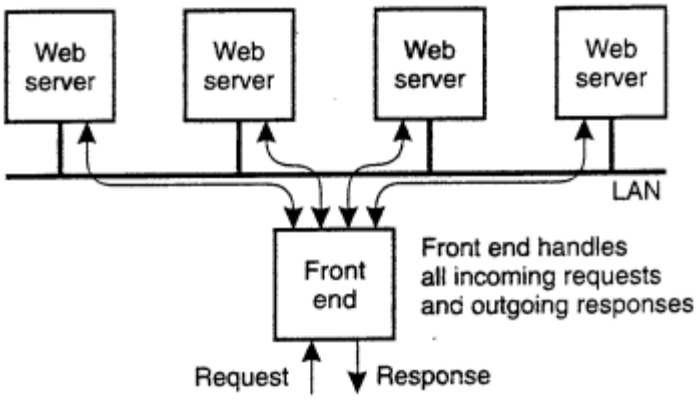

Abbildung 8: Servercluster im Zusammenspiel mit einem Frontend [13]

Als Konsistenzprotokoll sollte in diesem Fall ein Protokoll für einen replizieren Schreibvorgang gewählt werden, damit auch alle Replikate genutzt werden können. Eine Möglichkeit wäre es aktive Replikate zu verwendet. Dabei hat jedes Replikat ein Prozess, welcher die Aktualisierungs-Operationen 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, der eine absolute Reihenfolge festlegt. Die Operationen werden, im weiteren Sinne, in der festgelegten Reihenfolge ausgeführt.

5.3 Ansatz Datenbank

Einem Ausfall der Datenbank könnte durch eine verteilte Datenbank vorgebeugt werden. Konkret müsste bei der Allokation der Fragmente eine Allokation mit Replikation durchgeführt werden, um den Ausfall von einzelnen Stationen kompensieren zu können.

6 Algorithmen und Protokolle

6.1 Verteilter Commit

Für den verteilen Commit wird häufig ein Koordinator verwendet, welcher allen Teilnehmern mitteilt, ob der Commit ausgeführt werden soll oder nicht. Dieses Verfahren alleine wird oft als „1-Phase-Commit-Protokoll“ bezeichnet. Dieses Protokoll hat den Nachteil, dass die Teilnehmer dem Koordinator nicht mitteilen können, wenn die Operation nicht ausgeführt werden kann. Der Fall könnte zum Beispiel eintreten, wenn die aktuelle Transaktion von einer anderen blockiert wird.

Genau dieses Problem wird durch eine komplexere Version, das „2-Phase-Commit-Protokoll“, behoben. Allerdings hat auch dieses Protokoll einen Nachteil. Der Ausfall des Koordinators kann im 2PC nicht effizient behandelt werden kann. Daher wurde das „3-Phase-Commit-Protkoll“ für diesen Zweck entwickelt.

6.1.1 Zwei-Phasen-Commit-Protokoll

Behandelt wird eine verteilte Transaktion mit einer Anzahl an Teilnehmern, die auf unterschiedlichen Computern laufen. Unter der Annahme, dass keine Fehler auftreten, besteht das Protokoll aus zwei Phasen, die jeweils zwei Schritte umfassen. Die Abstimmungsphase (Schritte 1 & 2) und die Entscheidungsphase (Schritte 3 & 4).

1. Der Koordinator sendet eine VOTE_REQUEST-Nachricht an alle Teilnehmer
2. Die Teilnehmer empfangen die VOTE_REQUEST-Nachricht und antworten entweder mit einer VOTE_COMMIT-Nachricht, welche dem Koordinator mitteilt, dass der Teilnehmer lokal darauf vorbereitet ist, seinen Teil der Transaktion mit Commit zu bestätigen, oder anderenfalls mit einer VOTE_ABORT-Nachricht.
3. Der Koordinator erfasst alle Antworten der Teilnehmer. Wenn alle Teilnehmer für einen Commit gestimmt haben, sendet der Koordinator eine GLOBAL_COMMIT-Nachricht an die Teilnehmer. Wenn einer oder mehrere Teilnehmer für Abort gestimmt haben, bricht der Koordinator die Transaktion ab und sendet eine GLOBAL_ABORT-Nachricht an alle Teilnehmer.
4. Die Teilnehmer, die für Commit gestimmt haben, warten noch auf die endgültige Reaktion des Koordinators. Erhält ein Teilnehmer eine GLOBAL_COMMIT-Nachricht, schreibt er die Transaktion in die Datenbank. Falls der Teilnehmer allerdings eine GLOBAL_ABORT-Nachricht erhält, wird die Transaktion lokal abgebrochen.

Es ist sehr wichtig, dass alle durchgeführten Schritte von den Teilnehmern gut dokumentiert werden.

Es kommt zu Problemen, wenn das 2PC-Protokoll in einem System verwendet werden soll, in dem Fehler auftreten können. Da der Koordinator und die Teilnehmer Zustände annehmen können, in denen sie auf eine Nachricht warten. Wenn ein Prozess nun abstürzt, kann es sein, dass andere Prozesse ewig auf die Nachricht warten. Daher werden Timeouts verwendet.

