Universität Hamburg

Fakultät für Mathematik,

Informatik und Naturwissenschaften

Bachelorarbeit

Performanz- und Reaktivitätssteigerung von OODBMS vermittels der Web-Caching-Hierarchie unter Einsatz und Adaption offener Standards

Felix Gessert

7gessert@informatik.uni-hamburg.de
Studiengang Informatik (BSc)
Matr.-Nr. 5945597
Fachsemester 6

Florian Bücklers

7bueckle@informatik.uni-hamburg.de
Studiengang Informatik (BSc)
Matr.-Nr. 5991327
Fachsemester 6

Erstgutachter: Professor N. Ritter
Zweitgutachter: Professor W. Lamersdorf

Performanz- und Reaktivitätssteigerung von OODBMS vermittels der Web-Caching-Hierarchie unter Einsatz und Adaption offener Standards

Autoren: Florian Bücklers, Felix Gessert

Abstract: In dieser Arbeit werden wir einen Ansatz vorstellen, mit dem die Infrastruktur des Webs für einen HTTP-basierten Zugriff auf objektorientierte Datenbanken dergestalt umgesetzt wird, dass durch die Web-Caching-Hierarchie eine beschleunigende, latenzverringernde Zwischenschicht entsteht. Diesen Ansatz nennen wir **Orestes** (***O****bjects* ***REST****fully* ***E****ncapsulated in* ***S****tandard-formats*).

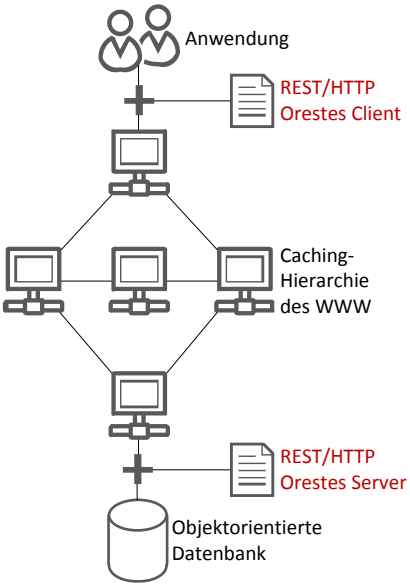


Abbildung 1 Zugriffsszenario

Wie in Abbildung 1 gezeigt, muss zu diesem Zweck sowohl eine HTTP-Schnittstelle entworfen werden, als auch eine geeignete Menge an Repräsentationsformaten zur Übertragung identifiziert, sowie ggf. adaptiert und erweitert werden. Dies sichert einerseits die Standardkonformität des Ansatzes und katalysiert andererseits die Nutzbarkeit für cloud- und webbasierte Anwendungsfälle, in denen eine Prävalenz von Schnittstellen auf Basis des REST/HTTP Paradigmas sich deutlich abzeichnet.

Im Verlauf dieser Arbeit werden wir zeigen, wie eine geeignete Architektur der Schnittstelle und die darunterliegende Ausgestaltung der Kommunikations- und Verarbeitungsprozesse eine Verbesserung der Datenbankperformanz (Entlastung) und der Latenzzeit (Caching) ermöglicht. Durch geeignete Mechanismen werden wir die Fähigkeiten der Caching-Hierarchie des Webs so nutzbar machen, dass Inhalte (Objekte) der Datenbank vorrübergehend in Web-Caches residieren. Auf diese Weise wird Skalierbarkeit auf Basis einer bestehenden Infrastruktur erzielt. Einerseits übertrifft diese Cache-Hierarchie an Verbreitung und Größe jedes andere Caching-Cluster und ist des Weiteren nicht mit einer finanziellen Belastung verbunden. Da das Validierungs- und Expirationsmodell dieser HTTP-Caches aber auf die Anforderungen des dokumentenbasierten Datenverkehrs des Webs ausgerichtet ist, werden wir durch passende Modelle eine Möglichkeit eröffnen, den spezifischen Anforderungen für das Caching von Objekten eines OODBMS dennoch gerecht zu werden.

Zu diesem Zweck ist sowohl auf der Clientseite (also der DB benutzenden Applikation) als auch auf der Serverseite (dem OODBMS) eine Komponente zu entwerfen, die diesen Ansprüchen gerecht wird. Die Validität dieser theoretischen Beschreibung werden wir durch eine Implementierung verifizieren. Diese Umsetzung wird eine von Client-Persistenz-API und OODBMS entkoppelte, beschleunigende HTTP-Übertragungsschicht sein. Es ist also das Ziel dieser Arbeit, durch eine standardkonforme Abbildung von OODB-Zugriffen auf das HTTP-Protokoll ein Konzept zum Caching von Objekten eines OODBMS durch die Web-Infrastruktur zu entwerfen.

Inhaltsverzeichnis

1 Einleitung

1.1 Überblick

Um das Ziel dieser Arbeit zu verdeutlichen, werden wir den Ablauf einer effektiven HTTP (*Hypertext Transfer Protocol*) basierten Kommunikation zwischen Anwendung und objektorientierter Datenbank skizzieren und der klassischen gegenüberstellen. Dieser Kommunikationsakt stellt den Kern dieser Arbeit dar. Seine Konzeption und Modellierung werden es ermöglichen, die im Abstract angeführten Ziele der Performanz und Reaktivitätssteigerung von Zugriffen auf OODBMS (*Object Oriented Database Management Systems*) zu realisieren.

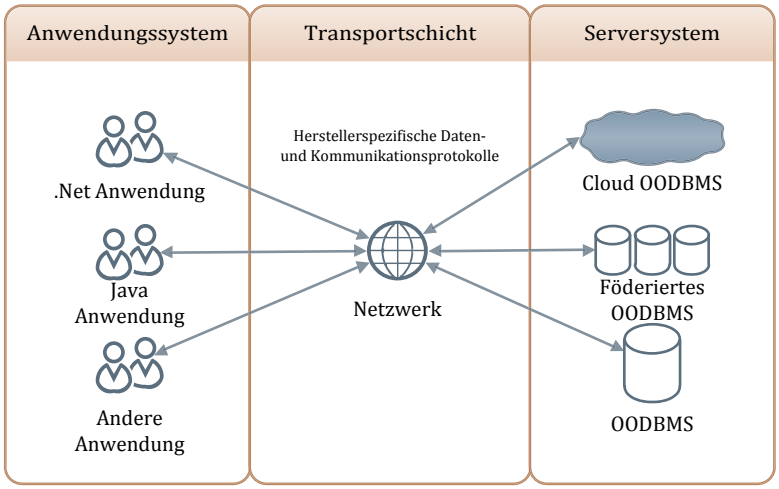


Abbildung 2 Klassische Applikations-OODBMS Kommunikation

Abbildung 2 zeigt die Komponenten, die an dem Ablauf der Kommunikation beteiligt sind: das *Anwendungssystem* setzt mithilfe einer programmiersprachenspezifischen *Bibliothek* Anfragen über ein *Netzwerk* an ein OODBMS ab, wobei ein *herstellerspezifisches Protokoll* auf der Transportschicht aufsetzt. Die Anfrage wird von dem *Serversystem* entgegengenommen, durch das *Datenbanksystem* verarbeitet und über die aufgebaute Verbindung beantwortet.

Um in dieser Anordnung die wichtigen Leistungskenngrößen, wie Durchsatz und Latenzzeit zu verbessern, sind lediglich Anpassungen innerhalb des Serversystems möglich. Typische Umstrukturierungen wären eine Master-Slave Replikation (Master beantwortet Lese- und Schreib-, der Slave lediglich Lesezugriffe) oder ein vertikaler Scale-Out (Ausbau der Hardware-Ressourcen).

Anders stellt sich ein HTTP basierter Zugriff (Abbildung 3) dar. Hier wird die Programmbibliothek zum Datenbankzugriff durch einen *REST/HTTP-Orestes-Client* erweitert, dessen Schnittstelle, Funktionalität und Architektur Untersuchungsgegenstand dieser Arbeit sind. Durch diesen REST/HTTP-Client wird je nach Anfragetyp eine HTTP-Nachricht erstellt, deren Inhalt durch ein geeignetes offenes Format repräsentiert wird **(1)**. Die Wahl und Austauschbarkeit geeigneter Formate und deren Anpassung wird in dieser Arbeit untersucht und ausgeführt. Wird während der Übertragung der Anfrage ein Web-Cache passiert, so

wird die Anfrage weitergeleitet, ggf. über eine Anzahl an Intermediate Hops (Knoten innerhalb der Cache-Hierarchie) **(2)**. Die eingehende Anfrage wird durch den Orestes-Server entpackt und dem OODBMS übergeben. Die daraus resultierende Antwort wird durch den Orestes-Server analog zu (1) als HTTP-Nachricht zurückgesandt **(3)**. Die Web-Caches, die von der Antwort passiert werden, stellen fest, dass ein aus ihrer Sicht unbekanntes Dokument vorliegt, speichern es in ihrem Cache und leiten es weiter **(4)**. So erreicht die Antwort wieder den REST/HTTP-Client, der aus dem Repräsentationsformat das Ergebnis der Anfrage an die Anwendung propagiert.

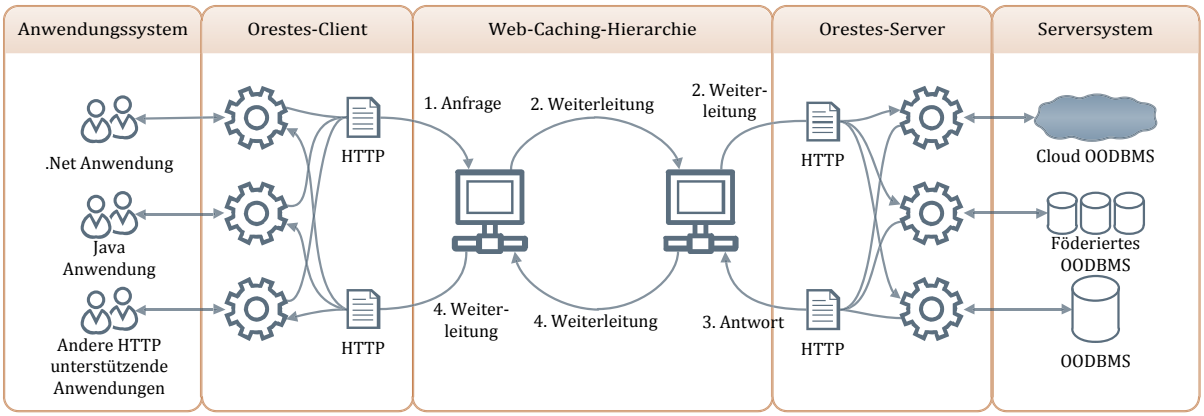


Abbildung 3 HTTP-basierte Kommunikation zwischen Applikation/OODBMS

Diese Schritte werden in Abbildung 4 zusammenfassend dargestellt.

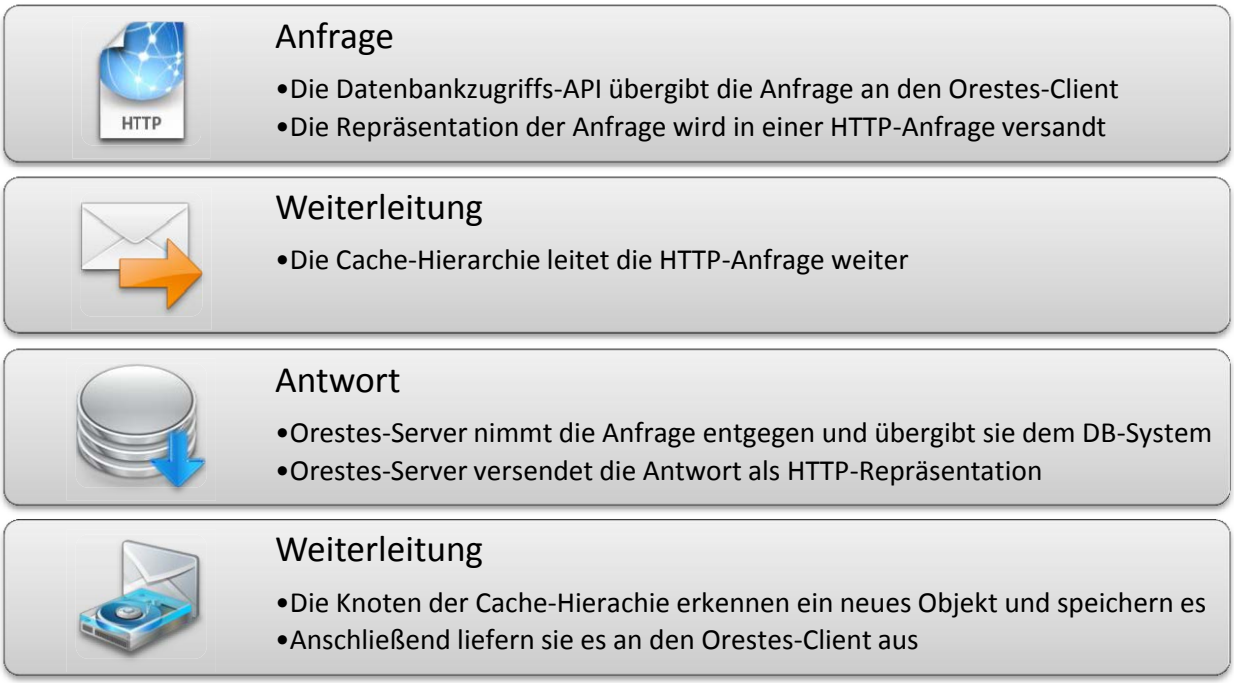


Abbildung 4 Schritte in der HTTP-Kommunikation zwischen Anwendung und OODBMS

In diesem ersten Szenario, bei dem eine initiale Anfrage gesendet wird, kommt der Einfluss der Cache-Hierarchie auf die Leistungskenngrößen des Gesamtsystems noch nicht zum Tra-

gen. Der entscheidende Vorteil wird deutlich, wenn eine Anfrage auf dieselbe Ressource (z.B. ein Objekte) erneut gestellt wird. Dabei können verschiedene Fälle eintreten, die in Abbildung 5 illustriert werden.

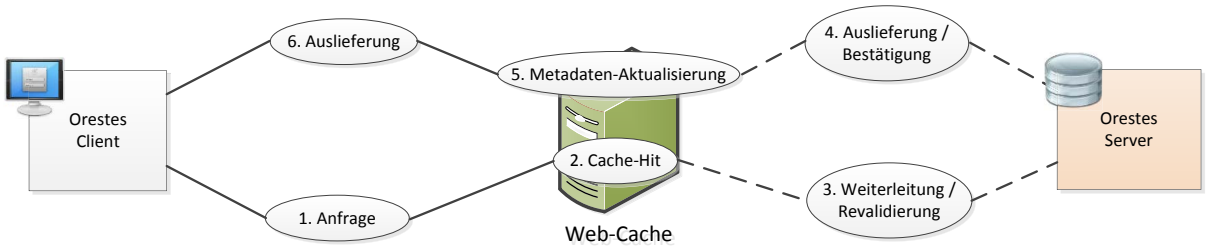


Abbildung 5 Anfrage über einen Web-Cache und mögliche Resultate

1. Die von der Applikation angenommene **Anfrage** wird vom Orestes-Client über HTTP versendet.
2. Erzielt der Web-Cache keinen **Cache-Hit** (das angefragte Objekt liegt nicht im Cache) fährt er mit Schritt 3 (Weiterleitung) fort, andernfalls gibt es zwei Möglichkeiten:
    a. Das Objekt im Cache ist durch die vormals vom Orestes-Server empfangenen Metadaten noch immer als valide (*fresh*) gekennzeichnet und der Client hat nicht explizit eine Revalidierung angefordert → der Web-Cache fährt mit Schritt 6 (Auslieferung) fort.
    b. Das Objekt im Cache ist nicht länger als aktuell (sondern *stale*) anzusehen, oder der Client hat eine explizite Revalidierung angefragt → der Web-Cache fährt mit Schritt 3 (Revalidierung) fort. Dabei wird die Anfrage zusätzlich mit der bekannten Versionsnummer, sowie einer Anweisung zur bedingten Auslieferung versehen (Auslieferung nur, falls bekannte Version stale).
3. Die Anfrage wird an den Orestes-Server (bzw. die in der Hierarchie folgenden Caches) **weitergeleitet.**
4. Die aktuelle Version des Objektes wird **ausgeliefert** (nebst seiner *Freshness*-Parameter, die ggf. aus statischen, statistischen oder heuristischen Berechnungen hervorgehen). Falls die Anfrage mit einer noch aktuellen Versionsnummer versehen war, wird die Gültigkeit ohne Auslieferung des Objekts **bestätigt**.
5. Der Web-Cache **aktualisiert** die Metadaten (speziell die *Freshness*-Informationen) und ggf. das Objekt selbst.
6. Das vom Orestes-Client angefragte Objekt wird **ausgeliefert** und vom Orestes-Client der Applikation übergeben.

Das Szenario zeigt, dass der Zustand des Caching-Systems bezüglich einer Ressource drei verschiedene Zustände einnehmen kann, die unterschiedlichen Einfluss auf die Performance von Kommunikation und Serversystem haben:

1. Der Cache besitzt die angefragte Ressource nicht → leichte Performanceeinbußen durch Verarbeitungsschritte der Caching-Hierarchie

2. Der Cache besitzt die angefragte Ressource, die jedoch stale (d.h. abgelaufen) ist → Performancegewinn durch Vermeidung einer erneuten Objektübertragung, im Falle einer unveränderten Ressource
3. Der Cache besitzt die angefragte, als aktuell (fresh) gekennzeichnete Ressource
    a. Das Serversystem wird entlastet → Performanzsteigerung
    b. Die Round Trip Time der Anfrage (d.h. Anzahl der passierten Netzwerkknoten und die Netzwerkstrecke) verringert sich → Reaktivitätssteigerung

In zwei von drei möglichen Szenarien wird also der Server entlastet und die Kommunikation beschleunigt. Speziell bei leseintensiven Anwendungsszenarien ist dadurch ein erheblicher Geschwindigkeitsgewinn zu erwarten.

1.2 Benennung des Projektes

Orestes

Objects **REST**fully **E**ncapsulated in **S**tandardformats is an architectural approach to web-centred object transfer, retrieval and representation, based on the REST Pattern, HTTP and open formats.

Abbildung 6 Projektgegenstand und Definition

In Anlehnung an verbreitete, aus griechischen Heldennamen bestehende Akronyme (Ajax, Jason, etc.), werden wir die zu entwerfende REST Schnittstelle und ihr konzeptuelles Fundament in analoger Weise als **Orestes** bezeichnen. Dies kann einerseits der Verbreitung des Ansatzes Vorschub leisten und fördert andererseits seine Kommunizierbarkeit. Orestes (Abbildung 6) steht dabei bewusst nicht für eine Implementierung, sondern lediglich für die Architektur und ihre Merkmale (Schnittstellen, Formate und Protokolle), die wir im Verlauf dieser Arbeit entwerfen werden.

Abbildung 7 Logo des Orestes-Projekts (Photographische Vorlage mit Genehmigung von H. Rönsch)

Die mythologische Figur des Orestes (gr. Ὀρέστης), Sohn des mykenischen Königs Agamemnon, ist eine Figur des trojanischen Krieges. Er wird später mit Wahnsinn (die Erinnyen im Bildhintergrund) für den Rachemord an seiner Mutter gestraft, aber nach seiner Freisprechung zum Herrscher über Sparta ernannt. Sein Aufstieg soll uns Metapher für die Rolle des Paradigmas „Objekt“ in der Persistenzschicht sein.

1.3 Umsetzung

Die Integration des Orestes-Clients in die Datenbankzugriffs-API ist aus Anwendungssicht vollständig transparent. Die Realisierung dieses Anspruchs erfolgt durch drei Komponenten:

- Die Implementierung der generischen Orestes-Übertragungsschicht.
- Die Einbettung der Orestes-Aufrufe in die Datenbankzugriffs-API.
- Die Einbettung einer Orestes-Aufruf-Ausführungsschicht auf Datenbankseite.

Im Rahmen dieser Arbeit werden wir eine Implementierung der Orestes-Übertragungsschicht vornehmen. Aus ihrer Umsetzung erwachsen die positiven Konsequenzen für Leistungsmerkmale (z.B. Latenzzeit). Die Implementierung der Orestes-Übertragungsschicht werden wir als **Orion** bezeichnen: **Or**estes **I**mplementation **O**n **N**etworklevel. Orion stellt also die Implementierung der Caching- und Schnittstellenkonzepte von Orestes auf Ebene von HTTP dar.

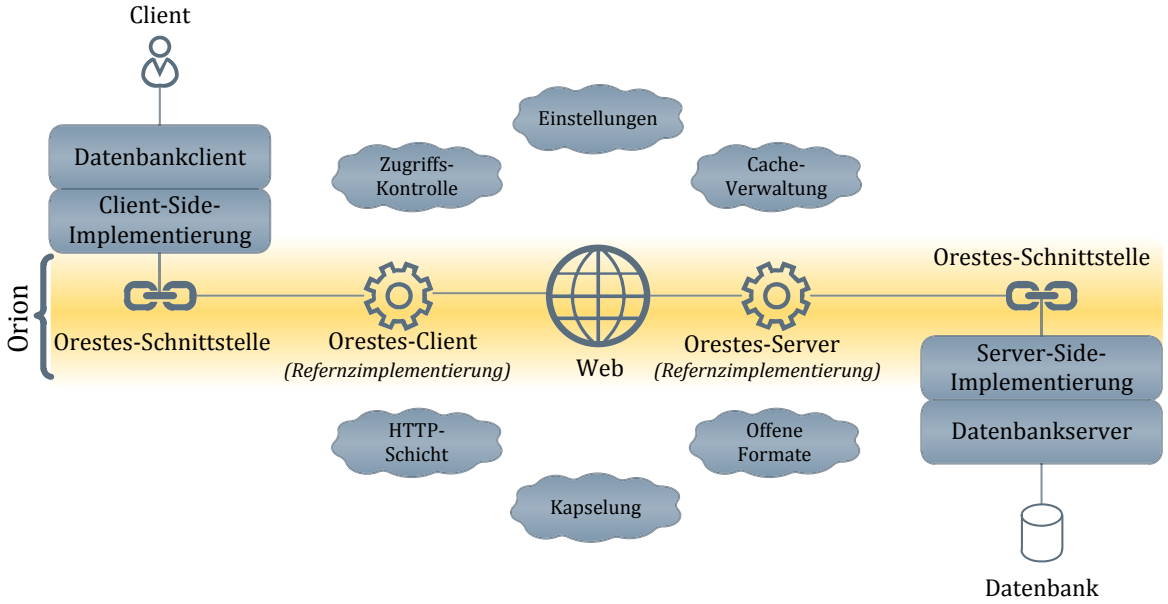


Abbildung 8 Die Implementierung der Orestes-Konzepte: Orion.

Abbildung 8 demonstriert die Struktur der Implementierung. Eine Client-Side-Implementierung im Datenbankclient bildet ihre Operationen (z.B. `makePersistent(obj)`) auf die Orestes-Schnittstelle des Orestes-Clients ab. Eine entsprechende Server-Side-Implementierung ist in der Lage, diese von dem Orestes-Server über dieselbe Orestes-Schnittstelle weitergegebenen Aufrufe auf den Datenbankserver abzubilden. Die clientseitig abgesetzten Aufrufe an der Orestes-Schnittstelle (z.B. Objekte laden) werden durch unsere Implementierung der HTTP-Übertragungsschicht (Orion) abgewickelt, wobei die performance-steigernden Konzepte dieser Arbeit zur Anwendung kommen.

1.4 Fragestellungen und Hypothesen

In diesem Abschnitt werden wir die wichtigsten zu untersuchenden Punkte und die diesbezüglichen Hypothesen und Fragestellungen vorstellen. Die Hypothese dieser Arbeit kann in folgende Aussage subsumiert werden: *„Durch ein geeignetes Zugriffs- und Kommunikationsmodell kann eine Performanz- und Reaktivitätssteigerung von objektorientierten Datenbanken auf Basis von Web-Protokollen und der Web-Infrastruktur erzielt werden“*.

1.4.1 Wahl und Adaption von Repräsentationsformaten

Für die Übertragung der Objekte ist die Auswahl mindestens eines konkreten Formats zur Übertragung notwendig. Im Zuge der durch HTTP implementierten *Content-Negotiation*, dem dynamischen Aushandeln des Repräsentationsformats einer Ressource zwischen Client und Server, können jedoch auch mehrere Lösungen nebeneinander existieren, ohne sich zu konterkarieren. Zu den Optionen und bekannten Vertretern aus dem Web-Umfeld zählen JSON, YAML, XML, HTML-Microformats und RDF. Für Orion werden wir eine Architektur entwickeln, die Austauschbarkeit und Koexistenz verschiedener Repräsentationsformate (z.B. für Objekte, Konfigurationen oder Querys) gestattet.

1.4.2 Struktur der REST/HTTP-Schnittstelle

Die Ausarbeitung einer geeigneten URI-Struktur zur Adressierung und zum Auffinden von Objekten (z.B. „http://orestes-srv.com/db/1992873“, mit „1992873“ als Objekt-ID) und anderen Ressourcen ist von großer Wichtigkeit, um dem Anspruch des HTTP-Protokolls als *Hypermedia* Protokoll und den Grundsätzen des REST-Architekturstils zu genügen. Alle Operationen der Schnittstelle müssen diskutiert und im Hinblick auf die verbreiteten Verarbeitungspraktiken von Web-Caches untersucht werden (z.B. URI-basierte Auswahlprädikate für Objektselektionen).

1.4.3 Entkopplung von Client und Server

Eines der Ziele der zu entwerfenden REST/HTTP-Schnittstelle ist die lose Kopplung von Client- und Serverseite. Diese garantiert die Austauschbarkeit der einzelnen Komponenten. Für den Orestes-Client werden wir ausarbeiten, auf welche Weise eine Metadaten-Verwaltung vorgenommen werden kann, beispielsweise zwecks der Prüfung, ob ein vom Cache empfangenes Objekt älter als ein lokales oder bereits geschriebenes ist. Für den Orestes-Server werden wir aufzeigen, nach welchen Gesichtspunkten eine effiziente Anfrageverarbeitung vorgenommen werden kann. In diesem Zusammenhang werden wird die These untermauern, dass eine Anfrage, die eine Anzahl von Objekten selektiert, am effektivsten durch eine URI-Liste beantwortet werden kann. Diese fordert den Orestes-Client dazu auf, die Objekte dediziert anzufragen, um ihre Cachebarkeit sicherzustellen.

1.4.4 Sitzungs- und Transaktionsverwaltung

Bei der Kommunikation zwischen dem Orestes-Client und dem Orestes-Server, muss ein Verfahren entworfen werden, das einerseits erlaubt den zugreifenden Benutzer zu authenti-

fizieren und andererseits sicherstellt, dass ausgelieferte Objekte nicht durch sitzungsbasierte Metainformationen ihre Cachebarkeit verlieren.

In Bezug auf die Transaktionsverwaltung werden wir ein Modell entwickeln, welches gestattet, Objekte separat zu übertragen und dennoch die Atomarität und Integrität der Transaktion sicherzustellen. Dabei ist ein standardkonformer Zuordnungsmechanismus von Requests zu Transaktionen zu konstruieren, um geschachtelte und parallele Transaktionen zu ermöglichen.

1.4.5 Behandlung von Referenzen und Sammlungstypen

Durch das Designziel einer separaten Objektübertragung (Wahrung der Cachebarkeit) entsteht die Notwendigkeit einer gesonderten Behandlung von Sammlungstypen und Referenzen (1:n, n:m und 1:1 Beziehungen). Auch der Grad der Indirektion bei Sammlungstypen muss diskutiert werden, da einerseits eine direkt eingebettete Verweisliste auf referenzierte Objekte eine höhere Volatilität der referenzierenden Objektrepräsentation nach sich zieht, andererseits durch eine Referenz zusätzliche Anfragen zum Dereferenzieren (Abrufen der Collection) generiert werden müssen.

1.4.6 Caching-Direktiven

Um dafür zu sorgen, dass die Einträge der Web-Caches so aktuell wie möglich sind und gleichzeitig so lange wie möglich gespeichert werden, müssen Richtlinien dafür entworfen werden, wann ein Objekt mit welchen HTTP-Caching-Direktiven ausgeliefert wird (z.B. während einer laufenden Transaktion). Ferner werden wir diskutieren, welche Erweiterungen des HTTP-Caching Modells existieren und in welcher Weise sie für Orestes nutzbar sind. Für die Abstraktionen von Orestes und ihre Entsprechungen in der REST/HTTP-Schnittstelle (z.B. Objekte, Schemata, Transaktionen) werden wir aufzeigen, in welchen Fällen und auf welche Weise Caching möglich ist.

1.4.7 Caching-Hierarchie

Die von dem Orestes-Server ausgelieferten, für das Caching vorgesehenen Repräsentationen durchlaufen bei einem webbasierten Nutzungsszenario eine Reihe von Web-Caches unterschiedlicher Art. Aus diesem Grund ist es wichtig zu untersuchen, mit welchen Techniken und Topologien Web-Caches in Netzwerken eingesetzt werden. Die Protokolle und Funktionsweisen von Web-Caches werden wir in der Konzeption des Generierungsschemas für Caching-Direktiven berücksichtigen. Auch die Auswirkungen von leistungssteigernden Netzwerk- und HTTP-Techniken (*Persistent Connections, Request Pipelining, Kompression, Simultaneous Connections, Delta Encoding*) auf Zwischensysteme werden wir dazu einbeziehen.

1.4.8 Zusammenfassung der Zielsetzung

Das Ziel dieser Bachelorarbeit liegt also in der Beantwortung und Diskussion der in Abbildung 9 stichwortartig zusammengefassten Fragestellungen. Mit den Lösungen dieser Fragestellungen werden wir eine Implementierung von Orestes als Übertragungsschicht vornehmen.

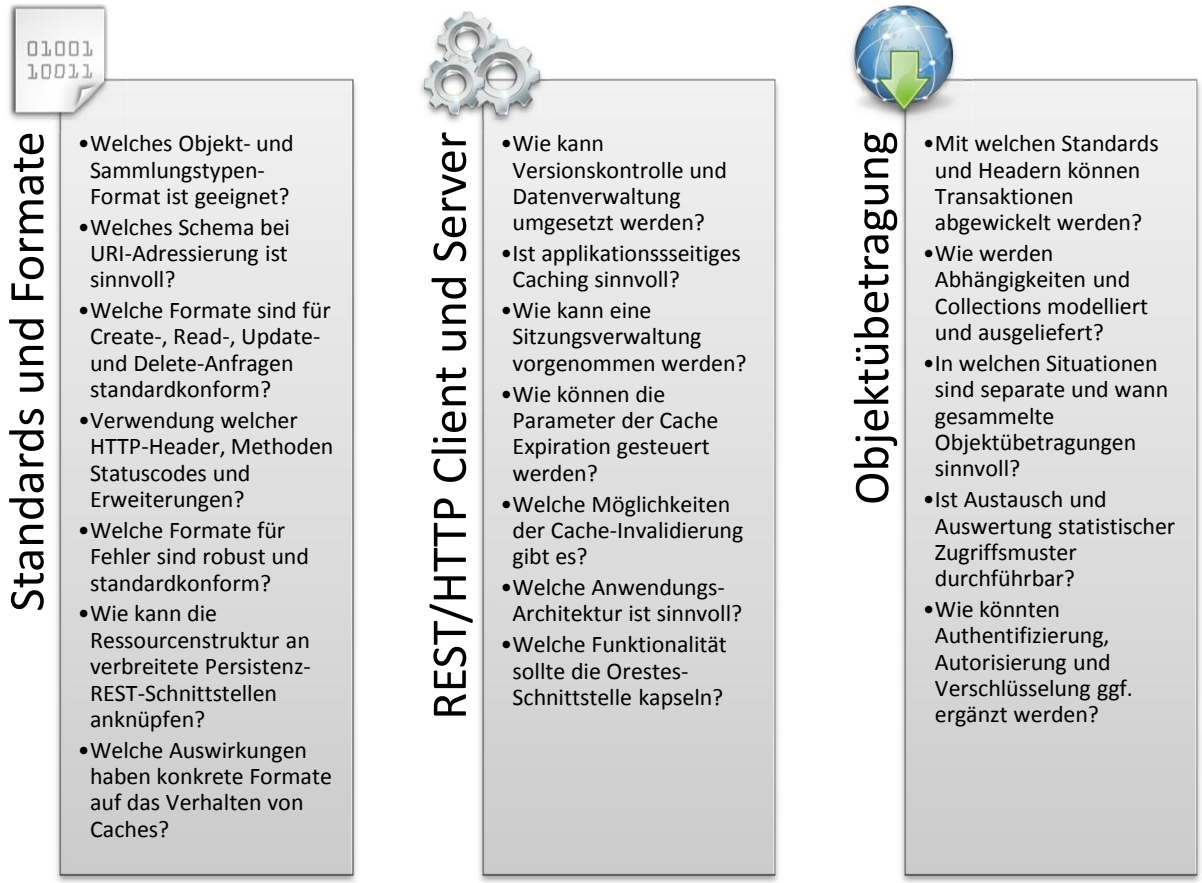


Abbildung 9 Zielsetzung: Beantwortung zentraler Fragestellungen

2 Zusammenhänge mit genutzten und verwandten Technologien

In diesem Kapitel werden wir einen Überblick über die Technologien geben, die im Rahmen des Entwurfs von Orestes relevant sind und ihren Zusammenhang mit Orestes ausarbeiten. Dabei werden wir einerseits die Grundlagen der technischen Aspekte untersuchen (HTTP, Web-Caching und OODBMS). Ebenso werden wir architekturbezogene Gesichtspunkte anhand des REST-Architekturstils und der zu Orestes verwandten Umsetzungen dieses Architekturstils aus dem Umfeld nichtrelationaler Datenbanken motivieren und erklären.

2.1 HTTP

Das *Hypertext Transfer Protocol* ist ein von der IETF (*Internet Engineering Task Force*) in RFC (*Request for Comment*) 2616 [FIG+99] spezifiziertes Protokoll der Applikationsschicht. Die erste Version von HTTP entstand 1989 am CERN, wo Tim Berners-Lee mit HTTP, HTML und URLs das technische Fundament für das WWW legte. Ziel von HTTP war es, ein einfaches, erweiterbares und schnelles Protokoll für den Austausch von Dokumenten zu sein. Der grundlegende Aufbau des Protokolls hat sich seit dieser Version kaum geändert, es wurden jedoch zahlreiche Erweiterungen vorgenommen, die inkrementell die Methoden und Header von HTTP erweiterten (siehe Abbildung 10).

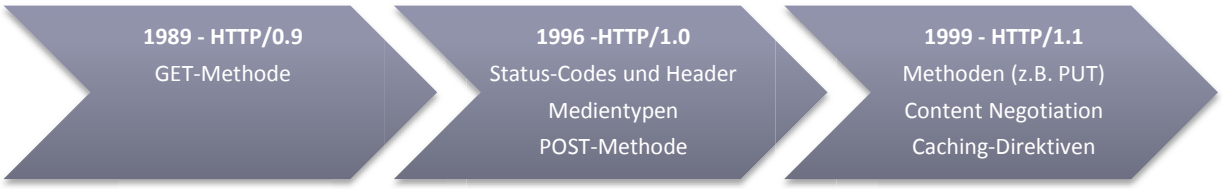


Abbildung 10 Entwicklung des HTTP-Protokolls

Die 1999 standardisierte Version ist noch immer die aktuelle, von zahllosen Browsern, Proxys, Caches und Servern implementierte Protokollversion. Grund hierfür ist einerseits der durchdachte Umfang und Aufbau von HTTP, der einfache Erweiterungsmechanismen für anwendungsspezifische Zwecke enthält (z.B. Definition neuer Header). Andererseits ist jedoch die Einführung eines neuen HTTP-Protokolls durch den Umstand seiner Verbreitung ebenso erschwert wie eine Migration von IPv4 nach IPv6. Einige diesbezügliche Bemühungen (z.B. HTTPnG, SMUX) scheiterten, so dass sich die aktuelle Weiterentwicklung von HTTP durch die HTTPbis-Working-Group der IETF derzeit auf die Beseitigung kleinerer Errata beschränkt.

2.1.1 Kommunikation

Bei HTTP handelt es sich um ein zustandsloses Client-Server-Protokoll, das in der Regel auf einem verlässlichen, verbindungsorientierten Transportprotokoll (ISO-OSI Schicht 4) wie TCP ausgeführt wird. Jede Interaktion zwischen Client und Server folgt dem Frage/Antwort-Schema, bestehend aus HTTP-Request und HTTP-Response. Der Austausch eines Request/Reponse-Paares (*Session*) ist unabhängig von der gesamten vorgehenden und folgenden

HTTP-Kommunikation. Deshalb wird HTTP als statuslos bezeichnet. Diese Statuslosigkeit ist der Motor und Garant der hohen Skalierbarkeit von HTTP. Dadurch, dass die Notwendigkeit serverseitigen Zustands entfällt, können verschiedene Server-Instanzen unabhängig voneinander agieren – Multithreading, Load-Balancing und Caching wird möglich.

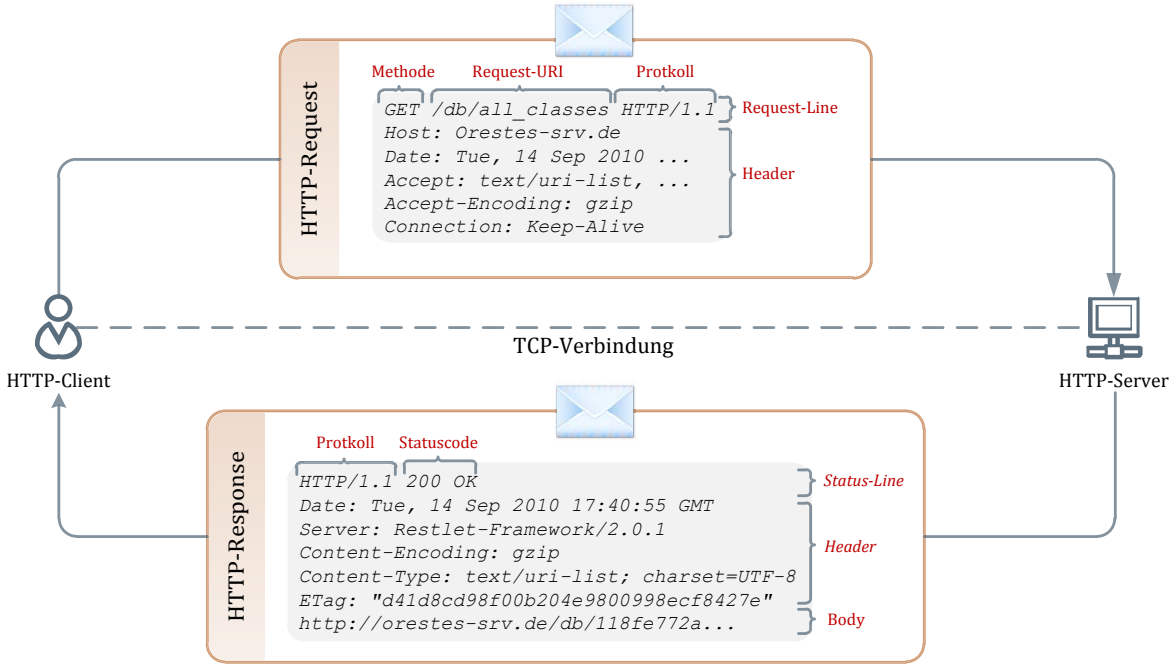


Abbildung 11 Request-Response Kommunikation bei HTTP

Die erste Zeile des Requests (*Request-Line*) enthält stets eine die Anfrage kennzeichnende Methode, eine Referenz auf die betroffene Ressource in Form eines Uniform Resource Identifiers (URI) und die verwendete Protokollversion. Die gleichermaßen festgelegte erste Zeile des Response (*Status-Line*) enthält ebenfalls die Protokollversion und zusätzlich einen Statuscode. Dieser Statuscode gibt über die Art des Ergebnisses Auskunft (z.B. Erfolg, Weiterleitung, Misserfolg). Die auf diese Zeilen folgenden Message-Header lassen sich in unterschiedliche Kategorien einteilen:

General Header

Können sowohl in Anfrage als auch Antwort verwendet werden und beziehen sich nicht auf den Message-Body (z.B. `Date` – Datum des Versands, `Cache-Control` – Anweisungen an Caching-Systeme).

Entity Header

Geben Auskunft über die Art des Message-Bodys (*Entity*), der sowohl in einer Anfrage, als auch einer Antwort enthalten sein kann (z.B. `Content-Encoding` – Codierung der Entity, `ETag` – Versionsnummer der Entity).

Request Header

Enthalten Informationen, die sich ausschließlich auf die Anfrage beziehen (z.B. `Accept` – verarbeitbare Medientypen, `User-Agent` – Clientanwendung).

Response Header
Enthalten Informationen, die sich ausschließlich auf die Antwort beziehen (z.B. `WWW-Authenticate` – Authentifizierung erwartet, `Retry-After` – Dauer vorübergehender Nichtverfügbarkeit der angefragten Ressource).

Die Menge aller in diesen Kategorien verwendbaren Header sind spezifiziert. Einige dieser Header sind explizit erweiterbar, so kann beispielsweise im `WWW-Authenticate` ein neues Authentifizierungsschema festgelegt werden. Zudem können Erweiterungen und neue Header, die für den öffentlichen Gebrauch bestimmt sind bei der IANA (Internet Assigned Numbers Association) registriert werden. Für neue, nicht-standardisierte, oder nur intern verwendete Header hat sich die Notation `X-My-New-Header` eingebürgert (X: *experimental*).

Methoden eines Requests (z.B. GET), die kennzeichnend für den gesamten Kommunikationsakt zwischen Client und Server sind, können zwei Prädikate besitzen: sicher (*safe)* und *idempotent*. Sichere Methoden sind solche, die keine unerwünschten oder unerwarteten Nebeneffekte erzeugen (z.B. eine einfache Abfrage). Idempotente Methoden erzeugen unabhängig davon, ob sie ein oder mehrmals ausgeführt werden stets die gleichen Nebeneffekte (z.B. die Entstehung einer neuen Ressource) ($request(uri) = request(request(uri))$).

Methode	*Safe*	*Idempotent*	Beschreibung
GET	x	x	Dient der Abfrage einer durch die *Request-URI* angegebenen Ressource. Durch `Range`-Header können auch Fragmente ausgeliefert werden.
HEAD	x	x	Gleicht einer GET-Abfrage, bei der vom Server kein Body übertragen wird.
POST			Dient dem Client um Daten an den Server zu übermitteln. Hierbei enthält der Request-Body die Daten.
PUT		x	Erzeugt oder ändert eine neue Ressource an der durch die *Request-URI* angegebenen Adresse.
PATCH			Diese noch im Standardisierungsprozess begriffene Methode dient der Durchführung von partiellen Updates, deren Änderungsanweisungen im Request-Body enthalten sind.
BATCH			Dient (sobald standardisiert) der Durchführung von Mehrfachupdates in einer Anfrage.
DELETE		x	Löscht eine Ressource, wobei die Löschung i.d.R. nicht instantan sondern asynchron erfolgt.
CONNECT			Stellt eine Verbindung zu einem speziellen Proxy her (beispielsweise zum SSL-Proxying).
OPTIONS	x	x	Fragt die möglichen Kommunikationsparameter des Servers oder einer Ressource ab (z.B. unterstützte Methoden).
TRACE		x	Fragt zu Debuggingzwecken eine Kopie des übermittelten Requests an (um z.B. Manipulationen durch Zwischensysteme zu entdecken).

Tabelle 1 Methoden von HTTP

Die Wahl der Methode zur Abbildung einer Operation spielt eine große Rolle für die Implementierung einer HTTP-Schnittstelle. Damit die GET-Methode *idemptotent* und *safe* ist, darf sie nur Abfragen abbilden. Seiteneffektbehaftete Operationen wie z.B. `GET /delete?id=1` stellen somit einer Verletzung der GET-Semantik dar (*unsafe Operation*). Ist eine Methode safe, so erlaubt sie dem Client also beliebigen Referenzen (URIs) seiteneffektfrei zu folgen. Idempotente Methoden erlauben dem Client eine Anfrage erneut auszuführen, z.B. bei Ausbleiben einer Antwort des Servers.

Statt einer Methode enthält die Antwort des Servers stets einen Statuscode und eine natürlichsprachliche Beschreibung dieses Codes (z.B. `201Created`). Die Statuscodes sind in fünf verschiedene Kategorien eingeteilt (siehe Tabelle 2), die das Ergebnis des Requests näher klassifizieren.

Kategorie	Beispiele	Beschreibung
1xx Informational	100 *continue* 101 *switching protocols*	Gibt keine Auskunft über Erfolg oder Misserfolg, sondern dient ausschließlich der Bestätigung eines Protokollwechsels (`Upgrade`-Header) oder der Ausführbarkeit einer Anfrage.
2xx Succesful	200 *OK* 201 *Created*	Bestätigt den Erfolg der ausgeführten Methode und expliziert diesen ggf. (→*Created*).
3xx Redirection	301 *Moved Permanently* 307 *Temporary Redirect*	Informiert den Client darüber, dass eine Umleitung (*Redirection*) erfolgen muss auf eine Ressource, die entweder im `Location`-Header oder im Message-Body angegeben ist.
4xx Client Error	404 *Not Found* 405 *Method Not Allowed*	Gibt an, dass nach Sicht des Servers ein clientseitiger Fehler aufgetreten ist (z.B. Abfrage einer nicht existierenden Ressource oder Syntaxfehler)
5xx Server Error	500 *Internal Server Error*	Gibt an, dass die Verarbeitung des Requests einen serverseitigen Fehler produziert hat.

Tabelle 2 Kategorien von Statuscodes bei HTTP

Nach einer durchgeführten Anfrage wird der Client also stets durch die Überprüfung des Statuscodes evaluieren, welche Folgeoperationen sinnvoll sind und ob der Vorgang erfolgreich war. Dies ist in Abbildung 12 an einem Beispiel demonstriert:

1. Der Client fragt per GET ein Dokument unter der relativen URI */dokumente/1*, für den virtuellen Host *example.com* ab. Dieser Host wird durch die Netzwerkbibliothek des Clients über DNS abgefragt und eine TCP-Verbindung über Port 80 initiiert. In den HTTP-Request wird das einzige Request-Header-Pflichtfeld `Host` eingetragen. Der Message-Body des Requests bleibt leer, da es sich um eine GET-Anfrage handelt.
2. Der Server empfängt die Anfrage, ruft die Ressource erfolgreich ab und schickt sie über die TCP-Verbindung mit einem Erfolgs-Statuscode zum Client.
3. Der Client erkennt den Erfolg seiner GET-Anfrage und löscht nun mit einer DELETE-Anfrage die Ressource.

4. Der Server bestätigt den Erfolg der Operation mit einem anderen Erfolgs-Statuscode, der impliziert, dass kein Message-Body in der Server-Antwort enthalten ist.

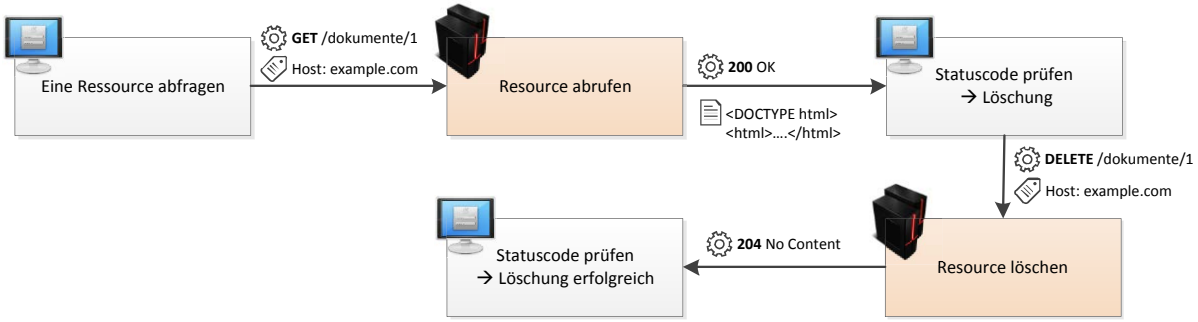


Abbildung 12 HTTP-Methoden und Statuscodes an einem Beispiel

Das geschilderte Beispiel betrachtet dabei zwei Parteien: den *Client* und den *Origin Server*. Der Origin Server stellt den Ort der eigentlichen Anfrageverarbeitung dar. Es können allerdings sogenannte Zwischensysteme (*Intermediaries*) an der Kommunikation beteiligt sein. Man unterscheidet bei HTTP zwischen:

Proxy
Diese nehmen in einer Position zwischen Client und Server selbst eine Zwitterrolle aus Client (gegenüber dem Origin Server) und Server (gegenüber dem Client) ein. Zu ihrem Einsatzweck zählt primär das Caching – sie können unter gewissen, regulierbaren Umständen Anfragen ohne Kommunikation mit dem Origin Server beantworten. Des Weiteren können sie Aufgaben wie Content-Filterung, Anonymisierung, Zugriffskontrolle, Content-Routing und Transcoding (Konvertierung von Inhaltsformaten) wahrnehmen.

Tunnel
Ein Tunnel agiert lediglich weiterleitend und verändert die ausgetauschten HTTP-Requests und Responses nicht. So können beispielweise SSL-Proxys über die CONNECT Methode in einen Tunnel umgewandelt werden, der transparent die Verschlüsselung auf Transportebene vornimmt und Anfrage und Antwort unverändert übermittelt.

Gateway
Ein HTTP-Gateway arbeitet wie ein Proxy-Server, der allerdings aus Sicht des Clients an die Stelle eines Origin Servers tritt. Der Gateway kann dabei Aufgaben von Anbindung verschiedener Protokollstacks bis Load-Balancing wahrnehmen.

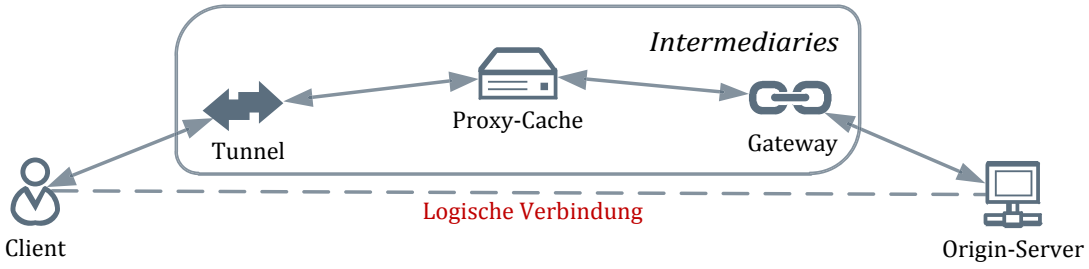


Abbildung 13 Intermediaries bei HTTP

Intermediaries sind auch für den Origin Server meist unsichtbar. Da die HTTP-Kommunikation aber stark von Parametern wie Standardkonformität und Caching-Funktionalität der beteiligten Zwischensysteme abhängt, setzen diese einen `Via`-Header, durch den der Origin Server Kenntnis von beteiligten Zwischensystemen erhält. Da die Verbindung auf der Transportschicht des Netzwerks stets zwischen benachebarten Zwischensystemen besteht (siehe Beispiel in Abbildung 13), werden zudem häufig zusätzlich Header übertragen, um die Identität des Clients zu transportieren, z.B. in Form seiner IP-Adresse.

Aufbauend auf diesen Grundlagen werden wir im Folgenden einige Aspekte von HTTP untersuchen, die für die Konzeption von Orestes eine Rolle spielen.

2.1.2 Ressourcen, Repräsentationen, Adressierung

Als Ressourcen werden in HTTP identifizierbare Konzepte und Informationen bezeichnet. Eine Ressource liegt dabei in einer oder mehreren Repräsentationen vor. Oft sind die Übergänge zwischen Ressource und Repräsentation fließend. So ist beispielsweise schwer differenzierbar, ob eine Visitenkarte und ein Bild zwei Repräsentationen einer Person, oder verschiedene Ressourcen sind.

Ressourcen werden in HTTP durch URIs (Uniform Resource Identifier) identifiziert. URI stellt die Obermenge von URLs (Uniform Resource Locator) und URNs (Uniform Resource Names) dar. Diese setzen sich aus den in Abbildung 13 gezeigten Bestandteilen zusammen.

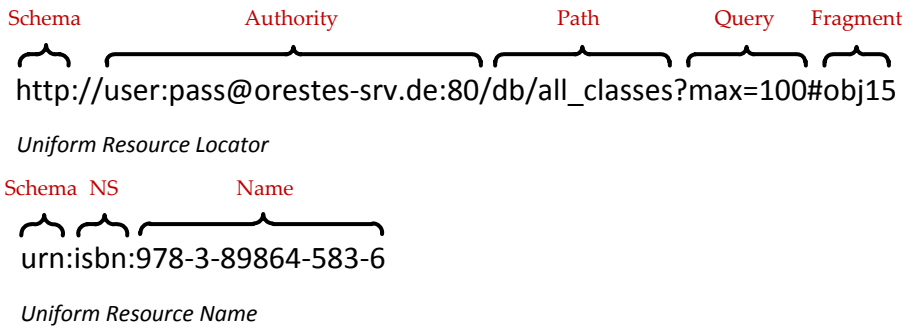


Abbildung 14 URL und URN

Eine URL unterscheidet sich von einer URN dadurch, dass sie dereferenziert werden kann, d.h. neben der Identifikation umfasst die URL auch die Lokalisation einer Ressource. Die Unterscheidung zwischen beiden ist jedoch lediglich von historischer Bedeutung [Til09, S.37], weshalb wir im Folgenden stets den Begriff URI verwenden werden. Bei der Konzeption von Orestes werden wir ausschließlich dereferenzierbare URIs verwenden, da URNs in der Theorie zwar langlebiger sind, aber einen erhöhten Aufwand für die Namensauflösung durch *URN Resolver* und *Resolver Discovery Services* bedeuten [MeSa04, S. 732].

Den auf die Authority folgenden Teil der URI werden wir verwenden, um die dauerhafte Identifikation von Ressourcen (Objekten, Klassen, etc.) zu erlauben. Der von Tim Berners-

Lee geprägte Satz „Cool URIs don't change" [Til09, S.46] macht noch immer ein wichtiges Designziel der Ressourcenidentifikation aus. Die Erweiterungen des URI-Standards, um IRIs (International Resource Identifier) mit erweitertem, internationalisiertem Zeichensatz für URIs oder XRIs (Extended Resource Identifier) mit der Trennung von Referenz und Benennung, werden wir aufgrund ihrer fehlenden Relevanz nicht beachten.

Die Struktur von URIs für bestimmte Ressourcen werden wir im Folgenden durch *URI Templates* [Gre10] beschreiben. Sie erlauben es, durch einen generischen Ausdruck mithilfe von Variablenexpansion Gruppen von URIs durch ein einheitliches Format zu definieren. Bei der Verwendung syntaktischer Element von URI-Templates beschränken wir uns auf einfache Variablen: das URI-Template „http://orestes-srv.de/db/{Namespace}/{Class}" enthält die Variablen *Namespace* und *Class* und ist mithin ein allgemeiner Ausdruck für eine Anzahl spezieller URIs.

2.1.3 Content Negotiation

Ressourcen können auf einem Server als unterschiedliche Varianten (*Repräsentationen*) angeboten werden. HTTP stellt deshalb einen als Content-Negotiation bezeichneten Mechanismus bereit, um eine gewünschte Repräsentation dynamisch auszuhandeln. Dabei kann es sich um *sprachspezifische* (z.B. `de-DE` oder `en-US`), *qualitätsspezifische* (z.B. `png-8` oder `png-24`), *codierungsspezifische* (z.B. `gzip` oder `deflate`) oder *formatspezifische* (z.B. `Atom-Feed` oder `JSON`) Varianten handeln. Die Aushandlung kann auf drei Arten geschehen:

Server-Driven Negotiation
Hier entscheidet der Server über eine geeignete Wahl der Repräsentation. Dabei bedient sich der Server bestimmter vom Client übermittelter Header-Felder wie `Accept` – gewünschte Medientypen, `Accept-Charset` – verwendeter Zeichensatz, `Accept-Encondig` – Codierung, `Accept-Language` – Sprache. Über die Angabe dieser Header kann der Client die einzelnen erwünschten Eigenschaften der Repräsentation mit quantitativen Präferenzen versehen. Die konkrete Auswahl der Variante trifft der Server dann in bestmöglicher Konformität zu diesen Präferenzen.

Agent-Driven Negotiation
Auf eine Anfrage des Clients antwortet der Server bei der Agent-Driven Negotiation zunächst mit einer Auflistung der verfügbaren Varianten der angefragten Ressource. Anschließend wird die vom Client ausgewählte Variante in einer zweiten Anfrage abgerufen. Da HTTP für die Darstellung der Alternativen kein standardisiertes Format und keine Header vorsieht und zudem die Antwort auf eine Anfrage um die Dauer einer Round-Trip-Time verlängert wird, ist Agent-Driven Negotiation nur in Kombination mit ausgeklügeltem Caching relevant.

Transparent Negotiation
Hierbei nimmt ein Proxy die Aushandlung der Repräsentation mit dem Origin Server wahr (Agent-Driven Negotiation) und liefert diese dem Client aus (Server-Driven Negotiation).

Für Orestes werden wir zunächst eine Form der Server-Driven Negotiation wählen, da diese performant durchführbar (kein zusätzlichen Latenzzeiten) ist und ohne nicht-standardisierte Verfahren auskommt, sich also agnostisch bezüglich der Infrastruktur verhält.