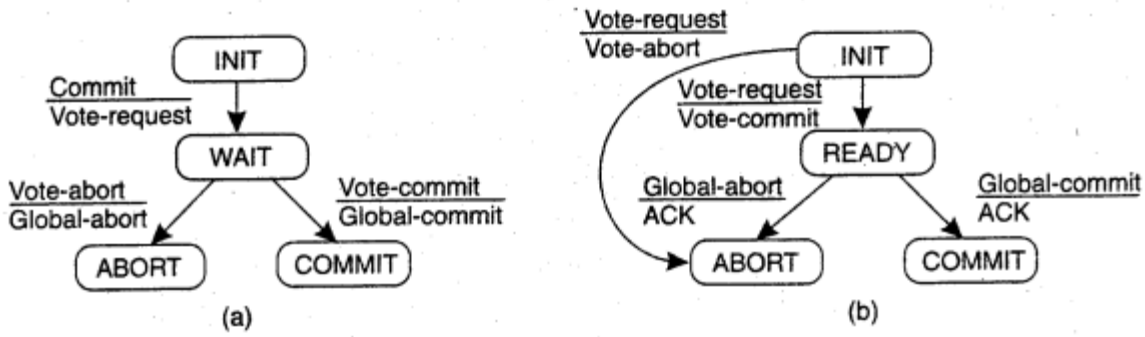

Abbildung 9: (a) Die Zustände des Koordinators im 2PC; (b) Die Zustände eines Teilnehmers [14]

Wenn die Teilnehmer im INIT-Zustand auf eine VOTE_REQUEST-Nachricht des Koordinators warten, und diese nicht nach einer bestimmten Zeit empfangen wird, beschließt der Teilnehmer die Transaktion abzubrechen und sendet eine VOTE_ABORT-Nachricht an den Koordinator.

Dieses Prinzip funktioniert aber auch in die andere Richtung. Wenn der Koordinator im WAIT-Zustand auf die Abstimmungen der Teilnehmer wartet, und nicht alle Antworten in einer speziellen Zeit eingehen, bricht der Koordinator die Transaktion ab und sendet eine GLOBAL_ABORT-Nachricht an alle Teilnehmer.

Wenn die Teilnehmer im READY-Zustand auf die Anweisungen des Koordinator warten, ob sie einen Commit oder einen Abort durchführen sollen, und diese nicht in einer bestimmten Zeit empfangen wird, kann der Teilnehmer nicht selbst entscheiden, ob die Transaktion abgeschlossen wird, oder nicht. Daher muss der Teilnehmer herausfinden, welche Nachricht der Koordinator gesendet hat. Die einfachste Lösung ist alle Teilnehmer blockieren zu lassen, bis der Koordinator die Nachricht senden kann. Anderenfalls ist es möglich, dass sich der Teilnehmer P an einen anderen Teilnehmer Q wendet, um dessen Zustand zu entnehmen, was die entsprechende Nachricht war. Hat Teilnehmer Q zum Beispiel den COMMIT-Zustand erreicht, muss der Koordinator eine GLOBAL_COMMIT-Nachricht gesendet haben.
Wenn sich Q allerdings noch im INIT-Zustand befindet, ist eine Situation aufgetreten, in der der Koordinator eine VOTE_REQUEST-Nachricht an alle Teilnehmer gesendet hat, welche T erreicht hat, aber Q nicht. Der Koordinator ist abgestürzt während er den Multicast gesendet hat. In diesem Fall können P und Q die Transaktion nur abbrechen.
Die schwierigste Situation entsteht, wenn sich Q ebenfalls im READY-Zustand befindet und auf die Antwort des Koordinators wartet. Wenn sich dann alle Teilnehmer im READY-Zustand befinden, kann keine Entscheidung getroffen werden, da dazu eine Nachricht des Koordinators notwendig wäre. Daher müssen alle Teilnehmer blockieren, bis der Koordinator wiederhergestellt wird.
Anfragen dieser Art werden als DECISION_REQUEST-Nachricht von P via Multicast an alle Teilnehmer gesendet.

Wenn der Koordinator nicht alle Antworten der Teilnehmer in der vorgeschriebenen Zeitspanne empfängt, muss er davon ausgehen, dass ein oder mehrere Teilnehmer ausgefallen sind. In diesem Fall sollte der Koordinator die Transaktion abbrechen und eine GLOBAL_ABORT-Nachricht an alle Teilnehmer senden. [14]

6.1.2 Drei-Phasen-Commit-Protokoll

Ein Problem des 2PC ist, dass die Teilnehmer nach einem Absturz des Koordinators eventuell keine endgültige Entscheidung treffen können. Es kann passieren, dass alle Teilnehmer blockieren, bis der Koordinator wiederhergestellt ist. Das Ziel des Drei-Phasen-Commit-Protokolls ist es, diese blockierenden Prozesse bei einem Absturz zu verhindern.

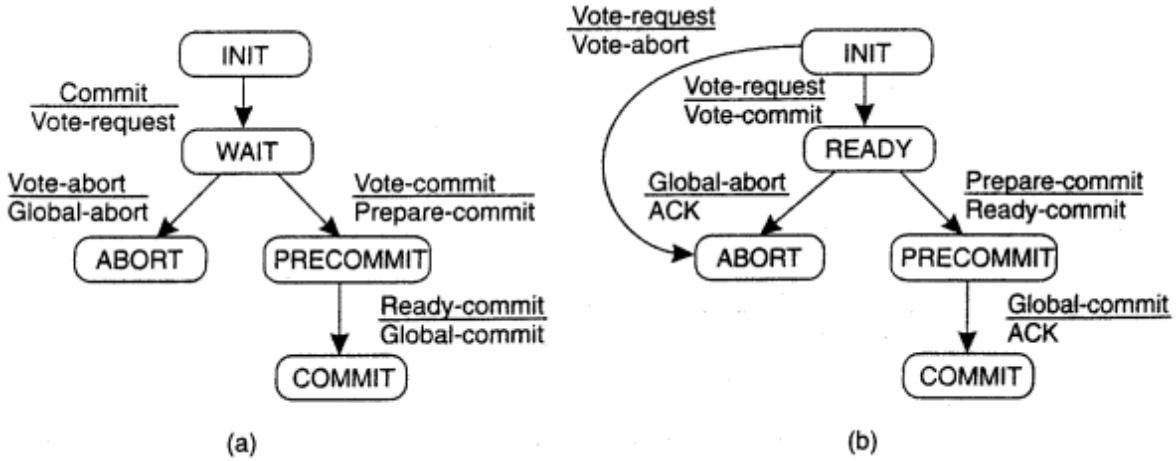

Abbildung 10: (a) Zustände des Koordinators im 3PC; (b) Zustände der Teilnehmer [14]

Wie im 2PC gibt es im 3PC einen Koordinator und eine Anzahl an Teilnehmern. Der große Unterschied ist, dass vor dem Commit (nach der Abstimmung) erneut, mit einer READY_COMMIT-Nachricht bestätigen müssen, dass sie bereit für den Commit sind. Es gibt zwei Bedingungen, die die jeweiligen Zustände erfüllen:

1. Es gibt keinen einzelnen Zustand, von dem aus es möglich ist, direkt einen COMMIT- oder ABORT-Zustand erreichen
2. Es gibt keinen Zustand, in dem es nicht möglich ist, eine Entscheidung zu treffen, von welcher aus man in den COMMIT-Zustand übergehen kann.

Der Koordinator sendet eine VOTE_REQUEST-Nachricht an alle Teilnehmer und wartet danach auf eine entsprechende Antwort. Wenn nicht alle Teilnehmer für den Commit stimmen, wird die Transaktion mit einer GLOBAL_ABORT-Nachricht abgebrochen. Stimmen alle Teilnehmer dafür, sendet der Koordinator eine COMMIT_PREPARE-Nachricht. Nun muss jeder Teilnehmer bestätigen, dass er für den Commit vorbereitet ist. Sobald das passiert ist, sendet der Koordinator die GLOBAL_COMMIT-Nachricht, durch welche die Transaktionen abgeschlossen werden.

Falls beim Koordinator in der PRECOMMIT-Phase eine Zeitüberschreitung auftritt, kann dieser daraus schließen, dass ein Teilnehmer abgestürzt ist. Der Koordinator weiß aber, dass der Teilnehmer zuvor mit einer VOTE_COMMIT-Nachricht gestimmt haben muss (sonst wäre es nicht zur PRECOMMIT-Phase gekommen) und teilt den anderen Teilnehmern durch eine GLOBAL_COMMIT-Nachricht mit die Transaktion abzuschließen. Der Koordinator verlässt sich darauf, dass das Wiederherstellungsprotokoll des abgestürzten Teilnehmers automatisch einen anderen Teilnehmer nach dem aktuellen Zustand fragt, wenn dieser wiederhergestellt wird.

Auch hier gibt es ein paar Situationen, in denen die Teilnehmer blockieren können. Wie im 2PC kann es passieren, dass ein Teilnehmer im INIT-Zustand auf die Aufforderung zur Abstimmung des Koordinators wartet und aufgrund eines Timeouts in den ABORT-Zustand übergeht, da er annimmt, dass der Koordinator abgestürzt ist. Wenn der Koordinator sich währenddessen im WAIT-Zustand befindet und auf die Antworten der Teilnehmer wartet, kommt es zu einer Zeitüberschreitung. Der Koordinator geht davon aus, dass ein Teilnehmer abgestürzt ist und bricht die Transaktion ab.
Falls eine Zeitüberschreitung bei einem Teilnehmer P im READY- oder PRECOMMIT-Zustand auftritt, muss P wie im 2PC einen anderen Teilnehmer Q nach dem aktuellen Zustand fragen, da der Koordinator wahrscheinlich ausgefallen ist. Befindet sich Q im INIT-Zustand, kann die Transaktion abgebrochen werden, denn Q kann nur in diesem Zustand sein, wenn kein anderer im PRECOMMIT-