2.1.4 Medientypen und Codierung

Die Kennzeichnung bestimmter Inhaltsformate geschieht in HTTP durch sogenannte MIME-Types (Multipurpose Internet Mail Extensions), meist lediglich als *Mediatypes* bezeichnet. Diese Medientypen sind nach Einsatzzweck und Name gruppiert und registriert. Für Orestes wird es nötig sein Medientypen zu transportieren, die aufgrund fehlender Erfordernis zuvor nicht für den Einsatz im Web registriert wurden (z.B. proprietäre Queryformate). Dafür bestehen zwei Möglichkeiten: *Vendor-specific Mediatypes*, und *experimental Mediatypes*. Abbildung 15 zeigt Beispiele für Medientypen.

Medientyp	Format	Referenz
text/html	Inhalt als HTML beliebiger Version	HTML 4.01
application/atom+xml	Feeds im XML Format mit Atom-Semantik	RFC 5023
multipart/mixed	Erlaubt die gebündelte Übertragung von Repräsentationen unterschiedlicher Medientypen mit jeweiligen `Content-Encoding`-Headern	RFC 2046
application/x-orestes-query	Ein erfundener *experimental Mediatype* des *Application*-Einsatzzweiges	*inoffizieller Charakter*
application/vnd.versant.vql	Ein ebenfalls erfundener Mediatype für einen *vendor-specific* (→vnd) *Mediatype* (hier: Query)	

Abbildung 15 Beispiele für Medientypen

Langfristig ist aus Gründen der Interoperabilität eine Standardisierung neuer Orestes-Medientypen nach RFC 4288 sinnvoll. Allerdings ist dies ein bürokratisch anspruchsvoller Prozess, von dem wir deshalb vorerst absehen.

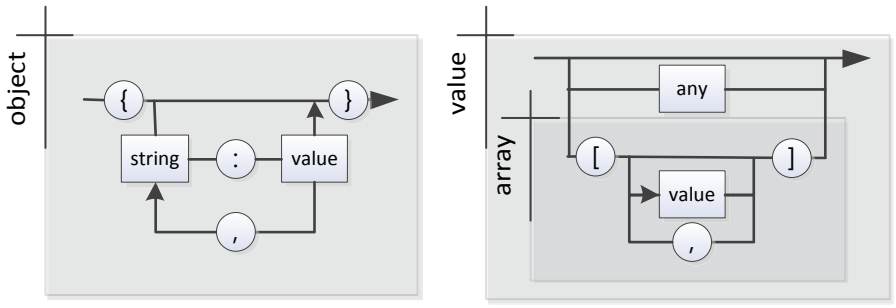


Abbildung 16 Das JSON-Format als Syntaxdiagramm

Medientypen die innerhalb des Orestes-Systems eine Option zur Datenübertragung sind, werden in Abbildung 17 vorgestellt. Wir werden die Implementierung von Orestes so gestalten, dass einerseits alle Formate durch Mechanismen der *Content-Negotiation* potentiell einsetzbar und austauschbar sind und andererseits alle notwendigen Datenstrukturen (Objekte, Klassenschemata, etc.) bereits als definierte JSON-Darstellung vorliegen. Das simpel gestaltete JSON-Format ist als Syntaxdiagramm in Abbildung 16 gezeigt. Es besteht ausschließlich

aus Objekten, Arrays und primitiven Typen. Die bestehenden Repräsentationsformate können durch zusätzliche (z.B. herstellerspezifische, proprietäre) Formate ergänzt oder ersetzt werden. So wird es beispielsweise möglich sein, sowohl mit einem Webbrowser auf ein Objekt zuzugreifen (HTML), als auch mit einem Orestes-Client einer bestimmten Programmiersprache (JSON).

JSON - *JavaScript Object Notation*	JSON ist ein leichtgewichtiges, schemaloses Format, bestehend aus Schlüssel-Wert-Paaren, Skalaren und Listen, die in beliebiger Tiefe ineinander verschachtelt werden können. Wie wir beschreiben werden, besitzt JSON bereits einen hohen Verbreitungsgrad unter modernen nichtrelationalen Datenbanken.
YAML - *YAML Ain't Markup Language*	YAML, entwickelt als Auszeichnungssprache zur Objektserialisierung und Datenrepäsentation hat einen starken Fokus auf menschlicher Lesbarkeit und einfacher Struktur.
XML - *Extentsible Markup Language*	XML umschließt als verbreitete Markup-Sprache des W3C viele untersütztende (z.B *DTD, XMLSchema, RelaxNG* - Schemadefinition; *XPath, XQuery* - Abfragen, *XSLT, STX* - Transformationen) und abgeleitete (z.B *Atom, XHTML, SOAP*, *SVG*) Technologien.
HTML - *Hypertext Markup Language*	Verwendet als *Microformat* kann HTML nicht nur formatierte Webdokumente abbilden, sondern auch Semantik als maschinenverarbeitbare Metadaten. So kann in einem Format eine menschen- und maschinenlesbare Repräsentation von Daten transportiert werden.
RDF - *Resource Description Framework*	RDF definiert ein aus Tripeln bestehendes Datenmodell, bei dem stets Subjekt-Ressourcen durch eine Relation (Prädikat) mit einer Objekt-Ressource verknüpft sind. Die so modellierten Daten können in verschiedenen Formaten (z.B. XML) dargestellt und über eine Querysprache (*SPARQL*) abgefragt werden.

Abbildung 17 Medientypen für Übertragungsformate von Orestes

Repräsentationen eines bestimmten Medientyps können in HTTP nicht nur als Ganzes abgefragt und generiert werden. Es können, falls durch den Sever explizit erlaubt, auch Dokumentfragmente (*Byte-Ranges*) angefragt werden. Auch kann der Server bei der Generierung der Antwort eine paketierte Codierung (*Chunked Encoding*) nutzen, bei der die Antwort vom Server nicht zusammenhängend, sondern aufgeteilt in Pakete bekannter Größe (*chunks*) übertragen wird. So kann der Server bei dynamisch generiertem Inhalt bereits mit der Übertragung beginnen, bevor der sonst verpflichtend anzugebene Inhalt des `Content-Length`-Headers bekannt ist. Weitere Repräsentationsmetadaten sind neben der Länge, die durch Content-Negotiation aushandelbaren Eigenschaften Codierung, Sprache, Medientyp und zusätzlich ein Hashwert (zur Überprüfung der Unversehrtheit) oder eine alternative URI der zugrundeliegenden Ressource. Bei der Codierung unterscheidet man bei HTTP insgesamt zwischen drei Formen der Codierung: dem Medientyp, seiner Codierung als Entity (`Content-Encoding`, z.B. für Komprimierung) und jener der Nachricht (`Transfer-Encoding`, z.B. paketierte Codierung)

Bei der Implementierung von Orestes werden wir *Byte-Ranges* und *chunked Encoding* aus Performanzgründen nutzen. Ein Orestes Client kann auf diese Weise Ressourcenfragmente ab-

fragen und profitiert von der verringerten Latenzzeit des HTTP-Response durch paketierte Codierung.

2.1.5 Sicherheit

Als Protokoll der Applikationsschicht kann HTTP auf einer verschlüsselnden Transportebene, wie TLS (Transport Layer Security) ausgeführt werden. Hierbei läuft der Prozess der Authentifizierung und Verschlüsselung transparent für die HTTP-Schicht ab. Es verlieren dadurch allerdings alle Intermediaries auf der verschlüsselten Transportstrecke die Fähigkeit, die HTTP-Kommunikation lesend oder schreibend zu verfolgen und deshalb auch ihre potentielle Funktion als Cache.

HTTP bietet jedoch selbst einen Mechanismus um Benutzer durch den Server zu *authentifizieren*. Ein Benutzer *authentisiert* sich dabei durch Benutzername und Passwort für einen bestimmten, benannten Bereich (*Realm*) für dessen Nutzung er *autorisiert* ist. Abbildung 18 demonstriert dies an einem Beispiel:

1. Ein Client greift auf eine geschützte Ressource zu
2. Der Server weist den nicht-authentifizierten Zugriff zurück und informiert den Client über das zu verwendende Authentifikationsverfahren und den *Realm*
3. Der Client identifiziert und authentisiert sich beim Zugriff durch die Übermittlung der durch Authentifikationsverfahren geforderten Daten (die Autorisierung)
4. Der Server liefert die Ressource aus und erwartet für jeden Folgezugriff die erneute Übermittlung des Autorisierungsheaders (→Zustandslosigkeit)

Abbildung 18 Authentifizierung eines Clients über HTTP

Für die Authentifizierung sieht HTTP zwei standardisierte Verfahren vor [FNH+99]:

Basic Authentication
Bei der Basic Authentication werden Benutzername und Passwort konkateniert und Base64-codiert übertragen. Da die Base64-Codierung reversibel ist kann das Passwort durch einen Angreifer ausgelesen werden. Des Weiteren ist die Identität des Servers nicht sichergestellt, das Passwort könnte also direkt zu einem Angreifer geschickt werden. Auf dem Weg vom Client zum Server könnte außerdem die Nachricht durch einen Angreifer verändert („Man-in-the-middle Attacke“) oder öfter verschickt werden („Replay Attacke“).

Digest Authentication
Um zu verhindern, dass Passwörter im Klartext übertragen werden, wird bei dem Digest-Verfahren mit einer kryptographischen Hashfunktion (z.B. MD5 oder SHA) ein *Digest* berechnet in dessen Berechnung Username, Passwort und ein vom Server übermitteltes zufälliges Token (*Nonce*) eingehen. Aus dem berechneten Digest kann weder die ursprüngliche Information berechnet werden, noch kann (mit durchführbarem Aufwand) ein gleicher Di-

gest aus anderen Ursprungsinformationen berechnet werden. Dadurch wird das Mitlesen von Passwörtern und Replay Attacken verhindert, gegen das Mitlesen der übertragenen Daten und das Verfälschen von Nachrichten bietet jedoch auch das Digest-Verfahren keinen Schutz.

Ein weiterer in der Praxis häufig angewendeter Mechanismus zur Authentifizierung ist die *Cookie-Authentifizierung*. Bei Cookies handelt es sich um Applikationszustand der durch den `Set-Cookie`-Header vom Server zum Client und bei jeder Folgeanfrage durch den `Cookie`-Header vom Client zum Server transportiert wird. Sie stellen einerseits eine bewusste Verletzung der Zustandslosigkeit dar und verhindern zudem die Cachebarkeit von Antworten, weshalb wir vollständig von der Verwendung von Cookies absehen.

Da also sowohl das Basic- als auch das Digest-Verfahren keine Unterstützung für die Sicherstellung der Integrität einer Nachricht und ihre Unlesbarkeit für Angreifer garantieren, können zusätzliche Verfahren angewendet werden. Um die erstere Beschränkung aufzuheben wird in der Regel *HMAC (Keyed-Hashing for Message Authentication*) eingesetzt. Dabei wird ein Shared Secret von Client und Server benutzt, um mit einer kryptographischen Hashfunktion eine Nachrichtensignatur zu erzeugen, die entweder als Teil des Message-Bodys oder als Erweiterung des `Authorization`-Headers übermittelt wird. Um neben Man-in-the-middle Attacken auch das unerwünschte Mitlesen des Message-Bodys zu verhindern, können die übertragenen Nutzdaten verschlüsselt werden. So kann beispielsweise ein zu übertragendes XML-Dokument mithilfe des XML-Encryption-Standards auf Element- oder Dokumentebene verschlüsselt und als HTTP-Nutzdaten übertagen werden.

Bei der Umsetzung von Orestes werden wir sowohl das Basic- als auch das Digest-Verfahren zur Authentifizierung anbieten. Die Architektur wird es zudem erlauben, auch verschlüsselte und signierte Repräsentationen durch benutzerdefinierte Format-Converter zu erzeugen und zurückzuverwandeln.

2.2 REST

REST (*Representational State Transfer*) ist ein Architekturstil, der im Jahr 2000 in einer Dissertation von Roy Thomas Fielding beschrieben wurde [Fie00]. Dieser Architekturstil ist eine a posteriori Erklärung für den Erfolg des World Wide Webs, an dessen Entwicklung Fielding als ein Autor von HTTP beteiligt ist. Mit der Dissertation abstrahierte Fielding von der konkreten Architektur von HTTP und erklärte die zugrundeliegenden Konzepte – unabhängig vom konkreten Protokolldesign und der eingegangenen Kompromisse. Grundsätzlich ist es möglich, eine REST-konforme (*RESTful*) Architektur auch auf Basis eines beliebigen Protokollstapels und einer anderen Infrastruktur als der des Webs zu konstruieren. REST beschreibt dazu eine Menge von Constraints (Randbedingungen), die eine solche Architektur erfüllen muss. Da jedoch in Form des HTTP- und URI-Standards bereits das einsatzfähige, REST-konforme Instrumentarium zur Errichtung einer solchen Architektur bereitsteht, ist REST bisher stets in einem solchen Kontext anzutreffen [Til09, S.8]. Wir werden im Folgenden die schrittweise Ableitung der Constraints gemäß [Fie00, K. 5] nachvollziehen und erklären, um anschließend auf eine kompakte Beschreibung des REST-Architekturstils in Form einfacher Richtlinien zu schließen. Diese Richtlinien werden in Orestes umgesetzt.

2.2.1 Ableitung durch Constraints

Das Prädikat *RESTful* bezeichnet die Einhaltung bestimmter Constraints, die in Abbildung 19 zusammengefasst sind.

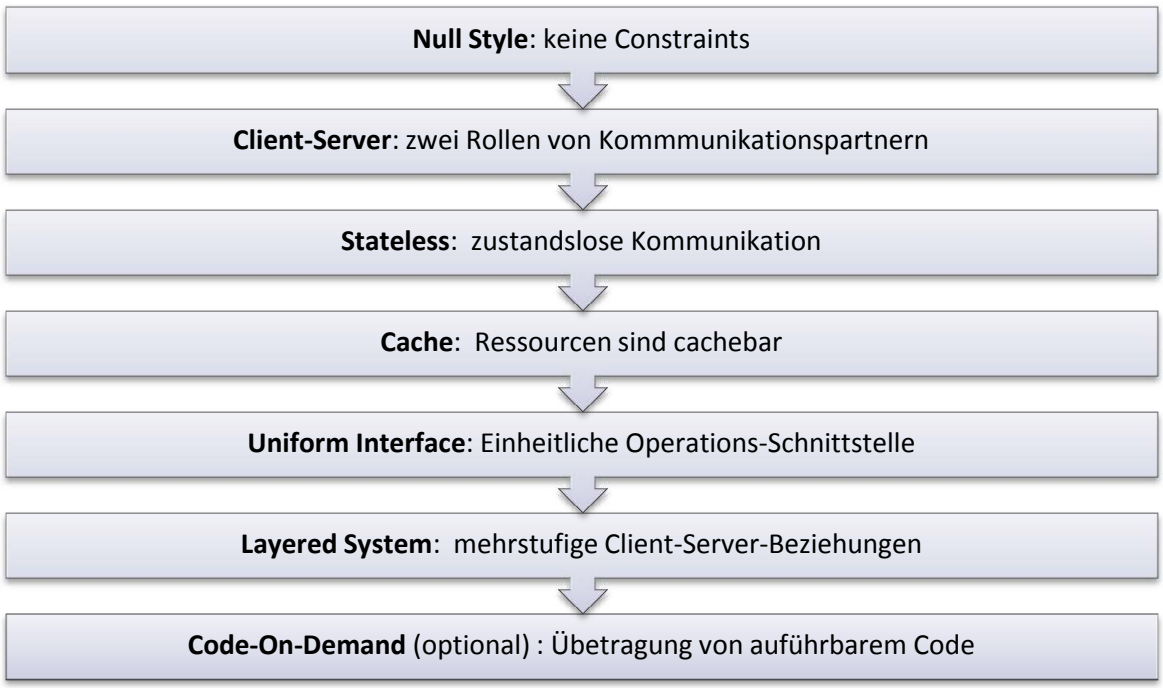


Abbildung 19 Constraints von REST

Ausgehend von dem sogenannten *Null Style* - einem Architekturstil, der keine Randbedingungen an die Ausgestaltung und Struktur einer Architektur stellt - werden schrittweise Randbedingungen eingeführt. Motivation für das Constraint-Set von REST ist, dass eine Ar-

chitektur, die alle verpflichtenden Constraints einhält, wünschenswerte emergente Eigenschaften vorweist, wie „performance, scalability, simplicity, modifiability, visibility, portability“ [Lac07].

Der erste Constraint ist der **Client-Server** Architekturstil. Prinzip des Client-Server Constraints ist „separation of concerns“ - die Trennung des User-Interface von Datenhaltung und Logik durch eine festgelegte Schnittstelle. Durch die Entkopplung wird einerseits der Client portierbar und andererseits erhöht die Entlastung des Servers seine Skalierbarkeit. So können sich Client und Server unabhängig voneinander weiterentwickeln, wenn die Schnittstelle (*Contract*) zwischen ihnen weiterhin eingehalten wird.

Dem hinzugefügt wird der **Stateless**-Constraint, der vorschreibt, dass die Client-Server-Kommunikation zustandslos ist. Das bedeutet, dass jede Anfrage vom Client alle für die Verarbeitung notwendigen Informationen enthält und keinen Bezug auf einen auf dem Server gespeicherten Kontext nehmen kann. Trotz der offensichtlichen Erhöhung der Skalierbarkeit handelt es sich um einen Trade-off: die Netzwerklast kann durch repetitive Daten belastet werden und notwendiger Serverzustand muss explizit in den Zustand von Ressourcen überführt werden.

Um diese potentielle Gefährdung der Netzwerkeffizienz zu vermeiden führt Fielding den **Cache**-Constraint ein. Dieser schreibt vor, dass Daten in einer Serverantwort implizit oder explizit als cachebar oder uncachebar markiert sind. Dies erlaubt Clients und Zwischensystemen Antworten zwischenzuspeichern. So kann bei guter Umsetzung die Anzahl der Client-Server-Interaktionen vermindert und die Skalierbarkeit und Performanz weiter erhöht werden.

Das zentrale Unterscheidungsmerkmal zwischen REST und anderen netzwerkbasierten Architekturstilen ist die Einhaltung des **Uniform-Interface**-Constraints. Diese einheitliche Schnittstelle trennt Client und Server unabhängig von den individuellen Anwendungsbedürfnissen und entkoppelt die angebotenen Server-Dienste (*Services*) von Implementationen. Das Uniform-Interface selbst ist ebenfalls an Constraints gebunden:

- **„Identification of resources“**: Ressourcen werden in jedem Request identifiziert (z.B. über URIs) und in Form von Repräsentationen an den Client übermittelt.
- **„Manipulation of resources through representations“**: Ein Client kann mit der Repräsentation einer Ressource und ihren Metadaten diese Ressource auf dem Server manipulieren oder löschen (Berechtigung vorausgesetzt).
- **„Self-descriptive messages“**: Jede Nachricht enthält ausreichend Informationen um die Art der notwendigen Verarbeitung abzuleiten (z.B. den Medientyp der Repräsentation oder seine Cachebarkeit).
- **„Hypermedia as the engine of application state“**: Abhängig vom Applikationszustand beschreibt eine durch den Server ausgelieferte Repräsentation alle durch den Client ausführbaren Aktionen als Hyperlinks auf andere Ressourcen.

Die Skalierbarkeit wird weiter erhöht durch die Einführung des **Layered-System**-Constraints. In einer derart geschichteten Architektur kann ein Client jeweils nur einen unmittelbar benachebarten Server sehen, so dass eine Hierarchie von Client-Server Beziehungen entsteht. Das Kontextwissen einer Komponente (Client oder Server) ist also beschränkt auf die Hierarchieebene auf der sie sich befindet. Es entsteht so die Möglichkeit, die für HTTP geschilderten Indirektionsketten zwischen Intermediaries zu errichten, die z.B. als Shared Caches oder Load-Balancer positiven Einfluss auf die Qualitätseigenschaften der Architektur nehmen können.

Der einzige optionale Constraint von REST, ist **Code-On-Demand**, die Möglichkeit die Funktionalität des Clients durch den Download von ausführbarem Code (z.B. einem JavaScript) zu erweitern. Das scheinbare Oxymoron, dass eine Randbedingung optional ist erklärt Fielding so, dass eine Architektur ein gewünschtes Verhalten unterstützt, aber in Abhängigkeit vom Kontext auch das Fehlen des Verhaltens toleriert. So könnte sich beispielsweise die Bereitschaft von Clients zum Download von ausführbarem Code auf den Kontext „Firmenintranet“ beschränken.

Bei Einhaltung der sechs beschriebenen, verpflichtenden Constraints kann eine Architektur also als RESTful bezeichnet werden. Es wird dabei deutlich, dass auf Basis von HTTP alle geschilderten Bedingungen direkt umsetzbar sind, da sie konzeptuell bereits ausnahmslos in HTTP verankert sind. Bei Architekturen die REST auf Basis von HTTP umsetzen – wie auch unsere Orestes Implementierung – wird deshalb der Begriff RESTful HTTP verwendet. Wichtig dabei ist, dass keineswegs jede auf HTTP basierende Schnittstelle zwangsläufig auch den REST-Prinzipien genügt. Zahlreiche Verletzungen sind möglich, Fielding selbst definiert einige Kriterien, die kennzeichnen, wann eine Architektur als RESTful zu bezeichnen ist [Fie08]:

- Eine REST-API sollte keine Veränderungen an dem zugrundeliegendem Kommunikationsprotokoll vornehmen. So ist beispielweise eine Neudefinition der Semantik der HTTP-GET-Methode zu einer Löschoperation unzulässig. Änderungen, die Standardisierungslücken beheben (z.B. die PATCH-Methode) sind zulässig.
- In einer REST Architektur müssen Medientypen zum Einsatz kommen, die Applikationszustand durch Hypermedia abbilden können.
- Der Server sollte die Benennung und Hierarchie von Ressourcen ändern können, ohne dass Client-Implementierungen angepasst werden müssen. Ein Client, der alle verwendeten Medientypen verarbeiten kann, sollte ausgehend von einer „Einstiegsressource“ andere Ressourcen dynamisch entdecken können.
- Ressourcen haben, im Gegensatz zu Repräsentationen, keinen Typ. Die Bedeutung eines Medientyps sollte nicht von der Ressource abhängen deren Repräsentation er darstellt, sondern eine globale, generische Semantik besitzen (Vermeidung impliziter, domänenspezifischer Konventionen, die enge Kopplung hervorrufen).

2.2.2 Zusammenfassung als REST-Prinzipien

In Anlehnung an [Til09] lässt sich RESTful HTTP auf Basis der vorangegangenen Ableitung durch sechs Kernprinzipien zusammenfassen (Abbildung 20).

Eindeutig identifizierbare Ressourcen

- Die Instanzen der Abstraktionen einer Anwendung (z.B. Benutzer, Schema) besitzen eine eindeutige Identifikation in Form einer URI

Verknüpfung/Hypermedia

- Applikationsfluss und Relationen zwischen Ressourcen werden durch Verknüpfungen modelliert

Standardmethoden

- Die standardkonforme Verwendung der HTTP-Methoden garantiert, dass sowohl bekannte, als auch unbekannte Clients die REST-Schnittstelle nutzen können

Unterschiedliche Repräsentationen

- Die unterschiedlichen Anforderungen an die Benutzung einer Ressource werden durch die unterschiedlichen Repräsentationen dieser Ressource abgebildet

Statuslose Kommunikation

- Serverseitiger Zustand wird entweder vom Client gehalten, oder in den Zustand einer Ressource verwandelt

Caching

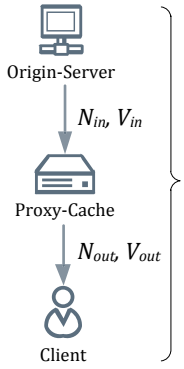
- Repräsentationen einer Ressource sollten, wenn die Möglichkeit besteht, stets in adäquatem Maße durch Caches speicherbar sein.

Abbildung 20 REST - Zusammenfassung als fünf Kernprinzipien

In dem Begriff RESTful HTTP kommen also die erstrebenswerten Eigenschaften einer HTTP-basierten Architektur zum Ausdruck. Aus diesem Grund stellt REST eine ideale Grundlage zur Erreichung der Ziele dieser Arbeit dar.

2.3 Caching

Die für diese Arbeit wichtigste Fähigkeit von HTTP ist das Cache-Controlling. Wegen seiner prominenten Bedeutung für den Erfolg von HTTP ist das Caching als ein REST-Constraint festgehalten. Die Effektivität eines Caches kann auf zweierlei Weisen quantifiziert werden (Abbildung 21). Die *Object Hit Ratio* H_O gibt das Verhältnis von Cache-Hits zu insgesamt bearbeiteten Requests (Cache-Hits + Cache-Misses) an. *Die Volume Hit Ratio* H_V ist analog für das übertragene Datenvolumen definiert [Gri02].



$$H_O = \frac{N_{out} - N_{in}}{(N_{out} - N_{in}) + N_{in}} = \frac{N_{out} - N_{in}}{N_{out}} = 1 - \frac{N_{in}}{N_{out}}$$

Object Hit Ratio

$$H_V = \frac{V_{out} - V_{in}}{V_{out}} = 1 - \frac{V_{in}}{V_{out}}$$

Volume Hit Ratio

Abbildung 21 Effektivitätsmaße für Caches

Ziel von Orestes ist es, durch eine geeignete Verwendung der Cache-Controlling-Mechanismen von HTTP die Object Hit Ratio (→geringe Latenz) und Volume Hit Ratio (→geringe Netzwerklast) von Intermediary-Caches bei der Übertragung von Objekten zu maximieren. Im ersten Schritt werden wir die Mechanismen beschreiben, die HTTP zum Cache-Controlling auf Client-, Zwischensystem- und Serverseite vorsieht. Anschließend werden wird Typen, Funktionsweise und Protokolle von Web-Caches aufzeigen.

2.3.1 Das HTTP-Caching-Modell

Ein wichtiges Ziel von Caches ist die *semantische Transparenz*. Ein Cache ist dann semantisch transparent, wenn es lediglich Einfluss auf die Leistungsfähigkeit hat, wenn eine Anfrage durch einen Cache anstelle des Origin Servers beantwortet wird [MeSa04, S.771]. Das Prinzip der semantischen Transparenz ist im Cache-Controlling von HTTP als Grundkonzept vorgesehen. Eine Abweichung von diesem Prinzip erfolgt nur auf explizite Anweisung des Clients oder Servers.

Das Sequenzdiagramm in Abbildung 22 zeigt zwei Anfragen eines Clients, die über einen Cache abgewickelt werden. Die erste Anfrage erzeugt einen Cache-Hit, die zweite einen Cache-Miss, der den Cache zur Weiterleitung der Anfrage zwingt. Aufgrund der semantischen Transparenz entspricht die Antwort auf die erste Anfrage genau der Antwort, die auch der Origin Server erzeugt hätte.

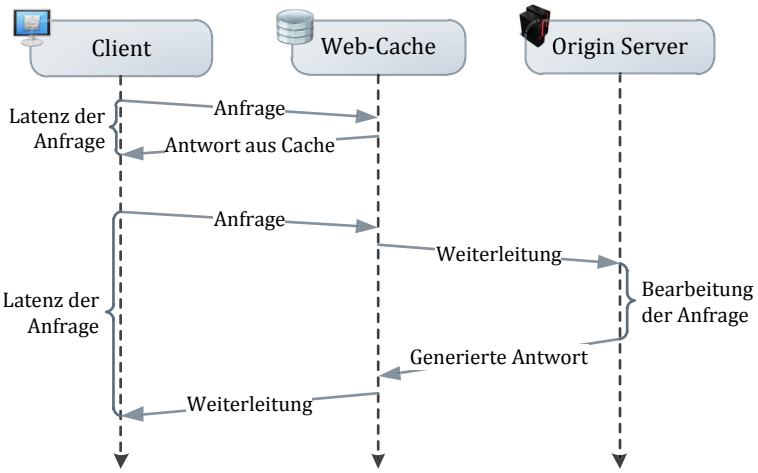


Abbildung 22 Involvierung eines Caches bei einer HTTP-Anfrage

Der Vorteil der aus dem Web-Cache befriedigten Antwort kommt in drei Aspekten zum Ausdruck:

1. Die Latenzzeit der Antwort ist erheblich geringer: Caches sind näher am Client positioniert und sind auf das performante Ausliefern von Cache-Einträgen ausgerichtet.
2. Der Origin Server wird entlastet: er wird bei einem Cache-Hit nicht kontaktiert.
3. Die Auslastung des Netzwerkes wird verringert: Die Anfrage passiert bei einem Cache-Hit nur einen Teil des Gesamtnetzwerkpfades.

Die ursprüngliche Implementierung des Cache-Controllings in HTTP/1.0 sah nur drei Methoden vor, um Cache-Management abzuwickeln: explizite Verfallszeitpunkte (`Expires`), konditionale GET-Anfragen (`If-Modified-Since`), Untersagung des Cachings (`Pragma`). Da diese unzureichenden Cache-Controlling-Header keine angemessene Steuerung des Cache-Verhaltens erlaubten, wurden die Caches zumeist umgangen (*Cache-Busting*). Aus diesem Grund war es ein erklärtes Ziel von HTTP/1.1 detaillierte Kontrollmöglichkeiten für Server und Client einzuräumen. Zwei Modelle setzt HTTP dafür um:

Cache-Expiration-Modell

Der Cache richtet sein Verhalten nach der Gültigkeitsdauer einer gespeicherten Ressource. Die Ressource wird solange als gültig (*fresh*) betrachtet bis ihre Gültigkeitsdauer überschritten ist (*Expiration*). Im Normalfall, der *Server-specified Expiration*, legt der Origin Server diese Gültigkeitszeitspanne als absoluten Zeitpunkt (`Expires`) oder relative Zeitangabe (`max-age` Direktive) fest. Eine absolute Zeitangabe ist dabei wenig zuverlässig, da trotz der von HTTP vorgeschriebenen Synchronisation von Intermediaries über das Network Time Protocol (NTP), die Systemuhren selten synchron laufen. Bei einer relativen Zeitangabe wird der `Age`-Header zugrunde gelegt, der angibt, wie lange die letzte Servervalidierung der Ressource zurückliegt. Er wird beim Ausliefern von Cache-Einträgen stets neu berechnet, unter Einbeziehung von Netzwerk-Round-Trip-Times und der Verweildauer im Cache. Wird durch den Origin Server keine explizite Angabe über die Frische einer Ressource gemacht verwendet der Cache *Heuristic Expiration*. Dazu berechnet der Cache mithilfe des `Last-`

`Modified`-Headers eine heuristische Gültigkeitsdauer der Ressource. Wird eine solche Ressource ausgeliefert, macht der Cache dies durch einen `Warning`-Header deutlich (`113 Heuristic Expiration`), der zur Übermittlung Caching-bezogener Statusinformationen dient.

Cache-Validation-Modell

Eine von einem Cache gespeicherte Ressource kann trotz Ablauf der Gültigkeitsdauer noch immer gültig sein. Da diese Annahme einer Bestätigung durch den Origin Server bedarf, sieht das Cache-Validation-Modell einen Mechanismus vor, um Caches die Revalidierung einer expirierten Ressource zu ermöglichen. Grundlage dieser Revalidierung ist ein *Validator*, ein Header, der dem Origin Server erlaubt festzustellen, ob eine Ressource noch immer als gültig anzusehen ist. In einem *konditionalen Request* an die expirierte Ressource übermittelt der revalidierende Cache dazu den Validator an den Origin Server. Erkennt der Origin Server die Ressource anhand des Validators als noch immer gültig an, sendet er lediglich neue Caching-Header für die Ressource, ohne die Ressource selbst zu übermitteln (`304 Not Modified`). Sollte die Ressource jedoch nicht mehr gültig sein, wird der konditionale Request wie ein gewöhnlicher Request behandelt und die aktuelle Ressource ausgeliefert. Auf diese Weise wird die Validierungsanfrage unabhängig von ihrem Ausgang in einem Request/Response-Paar abgewickelt. Die HTTP-Validatoren sind:

- `Last-Modified` Header: Das Datum der letzten Änderung kann der Origin Server nutzen, um zu überprüfen ob seit dem Zeitpunkt Änderungen an der Ressource vorgenommen wurden
- `ETag` Header: Ein *Entity Tag* ist die Versionsnummer einer Ressource. Die Wahl ihrer Struktur bleibt dem Server überlassen (z.B. Hashwert einer Repräsentation, inkrementierter Zähler). Ein ETag kann ein *schwacher* oder *starker* Validator sein. Ein starker ETag ändert sich bei jedweder Änderung der Ressource. Ein schwacher ETag toleriert Änderungen, die nicht die Semantik der Ressource ändern. Diese subjektive Einordnung semantisch verändernder Eingriffe (z.B. das Hinzufügen eines Whitespace-Zeichens in einem Textdokument) obliegt dem Origin Server. Schwache ETags legen der Interaktion notwendigerweise gewisse Einschränkungen auf (z.B. Verbot von Byte-Range-Abfragen)

Die konditionalen Anfragen, die der Validierung expirierter Einträge mithilfe dieser Validatoren dienen werden durch folgende Header kontrolliert:

- `If-Modified-Since`, `If-Unmodified-Since`: Die Anfrage wird ausgeführt, wenn die Bedingung zutrifft, die Ressource also seit dem angegebenem Datum geändert (`If-Modified-Since`) oder nicht geändert (`If-Unmodified-Since`) wurde.
- `If-Match`, `If-None-Match`: Die Anfrage wird nur ausgeführt, wenn die Ressource den angegebenen ETag besitzt (`If-Match`) oder nicht besitzt (`If-None-Match`).

Das Cache-Expiration-Modell erlaubt also die Steuerung der Gültigkeitsdauer einer Ressource in Caches. Das Cache-Validation-Modell umfasst die Erneuerung und Überprüfung expirierter („stale“ gewordener) Cache-Einträge.

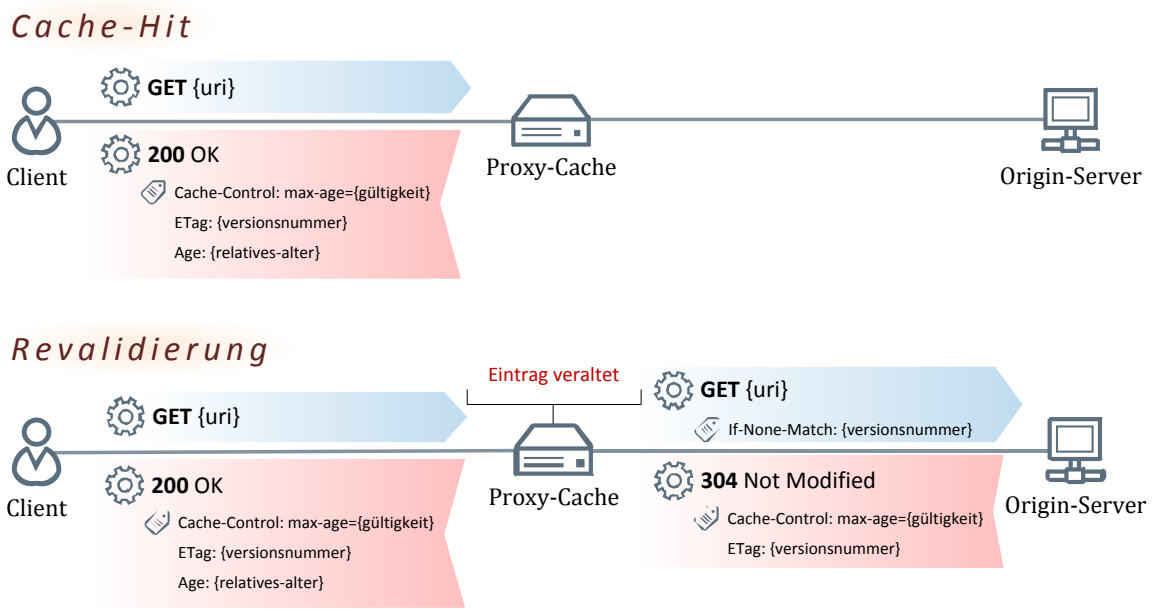


Abbildung 23 Das Expiration- und Validation-Modell von HTTP

Abbildung 23 zeigt beide Modelle an einem Beispiel: im ersten Szenario fragt der Client eine Ressource an, die der Cache gespeichert hat und deren Gültigkeitsdauer noch nicht überschritten ist. Ihr Validator ist ein starker ETag. Bei einer zweiten Anfrage auf die Ressource ist der Eintrag im Cache bereits stale. Daraufhin ruft der Cache mit einem konditionalen GET die Ressource vom Origin Server ab und überträgt dazu im `If-None-Match`-Header den ihm bekannten ETag. Da die Ressource zwischenzeitlich nicht modifiziert wurde, schickt der Origin Server lediglich zu aktualisierende Header, ohne die Repräsentation der Ressource selbst zu übertragen. Der Cache aktualisiert daraufhin seinen Eintrag und liefert ihn an den Client aus.

Die Aspekte anhand derer ein Cache eine Antwort eines GET- oder HEAD-Requests auf Cachebarkeit prüft sind in Abbildung 24 gezeigt. Neben der GET-Methode sind auch andere Methoden für das Caching relevant. Die Antworten auf HEAD-Anfragen werden in gleicher Weise wie die GET-Methode behandelt, sie sind – falls nicht anders gekennzeichnet – stets cachebar. Antworten auf die OPTIONS-, CONNECT- und TRACE-Methode werden nicht Die Methoden POST, PUT, DELETE, PATCH und BATCH invalidieren Cache-Einträge (sogenanntes *non-write-allocate*). Dies geschieht unabhängig davon, ob die Anfrage durch einen Erfolgs- oder Misserfolgsstatuscode vom Origin Server beantwortet wird. Derartige Invalidierungen leisten einen entscheidenden Beitrag dazu Cache-Einträge so aktuell wie möglich zu halten, da alle von einer ändernden Anfrage passierten Caches (bei einem Folgezugriff) ihren veralteten (invalidierten) Eintrag aktualisieren. Dies verhindert jedoch nicht, dass daran unbeteiligte Caches noch immer veraltete Einträge halten.

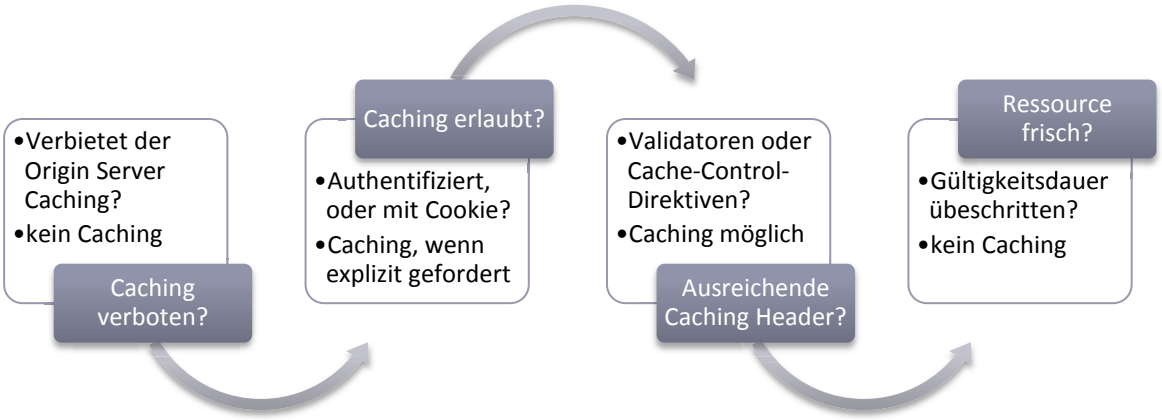


Abbildung 24 Analyse einer Antwort auf Cachebarkeit

2.3.2 Steuerung der Caches

Durch Cache-Control-Direktiven können sowohl Client als auch Server Angaben darüber machen, wie ein Cache mit einer gekennzeichneten Repräsentation umgehen sollte. Der Origin Server kann dabei genau spezifizieren, welche Arten von Caches eine Ressource speichern dürfen:

- **public**: Die Antwort darf immer (z.B. auch bei einer autorisierten Antwort) und von jeder Art von Cache gespeichert werden.
- **private**: Die betreffende Ressource darf nur in Client-Caches gespeichert werden.

Der Origin Server kann zudem einschränken wie eine Ressource cachebar ist:

- **no-cache**: Die Ressource darf nur aus dem Cache zurückgegeben werden, falls dieser sie zuvor validiert hat. Die Direktive kann sich zudem auf eine Anzahl von Headern beziehen. Diese Header müssen stets revalidiert werden, der Rest der Repräsentation darf unverändert ausgeliefert werden.
- **no-store:** Kein Cache darf die Ressource in einem nicht-transienten Speicher lagern.
- **must-revalidate**: Vor jeder Auslieferung muss durch den Cache eine Revalidierung durchgeführt werden.
- **proxy-revalidate**: Client-Caches sind von der Revalidierungspflicht ausgenommen.
- **no-transform**: Die Manipulation von Medientypen ist Intermediaries untersagt.
- **Vary**: Legt diejenigen Header fest, anhand derer ein Cache die auszuliefernde Repräsentation einer Ressource determiniert (*Selecting Header*). Jeder nachfolgende Request in dem sämtliche Selecting Header (z.B. `Accept`) identisch mit jenen der ursprünglichen Anfrage sind, dürfen mit dem zugehörigen Cache-Eintrag beantwortet werden.

Die Dauer der erlaubten Speicherung kann der Origin Server feingranular steuern:

- **max-age**: Die Ressource ist für die angegebene Dauer als frisch anzusehen.
- **s-max-age**: Für **s**hared (public) Caches kann ein anderer Wert festgelegt werden.

- **Expires**: Ab dem angegebenem Datum ist die Ressource stale.
- **stale-while-revalidate:** Diese 2010 in RFC 5861 spezifizierte und bereits weitreichend implementierte Erweiterung des `Cache-Control`-Headers gestattet einem Cache eine bereits expirierte Ressource für eine angegebene Dauer weiterhin (stale) auszuliefern (mit einer `Warning`), während er gleichzeitig asynchron die Revalidierung ausführt.
- **stale-if-error**: Diese zweite, ebenfalls neu eingeführte Erweiterung erlaubt dem Cache bei Unerreichbarkeit des Origin Servers für eine angegebene Dauer eine abgelaufene Ressource anstelle eines Fehler-Statuscodes auszuliefern.

Nachdem durch präzise Angaben des Origin Servers eine Ressource für die Verwendung durch Web-Caches gekennzeichnet wurde, kann nun seinerseits der Client Anweisungen formulieren, die den Caches spezifische Instruktionen zur Beantwortung seiner Anfrage auftragen:

- **must-revalidate**: Der Client verlangt, dass die Ressource durch den Origin Server validiert wurde.
- **max-age, min-fesh:** Die gecachete Kopie der angeforderten Ressource darf das angegebene Alter nicht überschreiten, oder muss noch mindestens für die Dauer der angegebenen Zeitspanne frisch sein.
- **max-stale**: Der Client ist bereit eine abgelaufene Ressource zu akzeptieren, deren Verfallszeitpunkt nicht länger als die angegebene Zeitdifferenz zurückliegt.
- **only-if-cached**: Die angefragte Ressource soll nur dann zurückgegeben werden, wenn eine Kopie in der nächstgelegenen Cache-Gruppe vorliegt.
- **no-transform:** Eine durch den Cache veränderte Repräsentation ist inakzeptabel.

Die beschriebene Menge an Caching-Direktiven sind die Mechanismen die Orestes verwendet, um Objekte und andere Ressourcen auf Web-Caches zu verteilen. Die große Herausforderung bei der Konstruktion eines solchen Systems liegt in dem Umstand, dass Web-Caches reaktive (*on-demand*) Systeme sind [Not10a]. Das proaktive Revalidieren, kontextbezogene Prefetching und serverinitiierte Pushing von Cache-Einträgen sind Fähigkeiten, die nicht vorgesehen und spezifiziert sind. Das birgt die inhärente Gewissheit, dass auch veraltete Ressourcen (z.B. Orestes-Objekte) von Caches ausgeliefert werden, denn die Vorhersage von Änderungszeitpunkten durch den Origin Server bleibt zwangsläufig eine Heuristik. Das unumstößliche Problem besteht also darin, dass ein Origin Server stets einen Trade-Off zwischen Kontrolle (potentielle Änderbarkeit) und Performanz (Vorhandensein in Caches) wählen muss. Deshalb entstanden die in Tabelle 3 aufgeführten Versuche das HTTP-Caching-Modell um neue Fähigkeiten zu erweitern.

Mechanismus	Funktionsweise
Out-Of-Band Invalidierung	Viele Web-Caches (z.B. Squid [Jef10]) unterstützen proprietäre Purge- oder Push-Aufrufe, die es berechtigten Benutzern gestatten, Einträge des Caches zu invalidieren oder vorzuladen [ChaMa99]. Der Nachteil dieser

	Aufrufe ist, dass ein Origin Server alle Caches kennen und erreichen können muss, da andernfalls von unerreichten Caches veraltete Ressourcen ausgeliefert werden. Insgesamt wird eine Inversion der Kontrolle (auch *Hollywood Principle: „Don't call us, we call you"*) erzwungen: nicht der Web-Cache fordert eine Validierung, sondern der Origin Server invalidiert aktiv Ressourcen. Eine vollständige Ersetzung des Validierungs- durch das Invalidierungsmodell bedingt, dass der Origin Server lange Gültigkeitsdauern für Ressourcen festlegt, um das Auslaufen von Ressourcen gänzlich durch Invalidierungsaufrufe zu steuern. Da alle Web-Caches die langen Gültigkeitsdauern als Caching-Direktive erkennen, aber nur ein geringer Teil der Web-Caches Invalidierungen verarbeiten kann, entstehen veraltete Kopien in nicht invalidierbaren Web-Caches. Der Versuch eine PURGE-Methode für HTTP einzuführen (primär für Inter-Cache-Kommunikation) scheiterte bisher [Coo99].
Edge Side Includes (ESI)	Edge Side Includes (ESI) sind ein von Content Delivery Networks (CDNs) entwickelter Ansatz. Es handelt sich dabei um eine XML-basierte Markup-Sprache, die es erlaubt, Ressourcen in Web-Caches aus gecacheten und dynamischen Inhalten zusammenzusetzen. Durch die Definition von ESI-Templates, können verschiedene Ressourcen (mit eigenen Metadaten, wie Caching-Direktiven) in eine assemblierte Gesamtressource eingebunden werden. Dabei können einfache Überprüfungen und Failoveroptionen (z.B. alternative URIs) eingebettet werden. Ebenfalls Bestandteil des W3C-ESI-Standards ist ein neuer HTTP-Header (`Surrogate-Control`), mit dem ESI-Caching-Anweisungen an kompatible Caches übermittelt werden. Außerdem können durch ein HTTP-POST-basiertes Invalidierungsprotokoll Ressourcen und Fragmente in ESI-Caches invalidiert werden [JLL01].
Cache-Channels	Ein vielversprechender, für die Standardisierung vorgesehener Ansatz, ist die Cache-Channel HTTP-Erweiterung [Not07a], [CDL01]. Das Konzept der Cache-Channels steht dem der serverinitiierten Invalidierung diametral gegenüber: anstatt einzelne Ressourcen aktiv zu invalidieren wird inkrementell ihre Frische bestätigt. Zu diesem Zweck veröffentlicht der Origin Server einen XML-Atom-Feed unter einer URI, die er im `Cache-Control`-HTTP-Header angibt (*Publish/Subscribe-Paradigma*). Caches, die Cache-Channels implementieren (z.B. Squid), abonnieren diesen und wissen, dass alle gekennzeichneten Antworten so lange frisch sind, bis der Cache-Channel unerreichbar, oder im Cache-Channel eine Invalidierung veröffentlicht wird. Anstatt dass der Origin Server also Web-Caches aktiv kontaktiert, rufen diese periodisch den Cache-Channel Feed auf (*Polling*). Bei einer adäquat gewählten Frequenz (z.B. alle 60s) bleibt die Frische gecacheter Ressourcen hoch, bei gleichzeitigem Performancegewinn durch die Vermeidung häufiger Revalidierungen.

Tabelle 3 Erweiterungsversuche des HTTP-Caching-Modells

Neben den drei Ansätzen, die wohl als einzig praxisrelevante – da implementiert und eingesetzte – angesehen werden können [Not07b], existieren in der Literatur zahlreiche weitere Vorschläge für die Erweiterung des HTTP-Caching-Modells (PCV [CKR98], PSI [KrWi99], WCIP [CDL01], WCDP [NRT02], DOCP [ADJ+99]). Da die meisten dieser Erweiterungen

darauf abzielen, das HTTP-Caching-Modell durch ein neues Modell mit starker Cache-Konsistenz zu ersetzen, skalieren sie nicht ausreichend und sind schwer zu implementieren.

Die in Tabelle 3 geschilderten Erweiterungen sind jedoch in gewissem Umfang für Orestes nutzbar. Ein Cache-Channel ist eine sinnvolle Option, um in regelmäßigen Abständen Aktualisierungen (z.B. die Löschung eines Objektes) an Web-Caches zu propagieren. Da die entsprechende Erweiterung des `Cache-Control`-Headers von allen Web-Caches ignoriert wird, die unfähig sind ihn zu interpretieren, wird das normale HTTP-Validierungsmodell nicht aufgehoben, sondern nur ergänzt. Die Frische von gecacheten Ressourcen garantieren jedoch auch Cache-Channels nicht, da einerseits Änderungen innerhalb einer Abfrageperiode geschehen können und andererseits nicht alle Web-Caches in der Lage sind, Cache-Channels zu nutzen.

Die beiden anderen Erweiterungen (Purge, ESI) setzten die explizite Implementierung dieser Funktionalität in Web-Caches voraus. Informationen über die an der Kommunikation beteiligten Web-Caches können serverseitig aus dem `Via`-Header ermittelt werden, in den sich alle Intermediaries mit einer Identifikation eintragen. Mithilfe dieser Identifikation wäre der Origin Server in der Lage, implementierungsspezifische Caching-Funktionalität in unterstützenden Caches zu nutzen. Der große Aufwand einer Umsetzung dieses Prinzips verbietet jedoch die praktische Nutzbarkeit eines solchen Ansatzes.

2.3.3 Web-Caches

In diesem Kapitel werden wir näher untersuchen, welche Arten von Web-Caches existieren und für Orestes relevant sind. Web-Caches können in verschiedenen Formen entlang des Kommunikationspfades zwischen Client und Origin Server auftreten [GoTo02]:

Client-Cache / Server-Cache
Ein Cache kann direkt in die Implementierung des Clients oder Servers integriert sein (z.B. Caches von Web-Browsern, Caching-Modul des Apache-HTTP-Servers).

Forward-Proxy-Cache
Ein im Netzwerk des Clients installierter Proxy kann als Forward-Cache die Antworten auf ausgehende HTTP-Anfragen speichern. Er kann z.B. in Firmen als sogenannter *Egress Proxy* eingesetzt werden, um Latenzzeiten und Netzwerklasten zu verringern. Bei Internet Service Providern (ISPs) werden häufig analog arbeitende *Ingress Proxys* an Zugriffspunkten zu Kunden eingesetzt.

Reverse-Proxy-Cache
Ein Proxy-Cache kann als Reverse-Proxy (auch *Surrogate Proxy*) direkt vor einem Origin Server installiert sein und an seiner Stelle eingehende Anfragen beantworten (bei einem Cache-Hit) oder an den Origin Server weiterleiten (bei einem Cache-Miss).

Web-Proxy-Cache

Da HTTP-Anfragen auf geographisch entfernte Server zumeist die Netzwerke verschiedener ISPs passieren, setzen diese an den Verbindungspunkten autonomer Systeme (*Internet Peering Points*) *Network Exchange Proxys* ein. Das Internet besteht derzeit aus über 25.000 autonomen Systemen (AS) [Ber08, S.1]. Diese gehören neben den ISPs beispielsweise zu Hosting Dienstleistern, großen Firmen oder Krankenhäusern. AS definieren sich dadurch, dass sie nicht auf eine dritte Partei für den Internetzugriff angewiesen sind, selbst entscheiden können mit welchen AS sie Traffic austauschen und bei einer Registrierungsstelle (z.B. DENIC oder INTERNIC) verzeichnet sind. Da ein AS nicht mit jedem anderen AS unmittelbar verbunden sein kann, sind Abkommen unter den Betreibern der Netzwerke notwendig, um Traffic indirekt über Zwischensysteme weiterzuleiten:

- *Peering*: Zwei benachebarte autonome System tauschen über eine direkte Verbindung ohne Gebühren Traffic untereinander aus
- *Transit*: Ein AS vermittelt Traffic zwischen zwei anderen Netzwerken und berechnet dafür Transit-Gebühren

Die Position eines ISPs in der Frage, ob ein anderer ISP mit diesem aus ökonomischen Beweggründen ein Peering-Abkommen eingeht (um *Transit*-Gebühren einzusparen), spiegelt die Mächtigkeit und Abdeckung eines ISPs wieder. Diese teilt man deshalb in Tier-1-ISPs (nur Peering-Abkommen), Tier-2-ISPs (Peering- und Transit-Abkommen) und Tier-3-ISPs (nur Transit-Abkommen) ein. Es ist also aus wirtschaftlichen Gesichtspunkten für ISPs sinnvoll eine möglichst große Anzahl von HTTP-Anfragen durch Web-Caches zu befriedigen, um Transit-Gebühren und Netzwerkauslastung gering zu halten. Mit der Entwicklung der Peering-Abkommen hat sich gezeigt, dass an *Peering Point*s (Verbindungstellen zwischen zwei Peers) häufig neue Verbindungen zu weiteren Peers entstanden [Ber08, S.3]. Daraus haben sich unabhängige *Internet Exchange Points* (IXP) entwickelt, die als zentralisierte Schnittstelle gemäß der Peering Abkommen angeschlossener AS Traffic weiterleiten. Dies ist nicht nur deshalb profitabel, weil Investitionskosten in Infrastruktur eingespart werden, viel entscheidender sorgt diese Bündelung der Peering-Kanäle auch dafür, dass Web-Caches an zentraler Stelle (am Übergang ISP/IXP) besonders effektiv arbeiten können. Denn ein Public-Cache, über den eine große Anzahl von HTTP-Abfragen vermittelt werden, hat die Möglichkeit, viele Antworten zu cachen („warm cache“) und die Einträge aktuell („fresh cache“) zu halten. So erzielen dergestalt positionierte Caches eine besonders hohe Cache-Hit-Rate.

Web-Proxy-Caches werden auch als *Replica-Server* innerhalb sogenannter Content Delivery Networks (CDN) eingesetzt. Dabei handelt es sich um ein Software-as-a-Service Modell, bei dem der Betreiber des CDNs (z.B. ein ISP) Content von dem Orgin Server eines Kunden über ein *Distributionssystem* auf Replica-Server (Web-Proxy-Caches) des CDNs verteilt. Mithilfe eines *Request-Routing-Systems* werden die von Clients eingehenden HTTP-Anfragen an den nächstgelegenen Replica-Server weitergeleitet. CDNs umfassen also geographisch verteilte Cluster von Web-Caches, um mit möglichst geringer Latenzzeit HTTP-Anfragen beantworten zu können.

Abbildung 25 illustriert die fünf beschriebenen Typen von Web-Caches an einem Beispiel. Sie zeigt ein Client-Netzwerk, deren Clients jeweils einen dedizierten Cache besitzen (*Client Cache*). Am Ausgang dieses Netzwerks befindet sich ein *Forward-Proxy-Cache*, den sich alle Clients teilen (Shared Cache). Der Internetzugang des Client-Netzwerks erfolgt über den Tier-2 Service Provider C, der durch ein Transit-Agreement über ISP A mit dem Zielnetzwerk indirekt in Verbindung treten kann. ISP A und ISP B tauschen untereinander gebührenfrei Traffic aus (Peering). An ihren Peering Points sind jeweils leistungsstarke Web-Proxy-Caches vorhanden. ISP B leitet den Verkehr an den kleineren Service Provider D weiter, von wo aus er ins Server-Netzwerk gelangt. Dort teilen sich die Origin Server einen *Reverse-Proxy-Cache* und besitzen ihrerseits wiederum dedizierte *Server-Caches*.

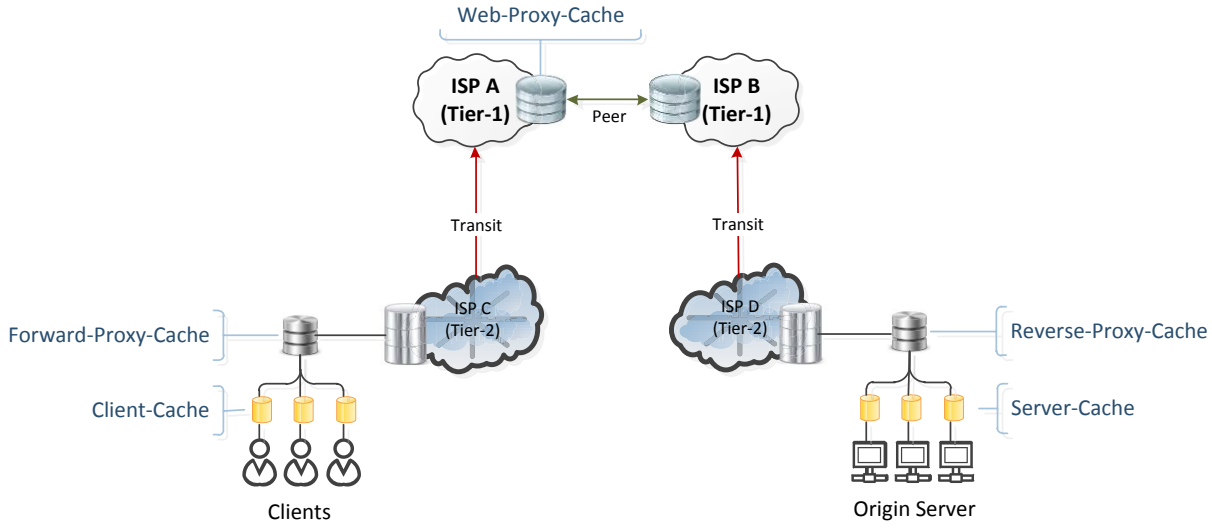


Abbildung 25 Typen von Web-Caches

Neben den von Service Provider D und C als Ingress-Proxies betriebenen Web-Proxy-Caches könnten innerhalb der beteiligten ISPs und Zwischensysteme eine Anzahl weiterer Web-Caches positioniert sein. Aus dieser Vielfalt von Web-Caches entsteht deshalb ein Geflecht, das meist als Web-Caching-Hierarchie bezeichnet wird, da es sich um eine baumartige Topologie handelt.