Zustand ist. Ein Teilnehmer kann den PRECOMMIT-Zustand nur dann erreichen, wenn der Koordinator vor seinem Absturz den PRECOMMIT-Zustand erreicht hat. Diesen Zustand kann er aber nur erreichen, wenn er von jedem Teilnehmer eine Stimme für den Commit erhalten hat. Mit anderen Worten, kein Teilnehmer kann sich im Zustand INIT befinden, während ein anderer Teilnehmer im Zustand PRECOMMIT ist. Wenn sich alle Teilnehmer die P kontaktieren können im READY-Zustand befinden (Mehrheit der Gesamtheit), wird die Transaktion abgebrochen. [14]

6.2 Begriffe der Konsistenz

6.2.1 ACID – Atomic, Consistent, Isolated, Durable

Jede Transaktion hat vier Eigenschaften die immer Zutreffen. Diese Eigenschaften ergeben das Akronym ACID.

6.2.1.1 Atomar

Alle Transaktionen sind atomar. Das bedeutet, dass eine Transaktion entweder vollständig oder gar nicht ausgeführt wird. Wenn sie ausgeführt wird, geschieht dies in einer einzigen, unteilbaren, unmittelbaren Aktion. Die Zwischenzustände der Transaktion sind nicht von außen einsehbar. [15]

6.2.1.2 Konsistent

Die zweite Eigenschaft besagt, dass die Daten in einer Datenbank auch nach der Transaktion konsistent sind, sofern die Datenbank davor konsistent war. Ein Beispiel ist das Gesetz der Gelberhaltung in Banken. Nach einer internen Überweisung muss der Geldbetrag in der Bank der gleich wie vorher sein. Während der Transaktion kann sich dieser aber kurzfristig ändern. Wichtig ist, dass diese Verletzung außerhalb der Transaktion nicht sichtbar ist. [15]

6.2.1.3 Isoliert

Die Eigenschaft „Isoliert“ sagt aus, dass bei einer gleichzeitigen Ausführung von mehreren Transaktionen das Endergebnis so aussieht, als wären alle Transaktionen sequentiell in einer bestimmen Reihenfolge ausgeführt worden. [15]

6.2.1.4 Dauerhaft

Die letzte Eigenschaft besagt, dass alle Ergebnisse oder Änderungen die in einer Transaktion erzielt werden, permanent in der Datenbank persistiert werden, sobald die Transaktion mit einem Commit abgeschlossen wurde. Kein Fehler nach dem Commit kann die Ergebnisse verwerfen oder rückgängig machen. [15]

6.2.2 BASE – Basically Available, Soft State, Eventually Consistent

BASE ist ein alternativer Ansatz der häufig in NoSQL-Datenbanken angewandt wird. Auch hier wird das Akronym aus den Anfangsbuchstaben gebildet.

6.2.2.1 Basically Available

Daten können jederzeit abgerufen werden, sind aber eventuell „veraltet“. Wird beispielsweide die Timeline von Twitter aufgerufen, kann es sein, dass eine veraltete Version geladen wird, und direkt danach eine Aktualisierung verfügbar ist. [16]

6.2.2.2 Soft State

Die Daten des Systems bleiben nicht bestehen. Werden sie in einem festgelegten Zeitintervall nicht aufgefrischt oder aktualisiert, werden sie verworfen und sind nicht mehr verfügbar. [16]

6.2.2.3 Eventually Consistent

Diese Eigenschaft besagt, dass er Datensatz irgendwann konsistent sein wird, sofern eine hinreichend lange Zeit ohne Schreibvorgänge und Fehler gegeben ist. [16]

6.3 2-Phasen-Sperrprotokoll

Beim 2-Phasen-Sperrprotokoll (2PL) wird davon ausgegangen dass jede Transaktion zwei Phasen durchläuft:

1. Wachstumsphase: Die notwendigen Sperren werden gesetzt, können aber nicht freigegeben werden
2. Schrumpfphase: Die Sperren werden freigegeben, können aber nicht angefordert werden

Dieses Protokoll kennt drei Sperrzustände:

1. Read-Lock (auch Shared-Lock): mehrere Prozesse können lesend auf das gesperrte Objekt zugreifen.
2. Write-Lock (auch Exclusive-Lock): nur der sperrende Prozess kann auf das Objekt zugreifen und dieses auch schreiben.
3. Entsperrt

Es gibt zwei Erweiterungen des 2PL:

1. Konservatives 2-Phasen-Sperrprotokoll:
   Alle für die Transaktion notwendigen Sperren müssen vor der ersten Aktion erworben worden sein. Dieser Vorgang wird auch als Preclaiming bezeichnet.
2. Striktes 2-Phasen-Sperrprotokoll:
   Alle gesetzten Write-Locks werden erst nach der letzten Operation einer Transaktion freigegeben. Das verhindert, dass sich Transaktionen die sich gegenseitig beeinflussen sich stufenweise zurücksetzen.