2.3.4 Funktionsweise von Web-Caches

Bei den drei Web-Caches auf Netzwerkinfrastrukturebene – Web-Proxy-Cache, Forward-Proxy-Cache, Reverse-Proxy-Cache - handelt es sich um Proxys (engl. „Stellvertreter“). Diese Caches sind im Rahmen von Orestes von oberster Bedeutung. Bei den Proxy-Caches lassen sich allgemein zwei Typen unterscheiden: hardware-basierte (z.B. ApplianSys CACHEbox, Blue Coat ProxySG) und software-basierte (z.B. Squid, Varnish, Microsoft Forefront TMG). Bei hardware-basierten Cache-Servern sind alle Komponenten optimiert für den ausschließlichen Einsatz als Proxy-Cache. Ein software-basierter Proxy-Cache hingegen kann als Serveranwendung ohne spezielle Anforderungen an die Hardware ausgeführt werden. Zahlrei-

che software-basierte Proxy-Caches sind als Open-Source-Produkte für verschiedenste Betriebssysteme verfügbar [Jef10].

Für die Beteiligung der Proxy-Caches an der HTTP-Kommunikation ist es notwendig, dass ein Client, der eine Verbindung zu einem HTTP-Origin Server aufbauen will (i.d.R. über TCP) diese Verbindung tatsächlich mit einem Proxy eingeht. Dazu existieren drei Verfahren: die explizite Proxy-Konfiguration, automatische Proxy-Konfiguration und Interception-Proxys. Die Mechanismen zur Umsetzung dieser drei Verfahren sind in Tabelle 4 gezeigt [GoTo02, S.451].

Mechanismus	Funktionsweise
Explizite Konfiguration	Der HTTP-Client ist explizit konfiguriert für die Verwendung eines bestimmten Proxys. Jede HTTP-Anfrage wird an diesen Proxy gesendet. Die Identität des Origin Servers bleibt in der absoluten URI der Request-Line und dem `Host`-Header erhalten.
Proxy Auto-Configuration (PAC)	Der HTTP-Client ruft eine PAC-Datei von einem Konfigurationsserver ab. Diese enthält eine JavaScript-Funktion, die für eine gegebene URI ausgibt, ob und wenn ja, welcher Proxy verwendet werden soll.
Web-Proxy Auto-Discovery Protocol (WPAD)	Mithilfe von DNS (Domain Name System) oder DHCP (Dynamic Host Configuration Protocol) ermittelt der HTTP-Client dynamisch einen Konfigurationsserver, von dem er eine PAC-Datei bezieht.
Web-Cache-Coordination Protocol (WCCP)	Ein Router wertet Zieladresse und Port von Paketen aus und leitet diese gemäß seiner Konfiguration an Proxys weiter. Um die IP-Adresse des Absenders nicht zu verlieren, wird das weitergeleitete Paket dazu gekapselt in einem Zwischenprotokoll übertragen. Der empfangende Proxy (der Web-Cache) kann auf diese Weise eingehende TCP-Verbindungsanfragen, die an den Origin Server gerichtet sind, an seiner Stelle annehmen und so seine Rolle übernehmen (*Interception*). Für den Client läuft dies transparent ab und erfordert keine Konfiguration, weshalb Interception-Proxys als Web-Caches in Zwischensystemen eingesetzt werden.

Tabelle 4 Verfahren zur Konfiguration von Proxy-Servern

Es ist also möglich, Proxy-Caches entkoppelt von der Client-Konfiguration durch das Web-Proxy Disovery Protokoll in einem Client-Netzwerk einzusetzen. Ebenso ist es möglich Web-Caches in beliebiger Anzahl durch Interception transparent für den Client als Intermediary an der HTTP-Kommunikation zu beteiligen. Abbildung 26 zeigt dies an einem Beispiel.

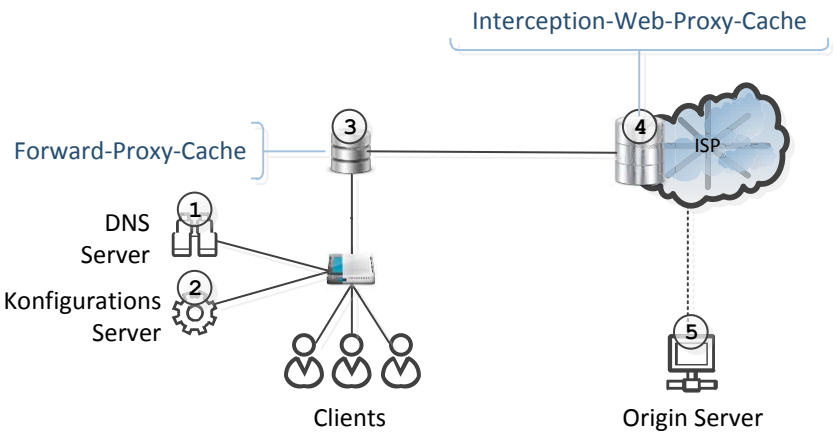


Abbildung 26 Beteiligung von Proxys bei einer HTTP Anfrage

1. Der Client erfragt gemäß dem WPAD-Protokoll einen zuständigen Proxy-Konfigurationsserver von einem DNS-Server (Broadcast-Anfrage).
2. Von diesem Server ruft der Client eine PAC-Datei ab.
3. Der Client setzt eine HTTP-Anfrage an den Origin Server ab, dazu verbindet er sich mit dem aus der PAC-Datei ermitteltem Proxy-Cache.
4. Der Forward-Proxy-Cache versucht bei einem Cache-Miss eine Verbindung zu dem Origin Server aufzubauen. Ein Router des vermittelnden ISPs registriert ein TCP-SYN-Paket auf Port 80 und leitet dieses und die folgenden Pakete an einen Interception-Web-Proxy-Cache weiter, wodurch der Forward-Proxy mit ihm verbunden ist.
5. Auch der Interception-Web-Proxy-Cache baut (bei einem Cache-Miss) eine Verbindung zum Origin Server auf, und setzt die empfangene HTTP-Anfrage erneut ab. Der Origin Server kann an dem `Via`-Header die Cache-Beteiligung und – falls gesetzt – die IP-Adresse des ursprünglichen Anfragers (`X-Forwarded-For`-Header) erkennen.

Da Web-Caches mitunter extremen Anforderungen ausgesetzt sind, haben sich Inter-Cache-Protokolle zur Benutzung von Web-Caches in Gruppen entwickelt [GoTo02, S. 473]. Mit diesen Protokollen ist es Web-Caches möglich, benachebarte Caches nach Vorhandensein eines Cache-Eintrages für bestimmte Ressourcen zu befragen und Zustandsinformationen auszutauschen. Auch können durch solche Protokolle Cluster von Web-Caches als ein logischer Cache angesprochen werden [GoTo02, S.478]. Die Überlegenheit der daraus erwachsenden irregulären Vermaschung von Caches gegenüber einer strengen baumartigen rchieHierachie äußert sich in mehreren Punkten [Gri02, S. 60]:

- Bei einer baumartigen Topologie befragt ein Cache bei einem Cache-Miss seinen übergeordneten Cache. Dieser stellt deshalb einen *Single Point of Failure* dar.
- Höher in der Hierarchie liegende Web-Caches werden stärker belastet. Um Skalierbarkeit solcher Caches zu ermöglichen, sollte es einerseits möglich sein, einen Cache-Miss in Abhängigkeit der Lastsituation an einen ausgewählten höher liegende Cache

zu delegieren. Außerdem sollten Caches auch vertikal skalieren, also die Last auf mehrere verteilt arbeitende Caches verteilen können.

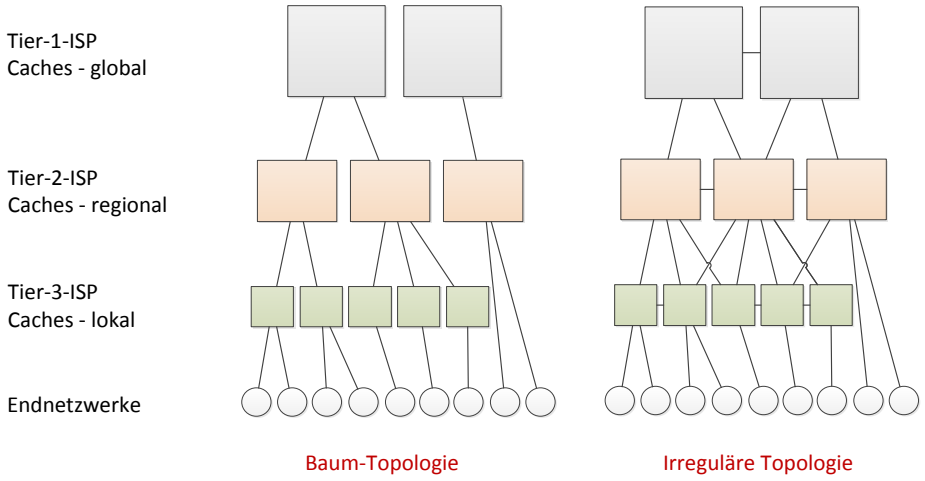


Abbildung 27 Topologien von Cache-Hierachien nach [Gri02]

Abbildung 27 zeigt den Unterschied zwischen einer baumartigen Web-Cache-Hierarchie (z.B. durch den Einsatz von Interception-Web-Proxy-Caches auf verschiedenen Netzwerkebenen) und einer auf Inter-Cache-Protokollen basierenden Cache-Hierarchie. In einer Cache-Hierarchie kommuniziert ein Cache mit *Neighbors* (Nachebarn), die wiederum *Siblings* (Geschwister) oder *Parents* (Eltern) sein können. Abgebildet ist eine Hierarchie von Web-Caches von drei ISPs unterschiedlicher Größe. Im Falle einer ausschließlich auf HTTP basierenden Kommunikation (Baum-Topologie) kommunizieren Caches dabei ausschließlich mit einem Parent. Werden jedoch Inter-Cache-Protokolle eingesetzt, können Caches mit Siblings und mit mehreren Parents kommunizieren. Zu beachten ist, dass für die Übertagung von Repräsentationen zwischen Caches noch immer HTTP verwendet wird. Lediglich der Austausch von Zustandsinformationen und Informationen über Speicherinhalte werden im Cache-Verbund durch die Inter-Cache-Protokolle ausgetauscht. Die eingesetzten Protokolle [SIM, S.49] zur Inter-Cache-Kommunikation sind in Tabelle 5 erläutert.

Protokoll	Funktionsweise
Internet Cache Protocol (ICP)	Mit dem UDP-basierten Protokoll ICP können Caches eines Cache-Verbundes ihre Neighbors nach einem Cache-Eintrag für eine URI befragen (per Multicast oder Unicast). Bei einem Cache-Hit eines Neighbors wird, von jenem mit der geringsten Antwortzeit, in einer HTTP-Anfrage die Repräsentation abgerufen. Bei Cache-Misses oder Timeouts wird die Anfrage an einen Parent delegiert. Das relativ alte ICP kennt neben der URI keine weiteren Repräsentationsmetadaten, ist aber noch immer weitverbreitet.
Hypertext Caching Protocol (HTCP) [RFC2756]	HTCP, verbessert ICP um Mechanismen zur Authentifizierung von Caches und dem Austausch von Repräsentationsmetadaten. Bei einem *False Hit* (Repräsentation bei Empfang von Neighbor bereits veraltet) kann der Cache so präziser reagieren. Außerdem können gezielt Varianten einer Ressource angefragt werden (z.B. `content-language=DE/de`).

Cache Digests [RoWe98]	Cache Digests stellen ein Möglichkeit dar, um, anders als bei ICP und HTCP, nicht Informationen über einzelne gespeicherte Repräsentationen auszutauschen, sondern ein Verzeichnis davon - den Cache Digest. Mithilfe eines *Bloom-Filters* [Blo70] generiert ein Neighbor aus einer Liste von gespeicherten URIs durch Hashfunktionen einen Digest festgelegter Größe. Dieser periodisch ausgetauschte Cache Digest kann nun von einem Cache verwendet werden, um zu ermitteln, ob ein Neighbor mit hoher Wahrscheinlichkeit eine Repräsentation der angefragten URI besitzt. Die Netzwerklast wird so gesenkt. Durch den periodischen Austausch und die Komprimierung des URI-Verzeichnisses als Cache Digest erhöht sich jedoch die Anzahl der False Hits.
Cache Array Routing Protocol (CARP)	Mithilfe von CARP kann eine Gruppe von Caches als ein logischer Cache agieren. Die gespeicherten Repräsentationen werden dazu durch einen Consistent-Hashing-Algorithmus disjunkt auf die beteiligten Server verteilt. Meist wird CARP in Kombination mit WCCP-Interception-Proxying eingesetzt. Der Router delegiert dabei, ausgehend von der deterministischen Hashwertberechnung, einen eingehenden Request an den geeigneten Cache. CARP eignet sich wegen der engen Kopplung nur für hochverfügbare Cache-Cluster [SIM, S. 48].

Tabelle 5 Protokolle der Inter-Cache Kommunikation

Die Hierarchien und Protokolle von Web-Caches bringen wichtige Konsequenzen für die Konzeption von Orestes mit sich:

- Caching-Hierarchien sind hochgradig skalierbar, deshalb ist eine serverseitige Vermeidung der Überlastung und Überfüllung von Web-Caches unnötig.
- Cache-Kohärenz (ausgelieferte Repräsentation eines Caches entspricht der des Servers) ist für cachebare Ressourcen generell problematisch (z.B. aufgrund von False Hits bei Inter-Cache-Protokollen) und kann deshalb nicht vorausgesetzt werden.
- Die Vielzahl der beteiligten Web-Caches auf unterschiedlichen Netzwerkebenen bedingt, dass der Einsatz von implementierungsabhängigen, serverinitiierten Anweisungen (z.B. Push, Purge) höchstens eine Ergänzung sein kann.
- Server- und Client-Caches sind für Orestes unnötig, da diese auf Netzwerkebene durch die verschiedenen Proxy-Typen generischer, performanter und austauschbarer eingesetzt werden können.
- Der Orestes-Server kann aufgrund beteiligter Inter-Cache-Protokolle und Routing nicht davon ausgehen, bei der Kommunikation mit einem bestimmten Client stets dieselben Caches anzutreffen.

2.4 OODBMS

Da Orestes darauf abzielt Leistungsparameter von Objektorientierten Datenbanken (OODBMS) zu verbessern, werden wir im Folgenden beispielhaft ihrer Funktionsweise erläutern. Eine umfassende Darstellung der Techniken von Objektorientierten Datenbanken würde den Umfang dieser Arbeit sprengen, deshalb werden wir die Motivation und Nutzen ihrer Verwendung in wenigen Kernpunkten zusammenfassen.

Die ca. 1980 entstandenen objektorientierten Datenbanken sind ein Gegenentwurf zu relationalen Datenbanken, die jedoch noch immer deutlich den Markt beherrschen (Marktanteil ca. 95% [Zic09]). Der Anspruch der OODBMS kommt in folgenden Punkten zum Ausdruck:

- Verwendung des gleichen Datenmodells in Programmiersprache und Datenbank
  - Kriterien des *Object-Oriented Database Manifesto* [ABD+92]: Complex objects, Object identity, Encapsulation, Types and classes, Type and class hierarchies, Overriding, overloading and late binding, Computational completeness, Extensibility
  - Nahtlose Integration in die verwendete Programmiersprache
- Hohe Performanz bei navigierendem Zugriff (dem Folgen von Referenzen)
- Unterstützung komplexer Anfragen
- Allgemeine Eigenschaften: Skalierbarkeit, Nebenläufigkeit, Persistenz, Verlässlichkeit, Verteilung, Replikation und Effizienz

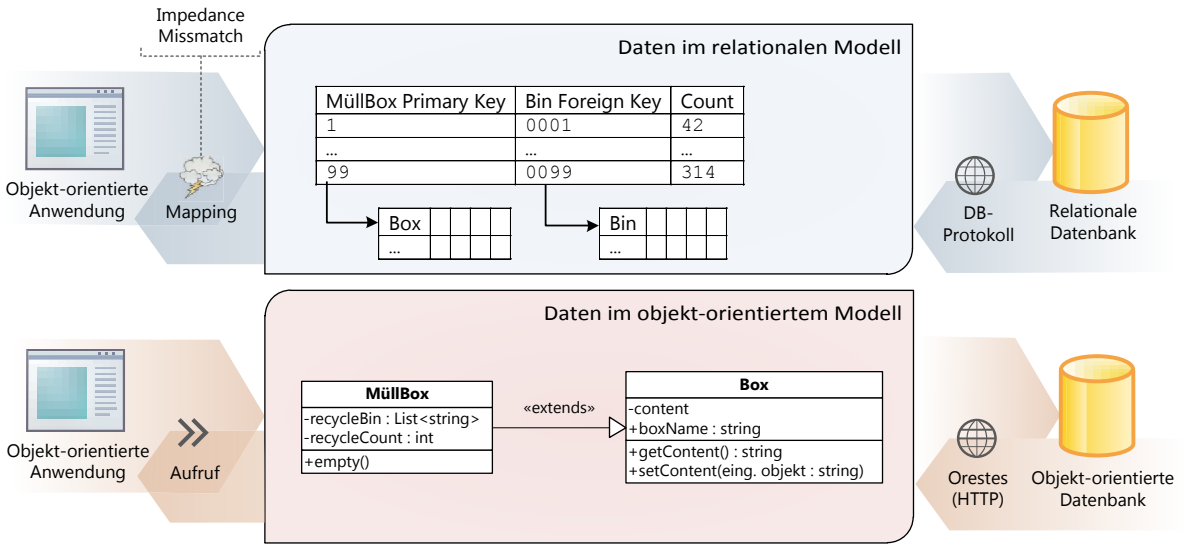


Abbildung 28 Das Zusammenspiel von Anwendung, Datenmodell und Datenbank

Abbildung 28 zeigt den Zusammenhang zwischen objektorientierter Anwendung, Datenmodell und Datenbank. Bei Verwendung einer relationalen Datenbank als Persistenzschicht der Anwendung ist *Objekt-Relationales Mapping* erforderlich, da zwei unterschiedliche Datenmodelle miteinander verbunden werden müssen → (*Impedance Missmatch*). In der Umgehung von Mapping-Frameworks (z.B HIBERNATE oder LINQ TO SQL) und der nativen Unterstützung von Objektorientierung liegt die entscheidende Daseinsberechtigung von

OODBMS. Aus diesem Grund werden OODBMS vorwiegend in Bereichen eingesetzt, wo Algorithmen und schneller navigierender Zugriff im Vordergrund stehen (z.B. in Telekommunikation, Finanzwesen, Simulation, Reservierungssystemen [Zic09, S.4]) während RDBMS in datenlastigen Szenarien deutlich überlegen sind.

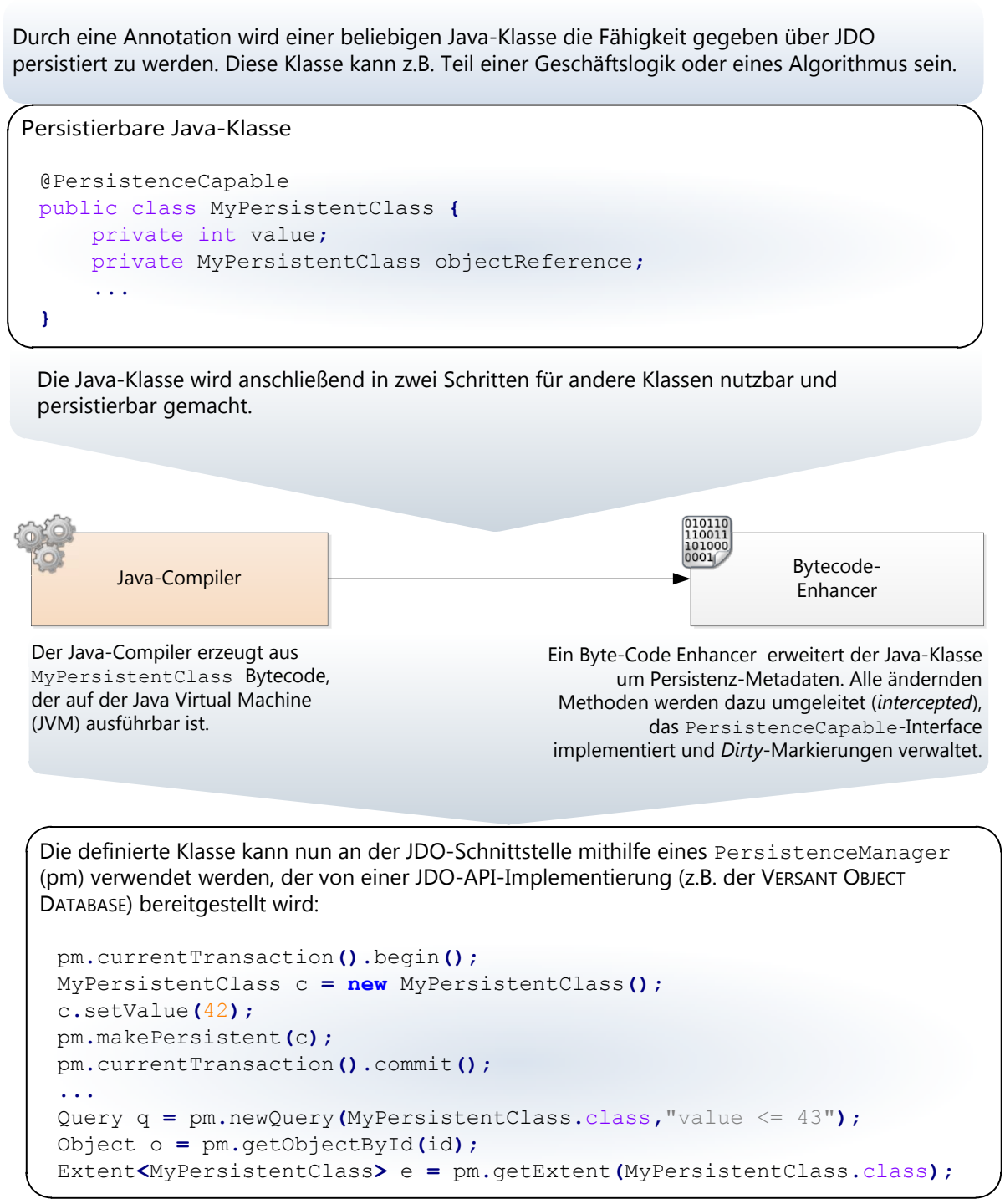


Abbildung 29 Die Verwendung eines OODBMS am Beispiels von Java/JDO

Die Verwendung eines OODMS werden wir an einem Beispiel verdeutlichen (Abbildung 29): Aus der Programmiersprache Java soll ein Objekt mithilfe des Persistenz-API *Java Data Objects* (JDO) in ein OODBMS (z.B. VERSANT OBJECT DATABASE) gespeichert und anschlie-

ßend wieder abgerufen werden. Bei JDO handelt es sich um eine standardisierte Client-Persistenz-API für Java, die sich aus einem 2001 von der *Object Database Management Group* (ODMG) dem *Java Community Process* (JCP) übergebenem Entwurf zur Integration einer objektorientierten Datenbank in Java entwickelt hat. Die an der JDO-Schnittstelle aufgerufenen Datenoperationen können sowohl auf relationale Datenbanken (durch Mapping-Deklarationen) als auch durch objektorientierte (nativ) abgebildet werden.

Es ist wichtig die Operationen auf Ebene von Orestes nicht mit jenen von Client-Zugriffs-APIs wie JDO zu verwechseln. Orestes ist auf einer niedrigeren Abstraktionseben angesiedelt, seine Operationen umfassen genau diejenige Grundfunktionalität, die notwendig ist, um Abruf und Übermittlung von Objekten, sowie Transaktions-, Klassen- und Schemaverwaltung auf Übertragungsebene auszuführen. Damit lassen sich komplexe, objektorientierte Persistenz-Schnittstellen auf die Orestes-Operationen abbilden. Diese Operationen wiederum werden aufgrund der genutzten Konzepte (z.B. Ausnutzung der Web-Caching-Hierarchie) besonders performant ausführbar sein und dadurch das Leistungsverhalten der Persistenz-API beeinflussen.

2.4.1 Eigenschaften und Abstraktionen

Orestes ersetzt also keineswegs Persistenz-APIs durch eine HTTP-orientiertere Schnittstelle. Aufgaben wie die Verwaltung von Objektzuständen (z.B. *Transient*: nicht persistent, *Persistent Dirty*: persistent aber noch nicht gespeichert, *Hollow*: persistent aber ohne geladene Werte) und Abhängigkeitsgraphen (z.B. zur Umsetzung von *persistence-by-reachability* beim Commit) obliegen allein der Persistenz-API. Aufgaben wie das Management von Netzwerk-Verbindungen und Caching-Direktiven hingegen obliegen alleine der Orestes-Implementierung (Orion). Die zentralen objektorientierten Abstraktionen von OODBMS, die in Orestes verwendet werden, sind:

Objektidentität und -version
In objektorientierten Datenbanken besitzt jedes Objekt eine Objektidentität (*OID*), die es eindeutig identifiziert. Sie dient der Adressierung (durch Referenzen). Objekte mit Wertsemantik werden gegenüber solchen *First-Class-Objects* (FCO) als *Second-Class-Objects* (SCO) bezeichnet, besitzen keine Identität und sind stets Bestandteil von anderen Objekten (z.B. eine Zeichenkette). Jedes identifizierbare Objekt (FCO) muss für die Nebenläufigkeitskontrolle zudem eine Version oder Änderungsdatum besitzen. Die OID dient in Orestes als Ressourcen-Identifikation, die Version (die auch ein Änderungsdatum oder Hashwert sein kann) als HTTP-Validator für Web-Caches.

Native Datentypen
Sammlungsdatentypen (*Collections*) sind Objekte, die sowohl in OODBMS als auch Orestes eine spezielle Behandlung erfahren: *Sets* (Mengen), *Maps* (assoziative Arrays), *Listen* und *Arrays*. Gleiches gilt für primitive Typen (z.B. Integer, Float, Boolean).

Klasse und Schema
Ein Objekt, das aus einer Anwendung persistiert werden kann, ist stets eine Instanz einer Klasse (eines Namensraums), deren Schema (Felder und Typen des Objekts) dem OODBMS zur Persistierung bekannt sein muss. Definition, Änderung (*Schemaevolution*) und Abruf sind auf Ebene von Orestes vorgesehen.

Transaktionen und Querys
Der Zugriff auf ein OODBMS erfolgt in der Regel transaktional. In Orestes sind Transaktionen wichtige Indikatoren für die Wahl von Caching-Direktiven und deshalb integraler Bestandteil der REST-Schnittstelle. Die Verarbeitung komplexer Anfragen ist eine wichtige Fähigkeit eines OODBMS und eines der Hauptunterscheidungsmerkmale gegenüber den im nächsten Kapitel geschilderten NoSQL-Datenbanken. Die Behandlung von Querys ist in Orestes ebenfalls darauf ausgerichtet, Web-Caching zu unterstützen.

2.4.2 Concurrency Control bei Web-Caching

Ebenso wie bei relationalen Datenbanksystemen kann die Mehrbenutzersynchronisation (*Concurrency Control*) in OODBMS durch verschiedene Verfahren realisiert werden: sperrbasierte Verfahren, Multiversionen-Synchronisation, Serialisierbarkeitstester, Zeitstempelverfahren oder optimistische Synchronisation. Orestes erfordert zwingend eine Mehrbenutzersynchronisation ohne Einsatz von Sperren. Dies liegt darin begründet, dass auf Einträge in Web-Caches keine Sperren erhoben werden können. Da unter den nicht sperrenden Verfahren der Serialisierbarkeitstester einen vollständigen Konfliktgraphen einschließlich der Leseoperationen (bei Orestes auch aus Web-Caches) halten muss, ist dieses selten implementierte Verfahren schwer für Orestes adaptierbar. Das Zeitstempelverfahren erzwingt Konfliktserialisierbarkeit, akzeptiert aber nur eine Teilmenge aller konfliktserialisierbaren Schedules. Die optimistische Synchronisation, wie z.B von der VERSANT OBJECT DATABASE implementiert, ist deshalb für Orestes das geeignetste Verfahren zur Mehrbenutzersynchronisation.

Die optimistische Synchronisation basiert auf der Annahme, dass Schedules selten Konfliktsituationen beinhalten [KuRo81]. Diese Annahme ist gerechtfertigt, wenn Leseoperationen mit einer höheren Häufigkeit als Schreiboperationen auftreten oder schreibende Transaktionen tendenziell zu anderen Zeitpunkten und auf anderen Objekten als lesende Transaktionen stattfinden. Bei der optimistischen Synchronisation durchläuft eine Transaktion drei Phasen:

1. **Ausführungsphase**: die Transaktion führt Lese- und Schreiboperationen aus. Schreiboperationen werden an Kopien der Objekte abgewickelt.
2. **Validierungsphase**: beim Commit wird eine Überprüfung auf Konflikte durchgeführt.
3. **Schreibphase**: bei erfolgreicher Validierung werden die Änderungen übernommen.

Um in der Validierungsphase eine Transaktion T auf Konflikte zu prüfen, wird die Menge der geschriebenen (*Write-Set*) und gelesenen (*Read-Set*) Objekte der Transaktion T, sowie ihre

zeitliche Ordnung benötigt: $WriteSet(T)$, $ReadSet(T)$, $O(T)$. Zwei Transaktionen T_i und T_j mit $O(T_i) < O(T_j)$ werden validiert, sind also konfliktserialisierbar, wenn folgende Bedingungen erfüllt sind [Kud07, S. 271]:

1. T_i ist beendet (Schreibphase abgeschlossen), bevor T_j die Validierungsphase beginnt.
2. Für jedes von T_i geschriebene und von T_j gelesene Objekt $obj \in WriteSet(T_i) \cap ReadSet(T_j)$ muss die Schreibphase von T_i vor dem Lesezugriff von T_j bereits abgeschlossen, obj also aktualisiert worden sein.
3. Für jedes von T_j geschriebene und von T_i gelesene Objekt $obj \in WriteSet(T_j) \cap ReadSet(T_i)$ muss der Lesezugriff von T_i vor der Validierungsphase von T_j erfolgt, obj also unverändert sein.

Da ein Client bei Orestes Antworten auf Leseanfragen aus Web-Caches erhält, wird durch Orestes optional das clientseitig protokollierte Read-Set einer committenden Transaktion übertragen. Er ist anschließend in der Lage, auf eine der in Abbildung 30 dargestellten Weisen den optimistischen Commit durchzuführen. Werden dabei veraltete Objektversionen im Read-Set erkannt, so wird die committende Transaktion zurückgesetzt. Bei Konflikten zwischen Transaktionen d,wir je nach Validierungsverfahren (*Forward-/Backward-Oriented Optimistic Concurrency Control*,) die committende, oder die noch aktiven, konfliktierenden Transaktionen zurückgesetzt [Här84].

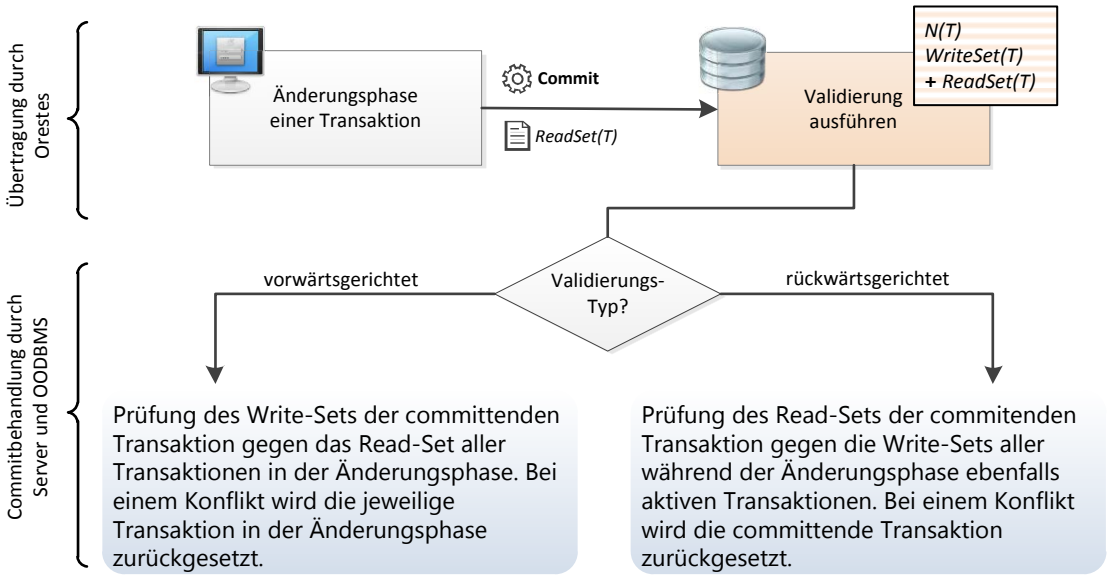


Abbildung 30 Verwendung des optimistischen Commits

Nachteile von objektorientierten Datenbanken müssen in Bezug auf Orestes ebenfalls berücksichtigt werden: fehlende Interoperabilität, mangelnde Standardisierung und uneinheitliche Architekturen [GrNo09].

Die fehlende Interoperabilität und mangelnde Standardisierung bedingen sich gegenseitig. Ein Maß an Verbindlichkeit von Standards, wie ihn beispielsweise SQL für relationale Datenbanken darstellt, existiert für OODBMS nicht (Querysprachen sind z.B. JDOQL, OQL,

VQL EJBQL, JPQL, SODA). Dies liegt zum einen an der sehr sprachspezifischen Einbettung und zum anderen an den verschiedenen Datenbank-Architekturen und -Funktionalitäten. Während die Architekturen relationaler Datenbank sich stark ähneln (server- und index-zentriert, relationale Algebra, etc.) gibt es bei objektorientierten im Wesentlichen drei unterschiedliche: container-, seiten- und objektbasierte. Lediglich eine objektbasierte Architektur eignet sich uneingeschränkt zur Nutzung über Weitverkehrsnetzwerke (WANs) und dadurch für Orestes, da die beiden anderen Architekturen die Übertragung ganzer Datenbank-Seiten und Plattensegmente (Container) erfordern [GrNo09, K. 14, S. 16].

In Bezug auf das bisher unzureichende Maß an Interoperabilität kann Orestes als ein Schritt in Richtung einer einheitlichen, universellen Objekt-Übertragungsschicht für OODBMS angesehen werden. Die Konzepte von Orestes orientieren sich an Objektorientierung und Web-Infrastruktur und sind unabhängig von einer konkreten Client-Persistenz-API und Spezifika konkreter OODBMS.

2.5 Ansätze webzentrierter Persistenztechniken

Um mit Orestes an Konzepte anzuknüpfen, die in Forschung und praktischem Einsatz bereits Erfolge verzeichnen, werden wir bestehende webzentrierte Ansätze zur Persistenz untersuchen. Aufgrund der Unzulänglichkeiten relationaler Datenbanken in Bezug auf Skalierbarkeit und Umgang mit semi- und unstrukturierten Daten [Wig09], haben sich in jüngerer Vergangenheit vermehrt alternative Ansätze entwickelt. Unter dem im Jahr 2009 neu geprägten Oberbegriff NoSQL („Not only SQL“) [Eva09] sind dabei diverse Storage-Engines entstanden, deren Fokus auf der Gewährleistung hoher Skalierbarkeit und anderer nichtfunktionaler Anforderungen liegt. Dazu gehören Kosteneffizienz (keine Lizenzgebühren, Verwendung billiger „Commodity Hardware“), Verfügbarkeit (durch Replikation) und Elastizität (dynamische Erweiterbarkeit durch neue Storage-Knoten). Im Gegenzug werden Zugeständnisse bei der Erfüllung der klassischen ACID-Kriterien (*Atomicity, Consistency, Isolation* und *Durability*) für Transaktionen eingeräumt und der Konsistenzbegriff gezielt aufgeweicht. Im Folgenden werden wir die Eigenschaften und HTTP-Schnittstellen dieser nichtrelationalen Datenbanken und Konzepte näher untersuchen, da sie Orestes in vielen Punkten nahestehen – sie nutzen die Infrastruktur und Protokolle des WWW und dienen der Persistenz und Datenhaltung [Pus10].

2.5.1 Konsistenz und Skalierbarkeit

Hintergrund der Zugeständnisse bei Konsistenzbedingungen ist zum einen die, gemessen an relationalen Datenbanksystemen, vergleichsweise geringe Komplexität der NoSQL-Datenbanken [Maz10], als auch das von Eric Brewer aufgestellte CAP-Theorem [Bro09][Bre00]. Nach diesem Theorem ist es für eine verteilte Datenbank unmöglich alle drei der folgenden Anforderungen zu erfüllen:

- **Consistency**, alle Clients sehen denselben Zustand der Daten.
- **Availability**, das System ist verfügbar – Clients können immer lesen und schreiben, der Ausfall eines Knotens wird durch die Verfügbarkeit einer Replik kompensiert.
- **Partition tolerance**, die aus Netzwerkausfällen resultierende Partitionierung des Clusters von Datenknoten wird durch das System toleriert.

Maximal zwei dieser drei Anforderungen können in einem verteilten System erfüllt werden, beispielsweise Consistency und Availability (CA) bei einem klassisches Single Site Cluster mit einem ACID-garantierendem Zwei-Phasen Commit-Protokoll. Dieses System, z.B. ein verteiltes Datenbanksystem, würde im Fall einer Netzwerkpartitionierung blockieren, um Inkonsistenzen der Datenbasis zu unterbinden [Dim10].

Jedes Datenbankdesign, das auf horizontale Skalierbarkeit abzielt, muss jedoch Daten partitionieren. Dies kann auf zwei kombinierbare Weisen geschehen. Bei der funktionalen Partitionierung [Pri08], werden verwandte Daten gemeinsam gespeichert (z.B. „Customers“, „Products“, „Communication“). Durch sogenanntes „Sharding“ können diese funktionalen Datengruppen weiter partitioniert werden (z.B. „Customers [A-L]“, „Customers [M-Z]“). Da also die horizontalen Skalierungsstrategien auf Partitionierung basieren, müssen entweder

Einbußen in Bezug auf die Verfügbarkeit (A) oder auf die Konsistenz (C) in Kauf genommen werden. Deshalb verzichten viele Vertreter der NoSQL Bewegung auf strikte Konsistenz (im Sinne der ACID-Kriterien) und geben der Skalierbarkeit und Verfügbarkeit Vorrang. Die Terminologie der Varianten von Konsistenzbedingungen umfasst [Mer10a], [Vog08]:

- **BASE** („Basically available, Soft-state, Eventually Consistent“, engl. Base) ist ein Gegenbegriff zu den ACID-Bedingungen. Kernpunkt ist der Tradeoff zwischen Konsistenz und Verfügbarkeit – das Aufgeben des starken Konsistenzanspruchs resultiert in einem „Soft-state“ des Systems mit der Garantie eines später eintretenden, konsistenten Zustands („eventually consistent”). Für deren Umsetzung schlägt D. Pritchett, der Namensvater des Ansatzes, persistentes und idempotentes Message Queuing vor [Pri08]. BASE oder „Eventual Consistency“ ist ein Oberbegriff für die Kategorien abgeschwächter Konsistenzbedingungen.
- **Monotonic Read** sichert zu, dass ein späterer Lesezugriff nie ältere Daten zurückgibt als ein früherer.
- **Monotonic Write** garantiert die Serialisierbarkeit (definierte zeitliche Reihenfolge) von Schreibzugriffen auf einem Datensatz. Ein Schreibvorgang ist also bei Ausführung eines darauf folgenden bereits abgeschossen.
- **Read Your Writes** garantiert, dass ein schreibender Client in einem darauffolgenden Lesezugriff mindestens seine geschriebenen Daten lesen kann. Die Garantie kann auf den Kontext einer Sitzung beschränkt sein („Session Consistency“) [Vog08].
- **Immediate Consistency** garantiert, dass nach einer Schreiboperation an einem Datensatz die Änderungen umgehend und in allen Knoten sichtbar sind [Mer10a].
- **Strong Consistency** erlaubt zusätzlich zu den Bedingungen der „Immediate Consistency“ atomare Lese- und Schreiboperationen an einem Datensatz (z.B. das atomare Ändern von zwei Attributen eines Datensatzes) [Mer10a].
- **Write follows Reads** garantiert, dass ein Schreibzugriff, der auf einen Lesezugriff folgt, auf einer mindestens so aktuellen Version des Datensatzes ausgeführt wird, wie der vorangehende Lesezugriff [Kap05].
- **ACID** Transaktionen sind nicht-unterbrechbare, isoliert ablaufende Operationsfolgen über ein oder mehrere Datensätze hinweg, deren dauerhafte Änderungen das System in einem konsistenten Zustand belassen.

Die beschriebenen Begriffe charakterisieren unterschiedliche Garantien, die ein System in Bezug auf die Konsistenz von Daten gewähren kann. Einige der Eigenschaften stehen in einer direkten Inklusionsbeziehung, so umfasst *Strong Consistency* beispielweise alle Garantien von *Immediate Consistency*. Andere Garantien, wie beispielsweise *Monotonic Read* und *Read Your Writes* stellen eine häufig umgesetzte Kombination verschiedener Garantien dar [Mer10b]. Diese Garantie bedeutet aus Applikationssicht, dass einmal geschriebene Daten sofort lesbar sind und nur monoton ansteigende Versionen von Datensätzen zurückgegeben werden. Die Wichtigkeit der Einhaltung bestimmter Konsistenzgarantien hängt stark vom Anwendungsfall ab. So wäre beispielsweise *Eventual consistency* mit *Read Your Writes*-

Garantie durchaus akzeptabel für die Änderungen eines Profilbildes in einem sozialen Netzwerk [Mer10b]. Das geänderte Bild würde der Nutzer sofort sehen (*Read Your Writes*), andere Nutzer erst zu einem späteren Zeitpunkt (*Eventual consistency*). Da diese aber vor dem Sichtbarwerden des neuen Bildes keine Kenntnis über eine Änderung besitzen, erzeugt das niedrige Maß an Konsistenz keine relevanten Nebeneffekte. Das Fehlen der *Read Your Writes*-Garantie würde den hochladenden Benutzer jedoch verwirren, da der Anschein erweckt wird, dass seine Änderung nicht übernommen wurde. Das Beispiel zeigt, dass Konsistenz ein kontextabhängiger Parameter ist. Diese Kontextabhängigkeit von Konsistenz und Datenmodell sind der Grund für die Vielfalt der verfügbaren NoSQL-Datenbanken. Die hierarchische Beziehung wichtiger Konsistenzgarantien ist in Abbildung 31 gezeigt: die stärksten Garantien gewähren ACID-Transaktionen, die Anforderungen der *Eventually consistent* Konsistenzmodelle nehmen stufenweise ab.

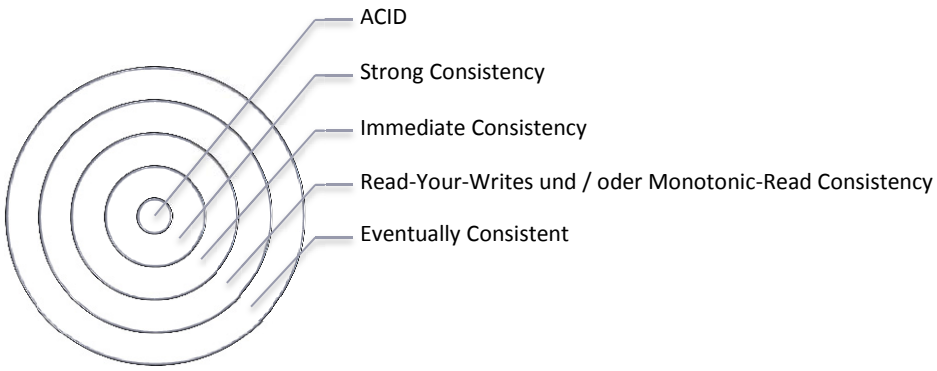


Abbildung 31 Hierarchie wichtiger Konsistenzbedingungen

Diese Konsistenzprinzipien werden wir auch in Bezug auf die Involvierung von Caches und der Architektur von Orestes näher untersuchen. Die Konsistenzgarantien von NoSQL-Datenbanken sind hauptsächlich auf der Ebene von Strong-Consistency und weicheren Bedingungen angesiedelt [Mer10a] (z.B. COUCHDB, RIAK). Bei NoSQL-Datenbanken, die der Konsistenz (C) einen besonders hohen Stellenwert beimessen (z.B. MONGODB und HBASE) werden im Gegenzug die Anforderungen der Verfügbarkeit (A) aufgeweicht [Hur10]. Beiden Typen von NoSQL-Datenbanken teilen also die Robustheit gegenüber Partitionierungen des Netzwerkes. Sie unterscheiden sich jedoch darin, ob alle Clients des Systems stets dieselben Daten sehen (C) und ob der Verlust eines Servers der Verfügbarkeit des Systems keinen Abbruch tut (A).

Das Konzept der NoSQL-Datenbanken lässt sich zusammenfassend charakterisieren als Datenspeicherung unter hoher Resilienz bezüglich der Netzwerkpartitionierung und Knotenverluste, mit ausgeprägter Fähigkeit zur horizontalen Skalierung bei verminderter Datenkonsistenz, Abfrage- und Modellierungsmächtigkeit.

2.5.2 Klassifizierung und Untersuchung

Die NoSQL-Datenbanken entstammen in großer Mehrheit dem Webumfeld, mit dem Ziel, den dort vorherrschenden enormen Speicherbedarf zu befriedigen [MTV10]. Dies macht ihre

Untersuchung für diese Arbeit besonders bedeutsam, da sie weitestgehend REST/HTTP Schnittstellen für den Zugriff vorsehen. Eine Reihe der von ihnen umgesetzten Konzepte sind zudem von generischem Charakter für eine REST/HTTP gestützte Datenübertragung und -Persistierung. Einige der verfolgten Strategien werden wir verbessern und anders gestalten, da sie grundlegenden Überlegungen von REST als Architekturstil zuwiderlaufen, oder diese unzureichend umsetzen. Eine Gegenüberstellung des Orestes-Ansatzes mit Vertretern der NoSQL-Datenbanken wird zudem die Ergebnisse vergleichbar machen.

Die verschiedenen nichtrelationalen Datenbanktypen lassen sich, neben objektorientierten, in vier Kategorien einteilen [BEF+10].

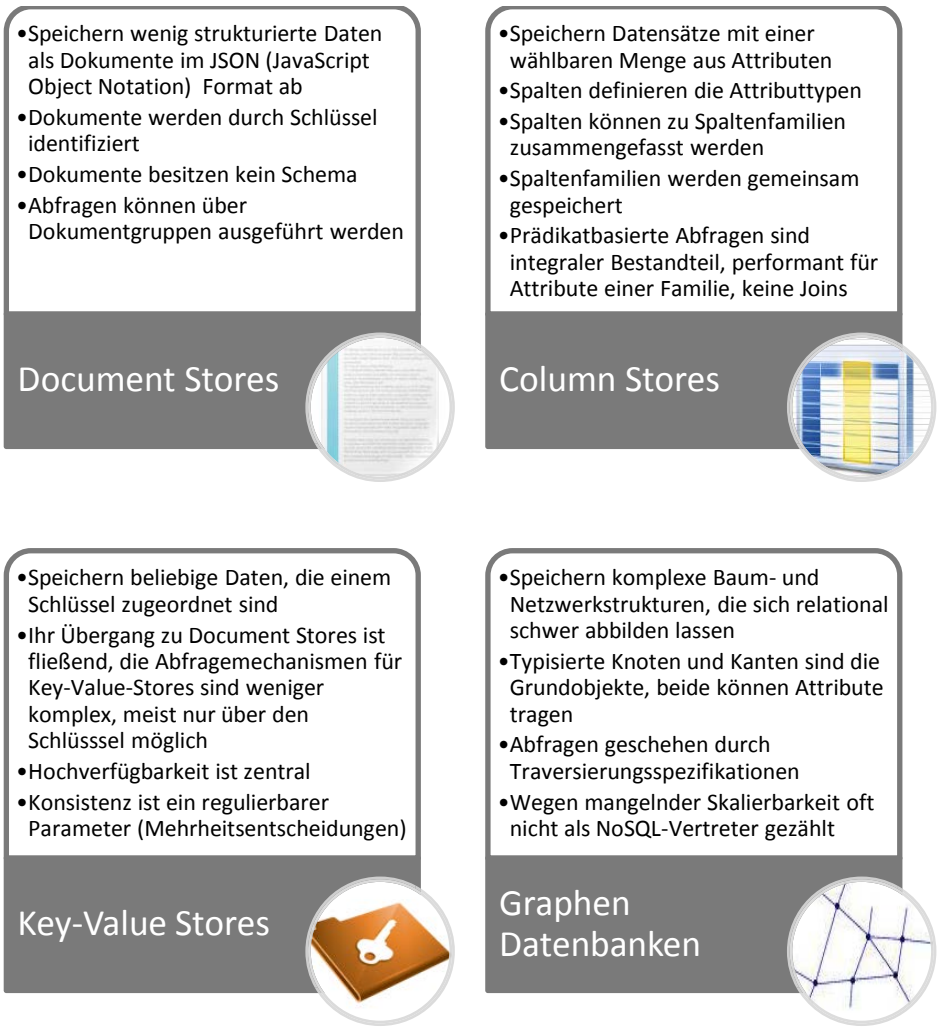


Abbildung 32 Typen von NoSQL Datenbanken

Da NoSQL Datenbanken derzeit keinem definierten Standard folgen, sind die Umsetzungen dieser vier Grundtypen recht unterschiedlich. Deshalb werden wir die verbreiteten Vertreter der Grundgattungen tabellarisch in Anlehnung an [Ell09] gegenüberstellen und anschließend die Konzepte der prominentesten untersuchen.

Typ	Name	Einsatz bei	REST	Abfragen	Replikation
Document Store	COUCHDB	BBC, Ubuntu One	Ja	Map-Reduce	Multi-Master
	MONGODB	Sourceforge, NYT	Ja	JSON-basiertes Query-By-Example	Master-Slave
Column Store	HBASE, BIG-TABLE	GoogleMaps, Youtube	Ja	Map-Reduce	Durch verteiltes Dateisystem
	CASSANDRA	Twitter, Facebook	Ja	Map-Reduce	Master-Slave
Key-Value Store	REDIS	Github, The Guardian	nein	Schlüssel	Master-Slave
	RIAK	Mochimedia	Ja	Schlüssel/ Map-Reduce	Multi-Master
Graphen DB	INFOGRID	Derzeit kaum	Ja	Traversierungs-Spezifikation	Multi-Master

Tabelle 6 NoSQL-Datenbanken

Wie Tabelle 6 zeigt, stellen im Wesentlichen alle verbreiteten NoSQL-Datenbanken eine REST/HTTP Schnittstelle zur Verfügung. Da diese jedoch teilweise einer wenig differenzierten Tunnelung über HTTP entsprechen, werden wir einige ausgewählte konsequente und gemeingültige Grundideen und Patterns gegen teilweise anzutreffende Konzeptionsfehler und Verletzungen des REST-Architekturstils abgrenzen.

2.5.2.1 REST- und Verteilungsarchitektur des Document Stores CouchDB

Apache COUCHDB („**c**luster **o**f **u**nreliable **c**ommodity **h**ardware“) ist ein Open-Source Document Store mit einer REST/HTTP Schnittstelle [And10]. Als universelles Abfrage- und Datenformat kommt JSON zum Einsatz, prozedurale Elemente werden durch JavaScript (ECMAScript) ausgedrückt. Die Grundprinzipien der Document Stores, unter denen COUCHDB als ein Vorreiter gilt [MTV10] gehen auf den Gründer Damien Katz zurück, dessen Vergangenheit als Entwickler des dokumentenorientierten Datenbanksystems Lotus Notes der Firma IBM einen prägenden Einfluss auf die Architektur und Schnittstelle dieses und anderer Document Stores hat.

Die Schnittstelle sieht vor, dass alle JSON-Dokumente (die zu speichernden Objekte) durch eine URI eindeutig identifizierbar sind. Sie enthalten neben der frei wählbaren inhaltlichen Ausgestaltung einen Identifier (die Endung der URI) und eine Versionsnummer („Revision“). Des Weiteren können ihnen unstrukturierte „Attachments“ (beispielsweise ein Bild) und in JavaScript formulierte Update-Validatoren, sowie spezielle Markierungen (z.B. „Replikationskonflikt“) angefügt werden. Operationen an den Dokumenten werden durch HTTP-Operationen an den URI-identifizierbaren Dokument-Ressourcen abgebildet. Um Operationen, die sich über mehrere Dokumente oder Metadaten erstrecken durchführbar zu machen, wurden eigene durch den Präfix „_“ gekennzeichnete, artifizielle Ressourcen eingeführt. Die Struktur des daraus entstehenden Ressourcen- und Operationsgefüges zeigt Tabelle 7 anhand exemplarischer Anfragen.

Operation	Abgebildet durch	Exemplarische Antwort
Datenbanken (Dokumentensammlung) auflisten	**GET** /_all_dbs	[“customers“, “transactions“, “db“]
Datenbank erstellen / löschen	**PUT** / **DELETE** /db	{“ok”: true, …}
Informationen über DB	**GET** /db	{“db_name“ : “Kunden“, …}
Änderungslog der DB	**GET** /db/_changes?since=99	{…}
Dokumente einer DB auflisten	**GET** /db/_all_docs?limit=10	{…, “rows“ : [{…}, {…}]}
Ein Dokument durch seine URI abrufen	**GET** /db/doc	{“_id” : “1”, “_rev” : “xyz”, “myAttribute” : “42”}
Dokumente durch ihre ID abrufen	**POST** /db/_all_docs {“keys“ : [“id1”, “id2”]}	{…, “rows“ : [{…}, {…}]}
Ein Dokument erstellen	**PUT** /db/id3 {“inhalt” : “A”}	{“ok” : true , “id” : “id3”, “rev” : “1-c002”}
Ein Dokument ändern	**PUT** /db/id3 {“inhalt” : “B”, “_rev“ : “1-c002”}	{…, “rev” : “1-c003”}
Ein Dokument löschen	**DELETE** /db/id3	{“ok”: true, …}
Mehrere Dokumente ändern oder erstellen (Bestehende Dokumente werden geändert, sonst Erstellung mit generierter ID)	**POST** /db/_bulk_docs {“all_or_nothing“ : false, {“_id” : “id1”, “a” : ”b” },..}	{“ok”: true, …}
Ein Attachment anfügen	**PUT** /db/doc/img.jpg?rev=5 *Binärdaten*	{“ok”: true, …}
Einmalige, unidirektionale Replikation zwischen CouchDB Instanzen	**POST** /_replicate {“source” : “*URL*/db”, “target” : “db”}	{“ok”: true, “history”: […] }
		Legende: **PUT**-HTTP Verb, {“data“ : …}-HTTP Daten

Tabelle 7 Ressourcen und Operationen bei dem Document-Store CouchDB

Die API von CouchDB enthält die Umsetzung wichtiger REST Patterns. Zum einen werden alle Objekte die im Rahmen der Schnittstelle von Bedeutung sind, in Form von Ressourcen modelliert, auch solche, die lediglich ein logisches, kein physisches Konzept abbilden (z.B. „_bulk_docs“). Auch werden unterschiedliche HTTP-Verben für CRUD-Operationen eingesetzt, was die Möglichkeiten und standardisierte Semantik von HTTP für die Infrastruktur nutzbar macht. Auch die URI-Parameter (z.B. „?limit=10“) werden im Sinne der HTTP-Semantik verwendet, da sie idempotent und sicher sind und die üblichen Filter- und Paginierungsfunktionalität bieten. Auch die Statuslosigkeit der Kommunikation ist umgesetzt, entstehende Zustände werden entweder vom Client gehalten (z.B. Revisionsnummer eines Dokuments) oder serverseitig als Ressource (z.B. neues Dokument) behandelt.