[17]

7 Konsistenz und Datenhaltung

7.1 Entwurf von verteilten Datenbanken

7.1.1 Fragmentierung in VDBMS

Es gibt mehrere Möglichkeiten eine Tabelle zu fragmentieren: horizontal, vertikal und kombiniert. Damit die Fragmentierung gültig ist, müssen drei Korrektheits-Anforderungen erfüllt sein:

- Rekonstruierbarkeit:
  Die ursprünglichen Daten lassen sich aus den Fragmenten wiederherstellen
- Vollständigkeit:
  Jedes Datum ist einem Fragment zugeordnet
- Disjunktheit:
  Die Fragmente überlappen sich nicht. Ein Datum kann nicht mehreren Fragmenten zugeordnet sein

Für das Kommende wird eine Beispielstabelle zur Veranschaulichung verwendet, die wie folgt aussieht. Der Primärschlüssel ist dabei PersNr:

Professoren						
PersNr	Name	Rang	Raum	Fakultät	Gehalt	Steuerklasse
2125	Sokrates	C4	226	Philosophie	85000	1
2126	Russel	C4	232	Philosophie	80000	3
2127	Kopernikus	C3	310	Physik	65000	5
2133	Popper	C3	52	Philosophie	68000	1
2134	Augustinus	C3	309	Theologie	55000	5
2136	Curie	C4	36	Physik	95000	3
2137	Kant	C4	7	Philosophie	98000	1

Abbildung 11: Beispielstabelle für die Datenbankfragmentierung [18]

7.1.1.1 Horizontaler Entwurf

Bei der horizontalen Fragmentierung wird die Relation (R) anhand der Reihen aufgeteilt. Die Reihen dürfen sich in den Fragmenten nicht überlappen.

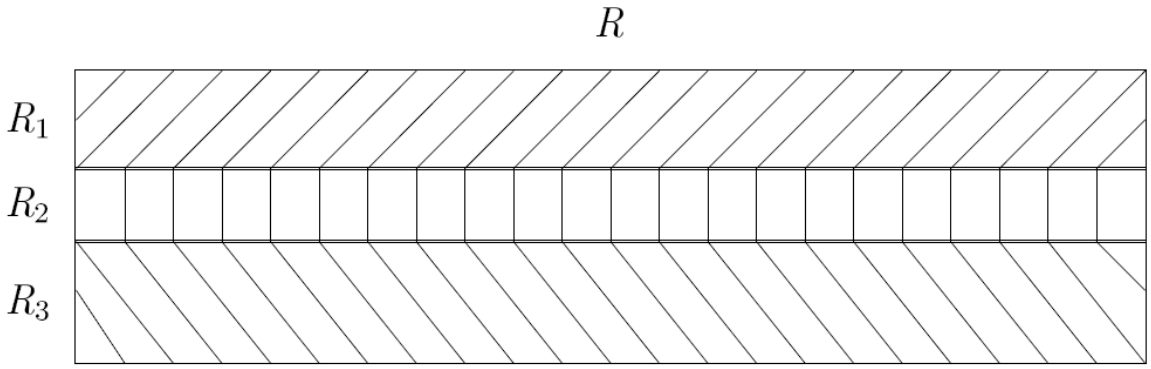

Abbildung 12: Darstellung einer horizontal fragmentierten Tabelle [18]

Damit man die Datenmengen voneinander trennen kann, müssen sogenannte Zerlegungsprädikate ($p_1 \dots p_n$) definiert werden, welche man auch als Zerlegungsbedingungen bezeichnen könnte. Aus diesen Prädikaten ergeben sich die Zerlegungen von R. So würden die Fragmente mit einem Prädikat aussehen.

$$R_1 \coloneqq \sigma_{p1}(R)$$
$$R_2 \coloneqq \sigma_{\neg p1}(R)$$

Dabei ist zu beachten, dass $\sigma_{p1}(R)$ jene Datenmenge ist, die die Prädikatsbedingung erfüllen und $\sigma_{\neg p1}(R)$ jene Datenmenge ist, die die Prädikatsbedingung nicht erfüllt.

Werden zwei Prädikate (p_1, p_2) verwendet, ergeben sich vier Zerlegungen:

$$R_1 \coloneqq \sigma_{p1 \wedge p2}(R)$$
$$R_2 \coloneqq \sigma_{p1 \wedge \neg p2}(R)$$
$$R_3 \coloneqq \sigma_{\neg p1 \wedge p2}(R)$$
$$R_4 \coloneqq \sigma_{\neg p1 \wedge \neg p2}(R)$$

Dabei werden die verschiedenen Datenmengen der Prädikate mit einem UND kombiniert. Generell kann man sagen, dass man aus n Zerlegungsprädikaten ($p_1 \dots p_n$) auch 2^n Fragmente erhält.