Die Fokussierung auf JSON als universelles Format ist eine konkrete Gestaltungsentscheidung für CouchDB, die zwar bezüglich der Benutz- und Erlernbarkeit vertretbar ist, den Grundsatz der Entkopplung von Repräsentation und Ressource aber verletzt [Til09, S. 14]. Diese Verletzung wird teilweise durch eingeführte Listen-Sammlungsressourcen aufge-

weicht, die als benutzerdefinierte Funktionen iterativ eine Dokumentenmenge in einem alternativen Format ausgeben können (z.B. RSS oder HTML). Eine weitere Lücke in der Konzeption der REST-API ist das Ignorieren von Hypermedia. In CouchDB werden URIs ausschließlich als Identifikator und Illusion einer Dateisystem-Hierarchie genutzt, nicht aber als Möglichkeit zur Referenzierung und Verknüpfung. Auch die Einführung von anwendungsspezifischen HTTP-Verben (COPY, MOVE) widerspricht dem REST-Gedanken aufs schärfste [Til09, S.11]. Zwar vermeidet dieser Ansatz die unspezifische Nutzung der POST-Operation, akzeptiert jedoch als Trade-Off die fehlende Semantik aus Sicht der Infrastruktur und die mangelnde Allgemeingültigkeit und Standardkonformität der Schnittstelle [RiRu07].

Von großer Bedeutung für die Rolle von CouchDB ist auch seine Verteilungsarchitektur und Replikationsbehandlung, bei der Möglichkeiten zum Einsatz kommen, die für Orestes in ähnlicher Weise nutzbar sind. So wird für die Mehrbenutzerkontrolle Multi Version Concurrency Control (MVCC) [Här87] eingesetzt (Abbildung 33), bei der ohne Sperrmechanismen Schreibzugriffe verarbeitet werden. Dabei wird solange die alte Version eines Dokuments ausgeliefert, bis ein neues vollständig geschrieben wurde. Ein solcher Schreibvorgang ist stets gleichbedeutend mit dem Anlegen eines neuen Dokuments mit einer geänderten Versionsnummer, die nach Abschluss des Schreibvorgangs als aktuell markiert wird („Append-Only-Strategy“ [Ho08]). Die Situation ist aus Applikationssicht konzeptuell äquivalent zu einem Szenario, in dem Caches veraltete Dokumente ausliefern, mit der Unterscheidung, dass ein erfolgreicher Schreibzugriff keine Invalidierung des veralteten Cache-Eintrags nach sich zieht, sondern erst durch eine optimistische Commit-Behandlung festgestellt wird, dass ein Datum veraltet ist.

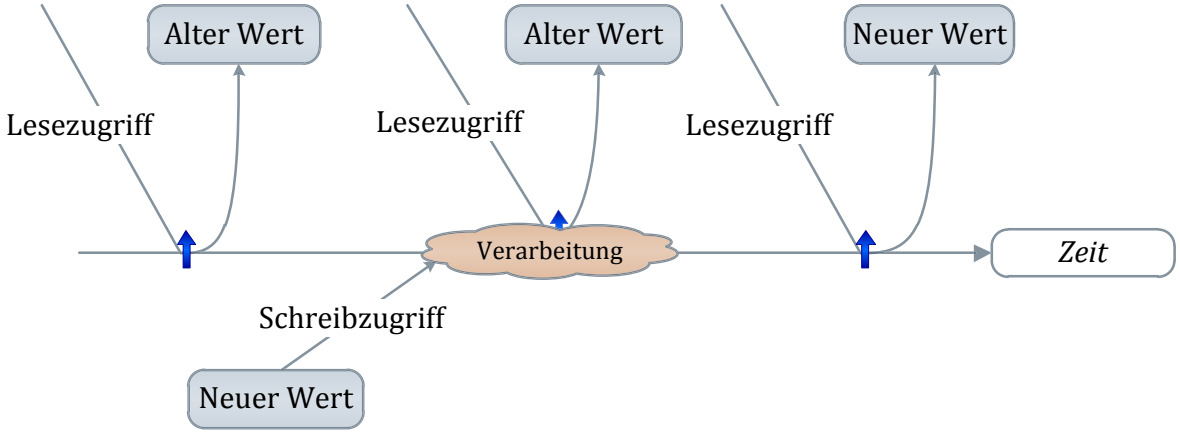


Abbildung 33 Das Prinzip des Multi Version Concurrency Control bei CouchDB

Aus diesem Grund ist es unverzichtbar, dass bei einer Update-Operation die Versionsnummer (in CouchDB Revision genannt) angegeben wird [RiRu07]. Dadurch wird entscheidbar, ob die Änderungsanfrage verarbeitet und das Dokument aktualisiert werden kann, oder ob ein Konflikt aufgetreten ist (HTTP `409 Conflict`). Dieser Fehler muss durch eine applikationsseitige Behandlung behoben werden, meist durch Anfragen der neusten Version und erneute Durchführung der Änderung.

Konfliktbehandlung muss auch im Zuge der Replikation durchgeführt werden. Bei der unterstützten inkrementellen Replikation werden periodisch oder einmalig Dokumente zwischen Instanzen ausgetauscht [RiRu07]. Da diese zwischenzeitlich geändert werden können, sind Konfliktsituationen möglich. Hierbei kommt ein konsistenter Algorithmus zum Einsatz der deterministisch für alle Knoten eine „gewinnende" Dokumentenversion identifiziert und als aktuell markiert. Um den Konflikt behandelbar zu machen, wird für diese und die „unterlegene" Dokumentenversion eine Konfliktmarkierung angefügt, die durch die Anwendungslogik behoben werden muss.

Diese von CouchDB eingesetzten Prinzipien, die viele Grundgedanken des Webs umsetzen, sind für die Konzeption von Orestes interessant. Sie zeigen einerseits Möglichkeiten für das Design einer Operationen- und Ressourcenstruktur für persistente Daten auf und belegen andererseits, dass schwache Konsistenzansprüche, wie sie ebenfalls aus der Verwendung von Web-Caches resultieren durch optimistische Transaktionsbehandlung auch im praktischen Einsatz behandelt werden können. Die Verwendung von Konfliktmarkierungen eröffnet zu dem eine Modellierungsoption, die von einem Replikationsszenario auch auf konkurrierende, nebenläufige Zugriffe mit veralteten Objekten übertragen werden könnte. Hierbei bestünde die Möglichkeit, die Objekte zweier konkurrierender Transaktionen redundant zu speichern und eine applikationsseitige Konfliktauflösung zu fordern.

Die Ausnutzung von Web-Caches setzt CouchDB bisher nur unzureichend um [MTV10]. Anfragen auf Dokumente werden zwar stets mit einem ETag ausgeliefert, jedoch ebenso mit der Caching-Direktive `must-revalidate`. Dies lässt das Potential der Web-Caching Hierarchie weitestgehend ungenutzt, denn jeder Request wird so für eine Revalidierung bis zu dem Origin Server propagiert. Neben der Replikation unterstützt CouchDB Skalierung nur auf Basis eines Sharding-Systems (Lounge). Dieses System verteilt Requests auf einzelne CouchDB-Instanzen und aggregiert ihre Zwischenergebnisse. Es erforderte explizite Konfiguration und dediziert dafür abgestellte Server- und Netzwerkinfrastruktur. Da CouchDB alle architekturellen Vorrausetzungen (*Optimistic Commit*) besäße, um die Web-Caching-Hierarchie in wesentlich größerem Maßstab für die Skalierung von Lesezugriffen zu nutzen, halten wir die Vernachlässigung des Caching-Constraints eine leistungsmindernde Verletzung von REST. Für Orestes werden wir nicht selektiv REST-Prinzipien einhalten, da die Emergenz positiver Leistungseigenschaften von dem gesamten Constraint-Set abhängt [Lac07]. Speziell dem Web-Caching wird häufig der Erfolg zugeschrieben, ein so hervorragend skalierendes System wie das WWW zu konstituieren [GoTo02, S. 135].

2.5.2.2 Abfrageformate und -Mechanismen in Document Stores

Da die inhärent schemalosen Dokumente in ihrer Struktur Objekten stark verwandt sind, ist es für Orestes von Interesse zu untersuchen, welche Abfrageformate für sie entwickelt und im produktiven Einsatz sind. Hierbei kann man die in Abbildung 34 dargestellten Ansätze unterscheiden.

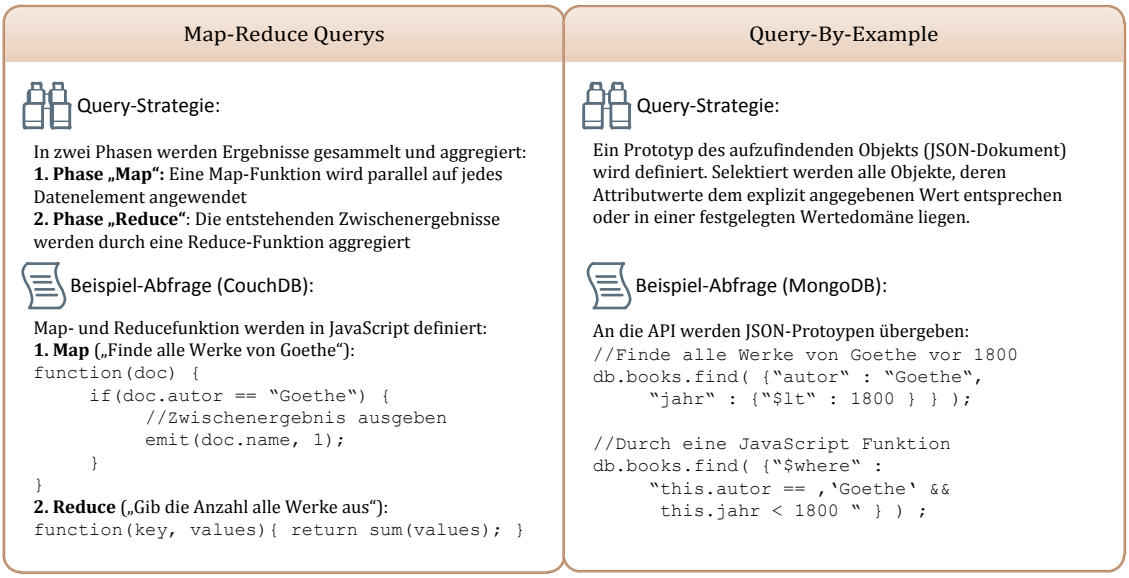


Abbildung 34 Query Mechanismen bei Document Stores

Bei Map-Reduce Querys handelt es sich um ein Verfahren zur verteilten Datenanalyse und -abfrage, das von J. Dean et al. in einem Google-Paper vorgeschlagen wurde [DeGh04]. Seine Daseinsberechtigung und wachsende Verbreitung zieht es aus seiner Skalierbarkeit, insbesondere der Fähigkeit die Map- und Reduce-Operationen auf Rechenknoten eines Clusters zu verteilen. Der so erreichbare hohe Grad an Parallelität liegt darin begründet, dass die Map-Operationen vollständig lokal und isoliert für jedes Datenelement durchgeführt werden können und die Reduce-Operationen auf den getrennten Mengen an Zwischenergebnissen arbeiten. Aus diesem Grund skalieren die Map-Reduce Querys mit dem Ausmaß der Datenpartitionierung und beteiligten Knoten.

Die Benutzung von Map-Reduce Querys vollzieht sich in CouchDB als Definition eines Views, der durch eine in JavaScript formulierte Map- und optionale Reduce-Operation auf einer Dokumentensammlung definiert wird. Das System führt bei einer Abfrage die Map-Funktion parallelisiert für alle Dokumente aus und generiert daraus einen B-Baum, sortiert nach dem von der Map-Funktion ausgegebenen Schlüssel. Auf dieser generierten Datenstruktur, einem Index der auch bei Änderungsoperationen aktuell gehalten wird, aggregiert die Reduce-Funktion die Key-Value-Paare zu einem skalaren Ergebnis. Diese Reduktion kann parametrisiert werden, indem eine Schlüsseldomäne und die Gruppierungsebene (ein Ergebnis pro Schlüssel, oder ein Ergebnis für alle Schlüssel) bei einer Anfrage spezifiziert werden.

Diesem Anfragemechanismus steht in der Praxis vor allem der Query-by-Example (QBE) Ansatz gegenüber, der in MONGODB zum Einsatz kommt und der Formulierung von Ad-hoc Querys dient [Cho10]. Dabei wird die in der Relationenalgebra als Selektion ($\sigma_{\{Prädikat\}}$) bezeichnete Operation definiert durch ein JSON-Objekt. Dieses JSON-Objekt ist gekennzeichnet durch Name-Wert Zuweisungen, die entweder einen konkreten Attributwert des aufzufindenden Objekts angeben, oder eine Einschränkung seines Wertebereichs. Diese Einschränkungen werden durch Vergleichsoperatoren („<“, „>“, etc.) ausgedrückt und die entstehenden Prädikate konjunktiv verknüpft (z.B. „Objekt x ist größer als 20 *und* kleiner als dreißig“). Neben den konditionalen Vergleichsoperatoren wurden zudem Operatoren für Constraints an Existenz, Größe und Datentyp eines Attributwertes, Erfüllung eines regulären Ausdrucks, sowie außerdem benutzerdefinierte, in JavaScript formulierte Selektionsprädikate eingeführt. Die Ausgabe dieser Selektion kann durch eine Order-By Klausel sortiert und die Ausführung durch Query-Optimizer-Hints beschleunigt werden. Die API folgt bei seinem Design und Einbettung in die Programmiersprache dem Fluent-Interface-Pattern, so dass jeder Verfeinerungsschritt der Anfragedefinition das benutzte Objekt zurückgibt und somit Anfragen der Art $db.students.find(\{'address.state' : 'HH'\}).limit(10).sort()$ erlaubt.

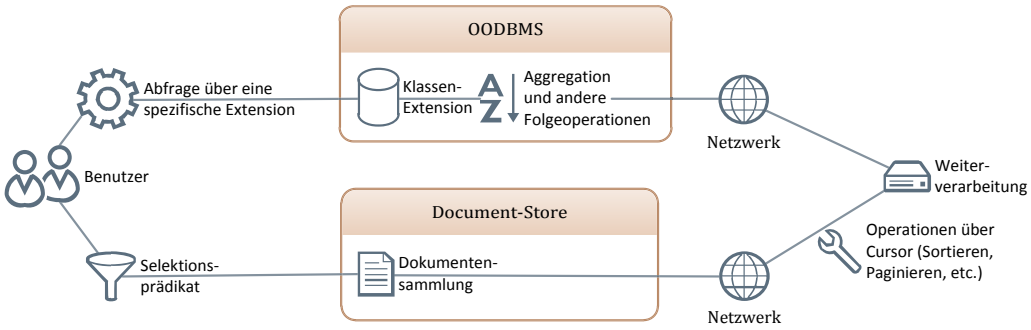


Abbildung 35 Anfrageverarbeitung von OODBMS gegenüber Document-Store (QBE)

Die Nutzbarkeit der vorgestellten Queryformate für Orestes hängt primär von ihrer Übertragbarkeit auf Objekte ab. Die Dokumente eines Document-Stores unterscheiden sich in dieser Hinsicht von Objekten eines OODBMS vor allem durch ihre gänzliche Schemalosigkeit. Dadurch, dass Objekte eines OODBMS Instanzen einer konkreten Klasse sind, entstehen Extensionen von Objekten, die ein gemeinsames, definiertes Schema besitzen. Wie Abbildung 34 zeigt, werden über diesen Extensionen Abfragen ausgeführt, wohingegen bei der QBE-Verarbeitung in Document-Stores stets eine Selektion über eine vollständige, heterogene Dokumentensammlung ausgeführt wird [Cho10]. Der Map-Reduce Ansatz ist für das Orestes-Szenario nur schwer übertragbar, da ihm eine andere Architektur zugrunde liegt – die einzigen verteilten Knoten aus Sicht von Orestes stellen Web-Caches dar. Diese können jedoch keine Queryverarbeitung durchführen, die über das Anfragen eines konkreten Objekts hinausgeht. Anders steht es um den QBE-Ansatz, hier wird auf einem standardisierten Datenformat (JSON) ein kohärentes Schema zur Formulierung von Selektionsprädikaten

definiert, deren Semantik direkt auf Objekte übertragbar ist. Es enthält jedoch keine darüber hinausgehende Queryparameter wie Sortierung und Aggregation.

2.5.2.3 REST-Architektur des Column Stores HBASE

Da das Datenmodell der Colum-Stores (engl. Spaltendatenbanken), dem relationalen Modell wesentlich näher steht als dem objektorientiertem, soll hier nur überblicksartig die Ressourcenstruktur der Schnittstelle von HBASE untersucht werden. Bei HBASE handelt es sich um eine Open-Source Implementierung der populären Google-Big-Table Spezifikation [CDG+06]. Das Datenmodell beschreiben die Autoren zusammenfassend mit den Worten *„Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes*"[BT, S. 1]. Dieses Modell wird in HBASE im Kontext des „Hadoop-Ecosystems" [Clo09], eines Frameworks für verteilte Software, unter der Apache Lizenz entwickelt. Da die Ressourcenstruktur der REST-Schnittstelle in weiten Teilen vom konkreten Datenmodell abstrahiert, lassen sich aus der Art ihrer Konzeption bei HBASE Rückschlüsse über durchführbare und sinnvolle Ressourcenstrukturen ableiten. Interessant ist auch, dass die REST/HTTP-API („Stargate") als ein Ersatz zu einem Remote-Procedure-Call-Framework („Thrift") entworfen wurde, dass als Binärprotokoll direkt auf TCP aufsetzt [AKM07]. In der Hinsicht ist der Ansatz verwandt zu Orestes – eine bestehende Kommunikationsschicht wird durch eine REST Architektur ersetzt. Tabelle 8 zeigt die Ressourcen (als URI-Template), sowie darauf ausführbare Operationen [Pur09].

Ressource	Verb	Bedeutung
/	GET	Abfragen aller Datenbanken
	POST	Tabelle anlegen
/{Tabelle}/schema	GET	Tabellen-Metadaten abrufen
	PUT	Schemaaktualisierung (Spaltentypen und -familien)
	DELETE	Tabelle löschen
/{Tabelle}/disable	POST	Tabelle deaktivieren
/{Tabelle}/enable	POST	Tabelle aktivieren
/{Tabelle}/regions	GET	Abrufen aller Datenpartitionen (von einzelnen Knoten als „Regions" über [key_{start}, key_{end}) verwaltet)
/{Tabelle}/row/{key}	GET	Datensatz mit Schlüssel „key" abrufen
	POST	Datensatz aktualisieren
	PUT	Datensatz ersetzen
	DELETE	Datensatz löschen
/{Tabelle}/Scanner	POST / PUT	Einen Scanner anlegen (entspricht einem Cursor oder Iterator in im relationalen Modell)
/{Tabelle}/Scanner/{id}	GET	Nächsten Wert des erzeugten Scanners abrufen
/{Tabelle}/Scanner/{id}	DELETE	Scanner löschen

Tabelle 8 Ressourcen und Operationen bei dem Column-Store HBASE

Das REST-Interface ist in seiner Struktur sehr einfach, zeigt aber die korrekte Verwendung von HTTP-Verben für CRUD-Operationen. An der Abbildung des Scanner-Konzepts wird

ein häufig eingesetztes Pattern deutlich: durch die POST-Methode wird an einer Ressource eine Subressource angelegt, deren vom Server mitgeteilte URI das neue Ziel für feingranulare Operationen ist. Web-Caching ist keine Komponente der Schnittstelle: Antworten des Servers werden ohne Validatoren (`ETag`, `Last-Modified`) ausgeliefert, wodurch Revalidierungen unmöglich sind. Ferner werden lediglich Anfragen auf einzelne Datensätze (Zellen) als cachebar ausgeliefert, jedoch ohne Steuerungsmechanismen, mit einer pauschalen Gültigkeitsdauer von 4 Stunden [Pur09].

2.5.2.4 REST-Architektur des Key-Value Stores RIAK

RIAK basiert, wie einige weitere Key-Value Stores (z.B. „Voldemort“, „Dynomite“ und „Redis“), auf den Ausführungen des „Dynamo Papers“ [Dynamo] von Amazon Inc. In ihm wird das Grundmodell eines horizontal skalierenden, hochverfügbaren Key-Value Stores mit abgeschwächten Konsistenzbedingungen gezeichnet. Da die Entwickler der REST-API von RIAK auch die Schöpfer eines verbreiteten REST Frameworks für die funktionale Programmiersprache Erlang sind [Phi10], setzt RIAK nach unserer Einschätzung den REST-Architekturstil in seiner Schnittstelle unter den nichtrelationalen Datenbanken am saubersten und konsistentesten um. Das Datenmodell von RIAK ist beschränkt auf Schlüssel/Wert-Paare, bei denen der Schlüssel als Zeichenkette und der Wert durch einen frei wählbaren MIME-Type typisiert ist. Die Schlüssel/Wert-Paare können in „Buckets“ gruppiert und über *N* Knoten repliziert werden. Die Datenzugriffe, die an einen beliebigen dieser Knoten gerichtet werden können, umfassen eine Angabe darüber, wie viele Knoten das Schlüssel/Wert-Paar erfolgreich gelesen ($R \leq N$) bzw. geschrieben ($W \leq N$) haben müssen („Quorum“), damit der Zugriff trotz möglicher Ausfälle und Inkonsistenzen als erfolgreich betrachtet wird. Die Knoten selbst tauschen Informationen über Änderungen und den Zustand des Systems asynchron, durch ein dezentrales *Gossip*-Protokoll untereinander aus. Tabelle 9 zeigt die Ressourcenstruktur und die durchführbaren Operationen für RIAK [Cri10].

Ressource	Verb	Bedeutung
/riak/{bucket} *?keys={bool}&props={bool}*	GET	Abfragen der Metadaten eines Buckets, oder aller in ihm abgelegten Schlüssel (abhängig vom Querystring)
/riak/{bucket}	PUT	Bucketkonfiguration setzen (z.B. Replikationswert N)
	POST	Schlüssel/Wert-Paar mit systemgeneriertem Schlüssel anlegen
/riak/{bucket}/{key}	GET	Wert für einen Schlüssel abfragen, Header können die Serverantwort konditional an eine Versionsnummer, oder ein Änderungsdatum knüpfen
	PUT	Ein neues Schlüssel/Wert-Paar speichern. Einem solchen Paar können zusätzlich benannte Referenzen (Links) auf andere Schlüssel angefügt werden
	DELETE	Schlüssel/Wert-Paar löschen
/riak/{bucket}/{key}/{link}	GET	Durch „Link-Walking“ Verknüpfungen unter den Schlüsseln folgen

/mapred	POST	Einen Map-Reduce Query mithilfe von übergebenen Map- und Reduce-Funktionen durchführen

Tabelle 9 Ressourcen und Operationen bei dem Key-Value Stores RIAK

Besonders deutlich wird die Umsetzung der REST Patterns in der Einbeziehung von Links. Hypermedia, also Verknüpfungen zwischen Ressourcen, sind ein Kernelement von REST und in RIAK ein mächtiges Instrument der Modellierungsschicht. Durch diese unidirektionalen Referenzen wird das wenig ausdrucksstarke Key/Value-Modell um eine neue Option der Beziehungsmodellierung angereichert, wodurch differenziertere Abfragen als das schlüsselbasierte Auffinden möglich werden. Die Einbeziehung von Hypermedia in die REST Schnittstelle stellt also eine zentrale architekturelle Komponente dar, deren Vernachlässigung HTTP um sein Erfolgskonzept berauben würde [Til09, S.10ff.]. In der Umsetzung der übrigen Operationen kommen auch in RIAK die bereits identifizierten Muster zum Einsatz: übergeordnete Sammlungsressourcen fassen Metadaten zusammen, die CRUD-Operationen an den untergeordneten Datenelementen werden durch PUT, GET, POST, DELETE abgebildet, Konzeptressourcen (z.B. „mapred“) modellieren ressourcenübergreifende Operationen.

Von Web-Caching macht RIAK im Rahmen der gegebenen Möglichkeiten Gebrauch: das HTTP-Validation-Modell wird dadurch unterstützt, dass Key/Value-Paare in Antworten stets einen `ETag` führen [Cri10]. Ohne Revalidierung aus Web-Caches befriedigte Antworten verbieten sich durch das Datenmodell von RIAK. Eine quorumsbasierte Konsistenzentscheidung ist nur möglich, wenn die Anfrage bis zu den Daten-Knoten weitergegeben wird.

2.5.2.5 REST-Konzeptionsfehler der Graphendatenbank INFOGRID

Als Graphendatenbank speichert INFOGRID Informationen in Form von Objekten (Knoten) und Beziehungen (Kanten). Die Knoten sind als Ressourcen über URIs identifizierbar. Eine Kernidee von INFOGRID besteht darin, diese Knoten in menschenlesbarer Form (HTML,RSS, etc.) abrufbar zu machen. Dies trägt dem REST Prinzip Rechnung, dass Ressourcen in verschiedenen Repräsentationsformaten verfügbar sein sollten und kann zusätzlich als Mechanismus der Selbstdokumentation eines Datenmodells dienen. Mithilfe sogenannter „Viewlets“ können Repräsentationen von Knoten der Graphendatenbank in verschiedenen Formaten verfügbar gemacht werden. Ziel soll es dabei sein, ein Framework für „RESTful user interfaces“ [Inf09a] bereitzustellen. Dies erfordert Möglichkeiten der Interaktion zwischen Client und Graphendatenbank, um Operationen an den Ressourcen (Knoten) abbilden zu können. Diese werden geschlossen durch POST-Requests abgebildet, deren Request-Body Informationen über Art und Parameter der vorzunehmenden Operation enthält (z.B. Löschen oder Anlegen eines Knotens) [Inf09b]. Dies verstößt gegen ein wichtiges REST-Prinzip: jede Operation sollte auf eine korrespondierende Operation des HTTP-Protokolls abgebildet werden. Dadurch, dass aus Sicht der Web-Caches die Semantik eines POST-Requests undefiniert ist, sind sie nicht in der Lage Cache-Einträge entsprechend der Operation zu behandeln. So ist es beispielsweise ausgeschlossen, dass ein Web-Cache seinen Eintrag für die Repräsentation eines Knotens invalidiert, sobald ein Client ein Lösch-Request versendet.

Dadurch wird eine der größten Stärken von REST konterkariert: die Protokollspezifität - die Verwendung der Mechanismen von HTTP - bleibt unberücksichtigt.

2.5.3 Übertragbare Prinzipien

Abbildung 36 zeigt zusammenfassend die bei der Untersuchung der ausgewählten Vertreter von Document-, Column-, Key-Value-Stores und Graphendatenbanken identifizierten, übertragbaren Gestaltungsmerkmale.

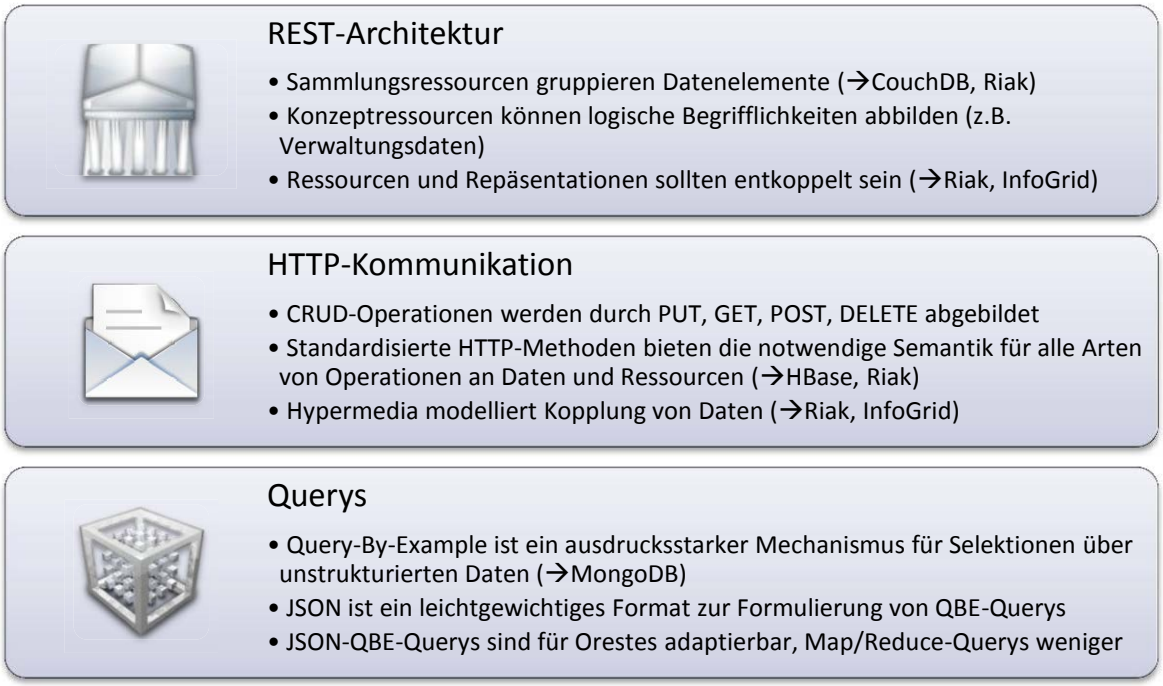


Abbildung 36 Übertragbare Prinzipien der NoSQL-Datenbanken

Bei der Umsetzung von Orestes haben wir den Anspruch, diese Merkmale zu berücksichtigen und zu übernehmen. Damit knüpft Orestes einerseits an den REST-Architekturstil an, berücksichtigt aber andererseits auch die Best-Practices, die sich aufgrund der wachsenden Verbreitung von NoSQL-Datenbanken und ihren REST-Schnittstellen abzeichnen.