Die Prädikate werden wie WHERE-Konditionen formuliert und könnten beispielsweise so aussehen:

$$p_1 \equiv Fakultät = 'Theologie'$$
$$p_2 \equiv Fakultät = 'Physik'$$
$$p_3 \equiv Fakultät = 'Philosophie'$$

Angewandt ergeben diese Prädikate die einzelnen Fragmentierungen. Dadurch, dass viele Kombinationen eine leere Menge liefern würden, gibt es bestenfalls vier Kombinationen mit Inhalt.

$$TheolProfs' \coloneqq \sigma_{p1 \wedge \neg p2 \wedge \neg p3}(Professoren) = \sigma_{p1}(Professoren)$$
$$PhysikProfs' \coloneqq \sigma_{\neg p1 \wedge p2 \wedge \neg p3}(Professoren) = \sigma_{p2}(Professoren)$$
$$PhiloProfs' \coloneqq \sigma_{\neg p1 \wedge \neg p2 \wedge p3}(Professoren) = \sigma_{p3}(Professoren)$$
$$AndereProfs' \coloneqq \sigma_{\neg p1 \wedge \neg p2 \wedge \neg p3}(Professoren)$$

[18]

7.1.1.2 Vertikaler Entwurf

Bei der vertikalen Frgamentierung werden Attribute mit gleichem Zugriffsmuster (wie oft wird auf das Attribut zugegriffen?) zusammengefasst. Die einzelnen Fragmente haben also einen Bruchteil der Attribute, die die vertikal gespaltene Tabelle ursprünglich hatte. Allerdings kann sie nicht beliebig aufgeteilt werden, da sie so nicht Rekonstruierbar wäre. Es gibt zwei Ansätze, die Rekonstruierbarkeit garantieren:

1. Jedes vertikale Fragment muss den Primärschlüssel der Tabelle enthalten
2. Für jede Datenreihe wird ein künstlicher Schlüssel erstellt, der die Reihe identifiziert. Jedes Fragment muss den künstlichen Schlüssel enthalten.

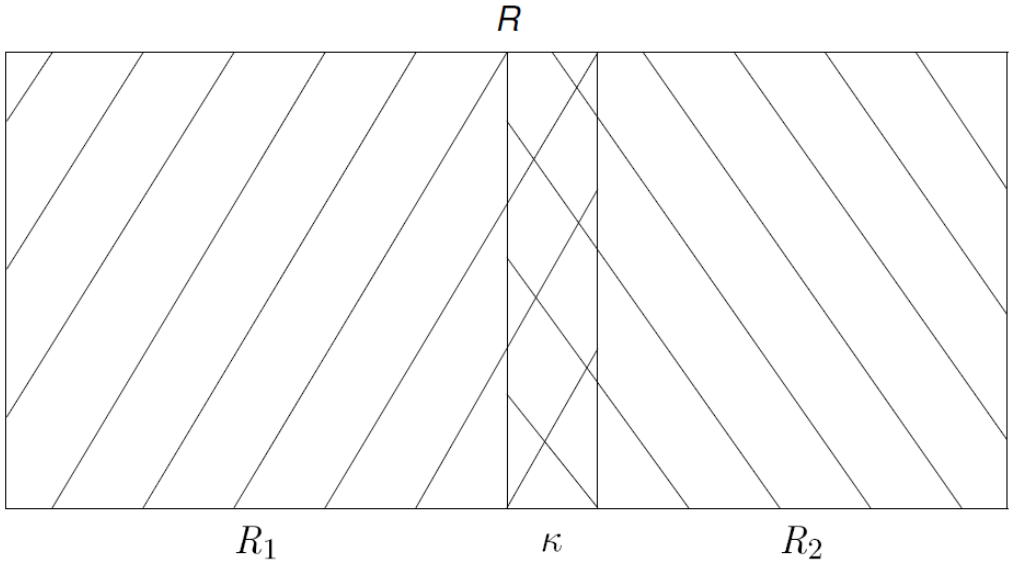

Abbildung 13: Darstellung einer vertikal Fragmentierten Tabelle [18]

$$\text{ProfVerw} := \Pi_{PersNr,Name,Gehalt,Steuerklasse}\,(Professoren)$$
$$Profs := \Pi_{PersNr,Name,Rang,Raum,Fakultät}(Professoren)$$

Dadurch, dass der Primärschlüssel (PersNr) in beiden Fragmenten enthalten ist, kann die Tabelle durch einen Join wiederhergestellt werden. Die Kondition des Joins lautet:
„ProfVerw.PersNr = Profs.PersNr“. Das Attribut Name ist bei beiden Fragmenten enthalten, da Personen relativ selten ihren Namen ändern und diese Redundanz daher toleriert werden kann.

[18]

7.1.1.3 Kombinierter Entwurf

Bei der kombinierten Fragmentierung werden die vertikale und die horizontale Fragmentierung auf dieselbe Relation angewandt.

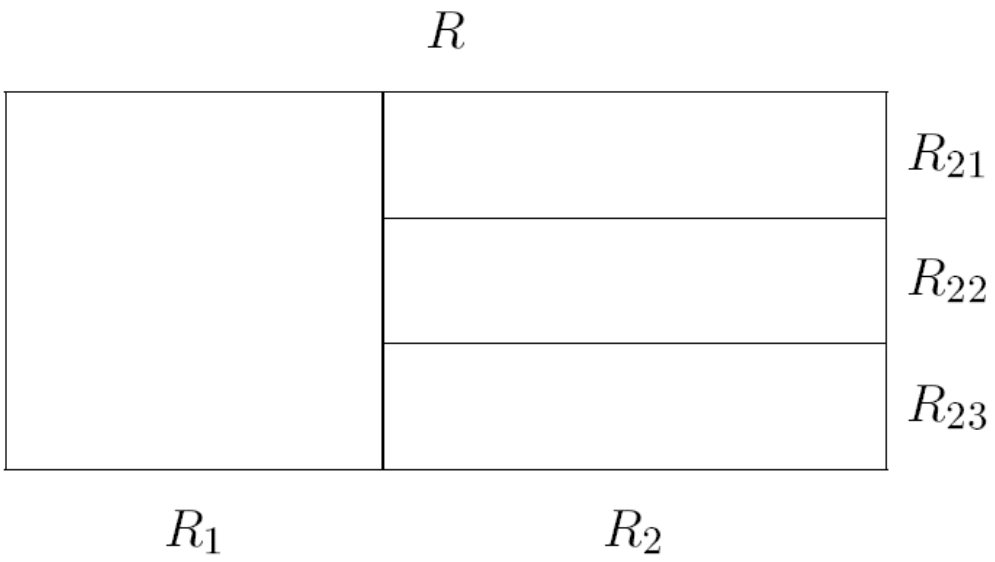

Abbildung 14: Darstellung einer auf eine horizontale- folgende vertikale Fragmentierung [18]

Die Fragmente aus Abbildung 14 können wie folgt wieder zusammengefügt werden:

$$R = R_1 \bowtie_P (R_{21} \cup R_{22} \cup R_{23})$$

p ist dabei ein Prädikat, dass den Primärschlüssel der beiden Fragmente R_1 und R_2 auf Gleichheit überprüft.

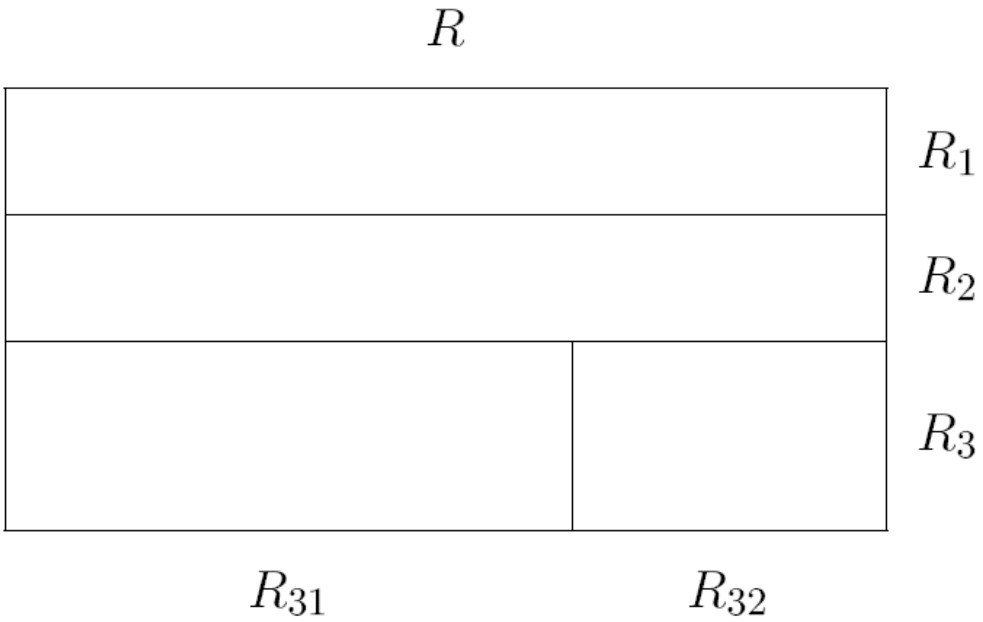

Abbildung 15: Darstellung einer auf eine vertikale- folgende horizontale Fragmentierung [18]

Die Fragmente aus Abbildung 15 können wie folgt wieder zusammengefügt werden:

$$R = R_1 \cup R_2 \cup (R_{31} \bowtie_{R_{31}.k=R_{32}.k} R_{32})$$

k ist dabei der Primärschlüssel der beiden vertikalen Fragmente. [18]

7.1.2 Allokation der Fragmente

Nach dem Festlegen des Fragmentierungsschemas, müssen die einzelnen Fragmente auf Stationen des VDBMS aufgeteilt werden. Dieser Vorgang wird als Allokation bezeichnet.

Es gibt zwei Arten der Replikation:

- **redundanzfreie Replikation**: Jedes Fragment wird genau einer Station zugeordnet. Kein Fragment darf auf zwei verschiedenen Stationen vorhanden sein.
- **Allokation mit Replikation**: Hierbei werden einige Fragmente repliziert und mehreren Stationen zugeordnet. Dieses Prinzip wird in Abbildung 16 grafisch darstellt.

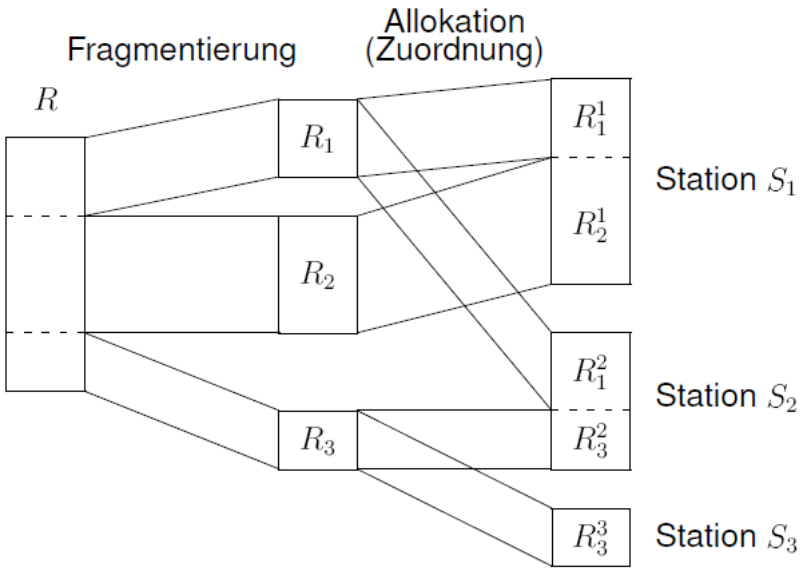

Abbildung 16: Fragmentierung und Allokation einer Tabelle

Wird ein Fragment R_1 der Station S_1 zugewiesen, so erhält es in dieser Abbildung den Namen R_1^1.

Bei dem verwendeten Beispiel würde das wie folgt aussehen:

Station	Bemerkung	zugeordnete Fragmente
S_{Verw}	Verwaltungsrechner	{*ProfVerw*}
S_{Physik}	Dekanat Physik	$\{PhysikVorls, PhysikProfs\}$
S_{Philo}	Dekanat Philosophie	$\{PhiloVorls, PhiloProfs\}$
S_{Theol}	Dekanat Theologie	$\{TheolVorls, TheolProfs\}$

Abbildung 17: Allokation der fragmentierten Tabelle

Bei diesem Beispiel handelt es sich um eine Allokation ohne Replikation, also eine redundanzfreie Zuordnung. [18]

8 Abbildungsverzeichnis

9 Literaturverzeichnis

[1] A. S. Tanenbaum und M. v. Steen, „Nachrichtenorientierte perisistente Kommunikation,“ in *Verteilte Systeme: Prinzipien und Paradigmen*, pp. 170-177.

[2] Apache, „Apache ActiveMQ -- Hello World,“ Apache, [Online]. Available: http://activemq.apache.org/hello-world.html. [Zugriff am 28 05 2016].

[3] S. Brinnich und N. Hohenwarter, SOA Ausarbeitung, 2015.

[4] K. Chandra, „Load-Distribution Methods,“ in *Load Balancing Servers, Firewalls and Caches*, pp. 28-31.

[5] T. Taschner und A. Kölbl, „Response Time,“ in *Load-Balancing Ausarbeitung*, 2015, pp. 7-8.

[6] K. Chandra, „Defining Session Persistence,“ in *Load-Balancing Servers, Firewalls and Caches*, pp. 50-51.

[7] K. Chandra, „The Megaproxy Problem,“ in *Load-Balancing Servers, Firewalls and Caches*, pp. 58-60.

[8] TechTarget, „high-performance computing (HPC),“ [Online]. Available: http://searchenterpriselinux.techtarget.com/definition/high-performance-computing. [Zugriff am 29 05 2016].

[9] PearsonHighered, „What is High Availability?,“ [Online]. Available: https://www.pearsonhighered.com/samplechapter/0130893552.pdf. [Zugriff am 29 05 2016].

[10] B. Smith, Beginning JSON, 2015.

[11] JSON.org, „JavaScript Object Notation,“ [Online]. Available: http://json.org. [Zugriff am 26 05 2016].

[12] S. Tilkov, M. Eigenbrodt, S. Schreier und O. Wolf, REST und HHTP: Entwicklung nach dem Architekturstil des Web, dpunkt.verlag, 2015.

[13] A. S. Tanenbaum und M. v. Steen, „Webservercluster," in *Verteilte Systeme: Prinzipien und Paradigmen*, pp. 600-602.

[14] A. S. Tanenbaum und M. v. Steen, „Verteiler Commit," in *Verteiler Systeme: Prinzipien und Paradigmen*, pp. 387-396.

[15] A. S. Tanenbaum und M. v. Steen, „Eigenschaften von Transaktionen," in *Verteilte Systeme: Prinzipiel und Paradigmen*, pp. 38-39.

[16] M. Pöhls, „NoSQL: ACID vs. BASE," [Online]. Available: http://edenora.de/posts/286/nosql-acid-vs-base/. [Zugriff am 26 05 2016].

[17] FH Köln, „2-Phasen-Sperrprotokoll," [Online]. Available: http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/2-Phasen-Sperrprotokoll. [Zugriff am 26 05 2016].

[18] A. Kemper, „Horizontale und vertikale Fragmentierung," in *Datenbanksysteme: Eine Einführung*, pp. 495-503.