3 Ausarbeitung und Architektur des Orestes-Systems

In diesem Kapitel werden wir zeigen, wie sich die in den vorangehenden Kapiteln identifizierten Anforderungen und Eigenschaften erfüllen lassen.

3.1 Ressourcenstruktur

In Anlehnung an die untersuchten NoSQL-Datenbanken und ihre REST-Schnittstellen, haben wir eine analoge Ressourcenstruktur entworfen, die als Hierarchie in Abbildung 37 gezeigt ist. Die wichtigsten *Primärressourcen* (Kernabstraktionen der fachlichen Domäne [Til, S.33]) sind Objekte, Klassenschemata, Transaktionen, Querys und Verwaltungsobjekte. Die Designentscheidung eine Transaktion als materialisierte Ressource zu betrachten, kann ungewöhnlich erscheinen, bietet jedoch entscheidende Vorteile gegenüber einer Modellierung als URI-Parameter oder impliziter Generierung durch Header-Informationen. Alle Interaktionen (z.B. das Allokieren neuer Objekt-IDs) können so an dieser Ressource und geeigneten *Subressourcen* (z.B „{active-transaction}/oids“) vorgenommen werden. Auch das Prinzip der *Konzeptressource*, die nur ein Konzept denominiert und nicht selbst dereferenzierbar ist, nutzen wir. So kann „/db“ nie als eine konkrete Repräsentation vorliegen, verweist jedoch auf alle zugeordneten Subressourcen. Das Prinzip auf verfügbare Subressourcen zu verweisen, erlaubt die dynamische Erschließung der REST-Schnittstelle durch sukzessives Folgen von URI-Referenzen. Aus diesem Grund verweist jede Ressource der Orestes-Ressourcenstruktur auf ihre zugeordneten Unterressourcen (z.B. durch Zurückgeben einer URI-List).

Durch *Listenressourcen* können alle verfügbaren Ressourcen eines dafür vorgesehenen Typs (z.B. „all_objects“) abgerufen werden. Die Mechanismen *Filterung* (z.B. „all_objects“ für bestimmte Klassen) und *Paginierung* (z.B. nur eine eingegrenzte Anzahl von OIDs abrufen) sind für feingranulare Abfragen ebenfalls verankert. Insgesamt erfüllt die Ressourcenstruktur dadurch die vom REST-Architekturstil vorgesehen Kriterien:

- **„Identification of resources“**: Der Client gibt bei jeder Anfrage an, an welcher Orestes-Ressource er eine Operation ausführt (z.B. „/db/class“). Angeforderte Ressourcen werden in einer aushandelbaren Repräsentation zurückgegeben (z.B. URI-List).
- **„Manipulation of resources through representations“**: Durch die vom Orestes-Server zurückgegebenen Repräsentationen kann der Client Änderungen vornehmen und an den Server zurückübermitteln (z.B. ein Objekt im JSON-Format).
- **„Self-descriptive messages“**: Jede Nachricht des Orestes-Servers enthält ausführliche Metadaten (spezifischer Medientyp, Caching-Direktiven, etc.). Bei Fehlern ist zudem eine natürlichsprachliche Beschreibung integriert.
- **„Hypermedia as the engine of application state“**: Der Client kann für jede Ressource mögliche Operationen (OPTIONS-Methode) und Subressourcen (GET-Methode) erfragen. Zudem enthalten Repräsentationsformate und HTTP-Nachrichten Referenzen auf Ressourcen (z.B. Objekte und Transaktionen).

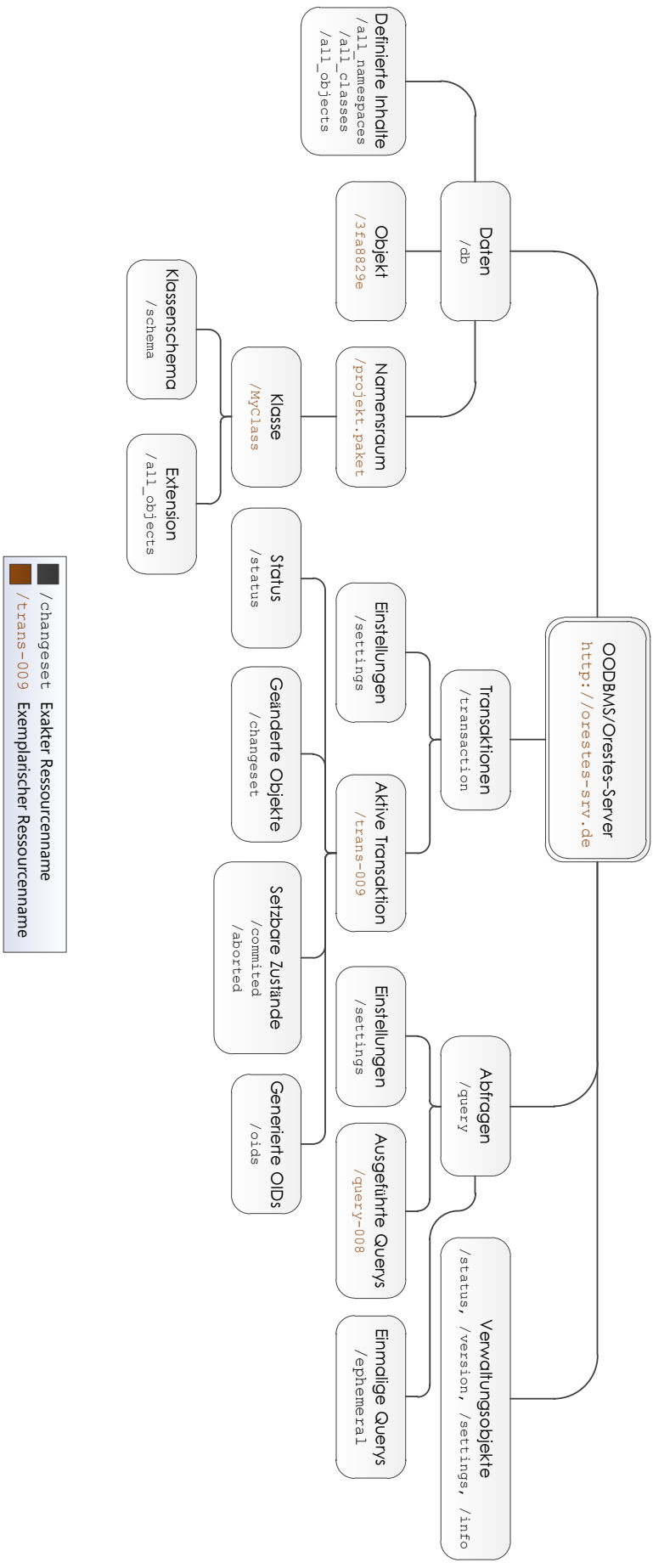


Abbildung 37 Die Ressourcenstruktur in hierarchischer Darstellung

3.2 Caching und Formatdesign

Für die Umsetzung des Orestes-Systems ist die Wahl des Übertragungsformates einer der wichtigsten Kernaspekte. Das Repräsentationsformat muss als ein Kompromiss zwischen der Objektdarstellung in Wirtssprache und der textbasierten Übertragungsform des HTTP-Protokolls konzipiert werden. Zudem soll das Übertragungsformat leichtgewichtig bleiben, um keinen Overhead bei der Übertragung von Objekten zu erzeugen. Der eigentliche Schwerpunkt liegt jedoch darin, dass das benutzte Format die Web-Caches darin unterstützt, Daten effizient zu cachen und dass diese ggf. eine Aktualitätsprüfung durchführen können. Hierbei muss berücksichtigt werden, dass Web-Caches immer nur die Aktualität einer ganzen Ressource überprüfen können und die Überprüfungslogik eines Web-Caches nicht erweiterbar ist. Des Weiteren können Web-Caches nur HTTP-Header auswerten, wodurch die besondere Herausforderung darin besteht, relevante Cachemetainformationen in HTTP-Headern zur Verfügung zu stellen, ohne jedoch die Semantik von HTTP-Headern zu verletzen. Ein weiterer Aspekt, der berücksichtigt werden muss ist, dass benutzerspezifische Daten so wenig wie möglich übertragen werden, da Web-Caches keine Möglichkeit besitzen, Daten für einzelne Benutzer zu speichern.

Hieraus lassen sich nun folgende Schlussfolgerungen für das Formatdesign ziehen:

- Kompaktes Format für die Datenübertragung
- Jedes Datum muss in einem eigenen Repräsentation übertragen werden, damit dieses Gecachet werden kann
- Referenzen sollten nur aus einer URI bestehen, damit Änderungen an den referenzierten Daten keine Änderung an eigenen Daten zur Folge hat
- Relevante Metainformationen, wie die Versionsnummer eines Objektes, müssen in einem cacheverständlichem HTTP-Header übertragen werden
- Es dürfen keine benutzerspezifischen Daten in den Formaten übertragen werden, um die Cachebarkeit der Daten zu gewährleisten

Im weiteren Verlauf werden wir die Einhaltung dieser Schlussfolgerungen stets an exemplarischen JSON-Formaten demonstrieren. Orestes ist jedoch abhängig von den Schlussfolgerungen, nicht von dem konkreten Format.

3.3 Klassen

Im Orestes-System gibt es Klassen wie in gängigen OODBMS. Diese liegen wie in OO-Sprachen üblich, in Packages oder Namespaces. Um Objekte einer Klasse anlegen zu können, muss dem Orestes-System zunächst explizit die Klasse und das Schema bekannt gemacht werden. Außerdem gibt es native Klassen, die bei der Übertragung eine besondere Verarbeitung benötigen und deswegen im Orestes-System stets existieren und nativ unterstützt werden. Sie besitzen im Gegensatz zu benutzerdefinierten Klassen kein Schema und können nicht geändert oder gelöscht werden.

3.3.1 Erstellen und Löschen von Klassen

Damit ein Objekt mithilfe des Orestes-Systems in einer Datenbank gespeichert werden kann, muss zunächst die Klasse des Objektes dem Orestes-System bekannt gemacht werden. Hierfür wird ein PUT-Request auf die neu zu erstellende Ressource durchgeführt. Der Pfad dieser Ressource setzt sich dabei aus dem Präfix „db“, gefolgt von dem Namespace, in dem die Klasse gespeichert werden soll und dem eigentlichen Klassennamen zusammen. Wird z.B. die `Box`-Klasse aus Abbildung 38 angelegt, so ergibt sich der Pfad zu „/db/projekt.package/Box“, wenn die `Box`-Klasse in dem Namespace „projekt.package“ liegt. Das Orestes-System legt, sofern noch keine Klasse in dem angegebenen Namespace vorhanden ist, zunächst die Namespaceressource an. Unter der Namespaceressource wird nun die Klassenressource und unter der wiederum die zugehörigen Unterressourcen bereitgestellt. Diese werden auch noch einmal in Abbildung 37 gezeigt.

Abbildung 38 Anlegen der Box Klasse

Wenn eine Klasse nicht länger benötigt wird, kann diese durch einen DELETE-Request an die Klassenressource entfernt werden. Jedoch muss vorher das Schema explizit gelöscht und sichergestellt werden, dass keine anderen Schemata aus dem Orestes-System noch immer auf diese Klassenressource verweisen.

Namespaceressourcen werden somit implizit erstellt und gegebenenfalls gelöscht, Klassen hingegen müssen explizit erstellt und gelöscht werden, damit ein Klassenschema angelegt werden kann. Die Erstellung und Löschung von Klassen und deren Schemata erfolgt in zwei Schritten, damit andere Klassen die in Schemata referenziert werden zum Zeitpunkt der Schemaspeicherung aus konsistenzgründen auf Existenz geprüft werden können. So kann sichergestellt werden, dass Schemata nur bekannte Typen referenzieren.

3.3.2 Erstellen, Bearbeiten und Löschen von Schemata

Damit ein Schema angelegt werden kann, muss zunächst die Klasse des Schemas und alle anderen Klassen, die das Schema referenziert, dem Orestes-System bekannt gemacht worden

sein. Das Schema selbst wird wie in Abbildung 39 gezeigt, durch ein PUT-Request an die Schemaressource, unter der Klassenressource, angelegt.

Schemata können auf die gleiche Weise, wie sie initial erstellt wurden, mithilfe eines PUT-Request auch nachträglich geändert werden. Dabei muss das Schema mit dem vorherigen Schema vereinbar sein. Dies ist dann auf jeden Fall gewährleistet, wenn bei der Schemaänderung nur Felder hinzugefügt oder gelöscht werden. Zudem kann ein Schema auch komplett geändert werden, wenn keine Instanzen der zu ändernden Klasse in der Datenbank existieren. Akzeptiert der Server die Schemaänderung, so antwortet er mit dem Statuscode `200 OK`.

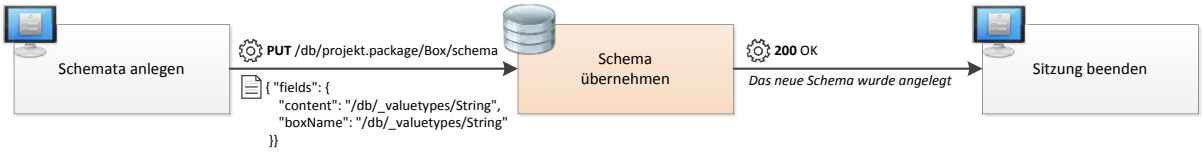


Abbildung 39 Anlegen des Box Schemas

Damit ein Schema gelöscht werden kann, müssen zunächst alle Ausprägungen dieser und aller Kind-Klassen zunächst gelöscht werden. Anschließend kann mit einem DELETE-Request die Schemaressource gelöscht werden. Auch hier antwortet der Server, bei Akzeptieren des Löschvorganges, mit dem Statuscode `200 OK`.

Schemata können, wie die meisten Datenstrukturen, mithilfe von JSON abgebildet. Ein Beispielschema für die `MüllBox`-Klasse ist im Formatbeispiel 1 abgebildet. In der obersten Ebene gibt es zwei Eigenschaften `superclass` und `fields`. Die `superclass` Eigenschaft ist optional und wird bei Klassen weggelassen, die keine Elternklasse haben. Ansonsten referenziert der Wert des `superclass` Schlüsselwortes auf die Elternklasse der Klasse. Das `fields` Schlüsselwort ist ein JSON-Objekt, in dem jeder Schlüssel wiederum ein Feld der Klasse definiert, dessen Wert eine Typ-URI aufweist. Diese Typ-URI identifiziert dabei den Typ des jeweiligen Feldes. Ein Typ ist dabei entweder eine Klasse oder ein nativer Typ.

```
1 {
2   "superClass"   : "/db/projekt.package/Box",
3   "fields"     : {
4     "recycleBin"   : "/db/_nativ.collection/List<String>",
5     "recycleLog"   : "/db/_nativ.collection/List</db/projekt.package/Vorgang>",
6     "recycleCount"   : "/db/_nativ/Integer"
7   }
8 }
```

Formatbeispiel 1 Schema der MüllBox Klasse

In dem JSON-Objekt werden nur neue Felder aufgelistet, die Felder die eine Klasse aus der Elternklasse erbt werden hier nicht erneut deklariert. Die `MüllBox`-Klasse aus Formatbeispiel 1 erbt somit von der `Box`-Klasse, die in dem „project.package“ Namespace liegt. Sie erhält zusätzlich, zu den geerbten Feldern aus der `Box`-Klasse, ein `recycleBin` Feld, dessen Typ ein String ist, ein `recycleLog` Feld, dessen Wert eine Liste von Vorgängen ist und ein `recycleCount` Feld vom Typ Integer.

Dadurch, dass die Schemata in Form einer Ressource unter einer festen URI abgelegt werden, wird gewährleistet, dass Klassenschemata von Caches gecachet und bei Bedarf revalidiert werden können. Schemata werden aber immer mit einem `Cache-Control: max-age=0` Header versehen, um Fehlverhalten auf der Clientseite vorzubeugen, da dieser ansonsten aus einem Cache veraltete Schemata erhalten könnte.

3.3.3 Vererbungshierarchien

Das Orestes-System erlaubt es beliebig tiefe Vererbungshierarchien von Klassen abzubilden. Dabei wird jedoch nur die Einfachvererbung unterstützt. In einem Schema wird die Vererbung durch das `superclass` Schlüsselwort ausgedrückt. Dabei ist dessen Wert immer eine Klassen-URI der Elternklasse. Eine Klasse kann jedoch nicht von einem nativen Typ mit Wertsemantik erben. Beim Anlegen des Klassenschemas, muss die Elternklasse dem Orestes-System bereits bekannt sein.

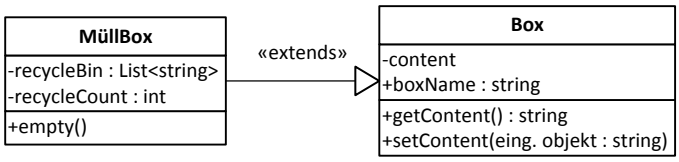


Abbildung 40 UML-Diagramm der Box und MüllBox Klasse

In Abbildung 40 wird noch einmal die Vererbungshierarchie von der `MüllBox`-Klasse aufgezeigt. Das Schema der `Box`-Klasse besitzt demnach kein `superclass` Schlüsselwort, da diese von keiner anderen Klasse erbt.

3.3.4 Assoziationsbeziehungen

Neben der Vererbungshierarchie gibt es in üblichen Objektstrukturen auch Assoziationen zwischen Objekten. Diese werden mit Hilfe von Referenzen in Form von Objekt-URIs ausgedrückt. Solche Objekt-URIs zeigen somit immer auf das zu referenzierende Objekt, und werden in Feldern des Objekts abgelegt. Der Typ dieses Feldes wird im Schema mithilfe einer Typ-URI ausgedrückt. Diese Typ-URI kann dabei eine Klassenressource identifizieren oder einen nativen Typen. Somit können auch native Sammlungen, die wiederum Referenzen enthalten können, referenziert werden.

Um dies zu verdeutlichen, betrachten wir die `recycleLog` Felddeklaration aus dem Formatbeispiel 1 der `MüllBox`-Klasse noch einmal genauer. Zunächst wird mit „/db/_nativ.collection/List“, einer absoluten URI, auf die Typ-Ressource der `List`-Klasse referenziert. Es handelt sich somit, bei Ausprägungen dieses Feldes, um Listen. Da in der Regel aber noch ausgedrückt werden soll, welchen Typ die Elemente in der Liste haben, besitzen diese einen generischen Parameter. Der generische Parameter wird ebenfalls als absolute URI-Referenz angegeben. Der generische Parameter aus der `recycleLog` Felddeklaration, referenziert mit „/db/projekt.package/Vorgang“ auf die Klassen-Ressource der `Vorgang`-Klasse. Insgesamt wird so in dem `MüllBox`-Klassenschema ausgedrückt, dass die `MüllBox`-

Klasse ein Feld mit dem Namen `recycleLog` besitzt, dessen Wert immer eine Liste von `Vorgang`-Objekten ist.

3.3.5 Native Typen, Klassen und Sammlungen

Es gibt zwei Arten von nativen Typen in Orestes. Zum einen gibt es Typen, die gesondert bei der Übertragung behandelt werden müssen, aber dennoch eine OID besitzen und zum anderen primitive Typen oder Klassen mit Wertsemantik, wie beispielsweise Strings oder Zahlen, die keine OID besitzen. Sie besitzen kein Schema, das angelegt, geändert oder gelöscht werden kann. Damit diese Klassen dennoch in einem normalen Klassenschema als Feldtyp referenziert werden können, ist ihnen der Namespace „_native“ zugeordnet. Diese Sonderstellung ist damit zu erklären, dass diese zum einem von Orestes gesondert behandelt werden müssen und zum anderen, dass gängige Programmiersprachen diese Typen ebenfalls nativ unterstützen. Primitive Typen oder Typen mit Wertsemantik haben zusätzlich die Einschränkung, dass von diesen nicht geerbt werden kann.

Ein Feld, dessen Typ ein String ist, also ein Typ mit Wertsemantik, ließe sich beispielsweise in einem Schema mit „/db/_native/String“ referenzieren. In Formatbeispiel 1 ist `recycleBin` ein solcher nativer Typ. Die Typ-URI „/db/_native/Integer“ definiert den Typ somit als `Integer`.

Neben den primitiven Typen gibt es auch noch die Sammlungsdatentypen. Anders als die anderen nativen Typen, werden diese Typen mithilfe einer OID referenziert. Dennoch müssen sie bei der Übertragung gesondert behandelt werden. Um dieser Sonderstellung Ausdruck zu verleihen, liegen diese in dem Unternamespace „_native.collection“. Sammlungen haben aber noch eine Besonderheit gegenüber allen anderen Typen im Orestes-System, das potentiell auch noch für normale Klassen eingeführt werden könnte. Sammlungen sind generisch und haben somit generische Parameter, mithilfe derer ausgedrückt wird, welche Typen in der Sammlung gespeichert sind. Die generischen Parameter werden in Anlehnung an die Java-Notation kommasepariert, zwischen spitzen Klammern angegeben und an den Klassennamen angehängt. In Formatbeispiel 1 ist `recycleLog` eine Liste, dessen Elemente von Typ `String` sind. Es sei hier noch angemerkt, dass die Kombination von primitiven Typen mit nicht primitiven Typen, in Sammlungen nicht erlaubt ist.

Obwohl es nicht möglich ist, zur Laufzeit native Typen anzulegen, gibt es dennoch die Möglichkeit im Orestes-System weitere native Typen hinzuzufügen, die dann ebenfalls die Semantik einer Sammlung oder eines Wertetyps annehmen können. Dies ermöglicht die spätere Implementierung und Ergänzung weiterer, eventuell benötigter Typen.

3.3.6 Fehlersituationen

Während der Datenoperationen am Orestes-System können verschiedene Fehler auftreten. Wenn ein Fehler aufgetreten ist, wird immer mit einem Response geantwortet, der einen passenden Fehlerstatuscode beinhaltet. Zusätzlich wird in den meisten Fehlersituationen im Inhalt des Response eine JSON-Objekt mitgeschickt, das weitere Details beinhaltet, aufgrund

dessen der Fehler aufgetreten ist. Die entstehenden Fehlersituationen und die zugehörigen Fehlercodes sind in der folgenden Tabelle aufgelistet:

Aktion	Fehlercode	Auftreten
Anlegen einer Klasse	409 Conflict	Wenn die anzulegende Klasse bereits existiert.
Löschen einer Klasse	405 Method Not Allowed	Wenn das Schema der zu löschenden Klasse noch existiert.
	409 Conflict	Wenn in der Datenbank noch ein Schema existiert, das die zu löschende Klasse referenziert.
Anlegen und ändern eines Schemas	404 Not Found	Wenn das Schema für eine nicht vorhandene Klasse angelegt wird oder das Schema andere Typen referenziert, die der Datenbank nicht bekannt sind.
Zusätzlich beim Ändern eines Schemas	409 Conflict	Wenn das neue Schema mit dem aktuellen Schema nicht kompatibel ist.
Beim Löschen eines Schemas	404 Not Found	Wenn das Schema einer nicht vorhandenen Klasse gelöscht werden soll.
	409 Conflict	Wenn es noch Objekte der Klasse gibt, deren Schema gelöscht werden soll.

Tabelle 10 Fehler die bei Klassen- und Schemaoperation auftreten können

3.4 Objekte

Neben den Klassen gibt es in dem Orestes-System auch die Ausprägung dieser Klassen in Form von Objekten (Instanzen). Bei dem REST-Schnittstellenentwurf für Objekte ist es besonders wichtig, dass jedes Objekt für seine gesamte Lebenszeit in dem Orestes-System unter einer festen und zugleich eindeutigen URI erreichbar ist, um die Cachebarkeit von Objekten zu gewährleisten. Außerdem müssen Referenzen auf andere Objekte auch in der Repräsentation als Referenzen behandelt werden, damit, wenn sich ein Objekt ändert, nicht auch alle anderen Objekte, die das geänderte Objekt referenzieren, ebenfalls aktualisiert werden müssen. Zudem muss die Version eines Objektes den Caches zugänglich gemacht werden, damit diese gegebenenfalls erkennen können, ob sie eine aktuelle Version eines Objektes besitzen. In Formatbeispiel 2 ist eine Ausprägung der `MüllBox`-Klasse zu sehen, die in dem vorherigen Kapitel

```
1  {
2    "class" : "/db/projekt.package/MüllBox",
3    "/db/projekt.package/MüllBox"    : {
4      "recycleBin"     : "/db/ef8346fac98b"
5      "recycleLog"     : "/db/8347efa98de2",
6      "recycleCount"   : "3"
7    },
8    "/db/projekt.package/Box"    : {
9      "content"    : "/db/89234ab8cd9e",
10     "boxName"    : "/db/9ca87def87ab"
11   }
12 }
```

Formatbeispiel 2 Format eines MüllBox-Objekts

definiert wurde. Auch die Ausprägungen werden, wie die Schemata, in JSON angegeben und besitzen einen ähnlichen Aufbau. Zunächst gibt es ein `class` Schlüsselwort, dessen Wert eine URI-Referenz auf die Klassenressource ist, dessen Instanz dieses Objekt ist. Auf der gleichen Ebene folgen nun Klassen-URIs der Klassen, die in der Vererbungshierarchie der Klasse des Objektes liegen. Dessen Werte sind wiederum JSON-Objekte, die die Werte der Felder repräsentieren die in dem jeweiligen Klassenschema definiert sind.

In dem `MüllBox`-Schema (Formatbeispiel 1) werden die drei Felder `recycleBin`, `recycleLog` und `recycleCount` definiert. Exakt die gleichen Felder, befinden sich wiederum in dem JSON-Objekt unter dem „/db/projekt.package/MüllBox“-Schlüssel des `MüllBox`-Objekts (Formatbeispiel 2). Hier werden nun die Werte der einzelnen Felder für das Objekt abgelegt. So besagt das Schema der `MüllBox`-Klasse, dass ein Objekt dieser Klasse ein `recycleCount` Feld von Typ Integer hat, dessen Wert, aus Formatbeispiel 2, die Zahl *3* ist.

3.4.1 Referenzen, Objekt-IDs und skalare Datentypen

Die Werte der Felder in Objekten sind entweder Referenzen auf andere Objekte, oder skalare Werte, die direkt gespeichert sind.

Eine Referenz auf ein anderes Objekt wird durch eine Objekt-URI des zu referenzierenden Objektes ausgedrückt. Hierfür muss ein Objekt stets unter der gleichen URI erreichbar sein. Um dies zu gewährleisten, benutzt Orestes, wie durch alle OODBMS umgesetzt, Objekt-IDs

für die Identifizierung gleicher Objekte. Das Format von Objekt-IDs ist in Orestes grundsätzlich nicht fest vorgeschrieben, zurzeit jedoch auf eine hexadezimale Zahl beschränkt. Die URI von einer Objektressource eines Objektes setzt sich in Orestes aus dem Präfix „db“, gefolgt von der eigentlichen Objekt-ID zusammen, ganz gleich von welchem Typ das Objekt ist.

In dem Formatbeispiel 2 soll das `recycleBin` Feld auf ein Objekt mit der ID *ef8346fac98b* referenzieren. Hierfür wird in dem Feld die absolute Objekt-URI gespeichert. Da diese sich aus dem Präfix „db“ und der Objekt-ID des zu referenzierenden Objektes zusammensetzt, ergibt sie sich zu „/db/ef8346fac98b“. Auch das `recycleLog` Feld enthält eine absolute Objekt-URI, da auch native Sammlungen, wie Listen referenziell behandelt werden.

Skalare Typen werden, anders als referenzielle Typen, direkt als Wert in dem jeweiligem Feld gespeichert. Hieraus ergibt sich die Notwendigkeit, dass das Orestes-System für jeden skalar übertragbaren Typen einen Konvertiervorgang ausführen muss, um den skalaren Wert in eine JSON Repräsentation zu konvertieren und umgekehrt. Deswegen muss jeder skalare Typ nativ von dem Orestes-System unterstützt werden.

Betrachten wir hierzu das `recycleCount` Feld aus dem Formatbeispiel 2. Dies ist ein skalarer Typ. Er enthält nicht, wie die referenziellen Typen, eine Referenz auf eine andere Ressource, sondern direkt den Wert des Feldes, in diesem Beispiel also den Integer-Wert *3*.

3.4.2 Sammlungsdatentypen

Sammlungen werden von Orestes wie skalare Typen gesondert behandelt. Eine Liste hat, wie andere Objekte, eine Objekt-ID und ist ebenfalls unter einer Objektressource abrufbar. Auch der Inhalt der Objektressource ist dem eines normalen Objektes ähnlich. In der obersten Ebene des JSON-Objektes gibt es ebenfalls ein `class` Schlüsselwort, dass die URI der entsprechenden Sammlungsklassenressource enthält. Zudem gibt es auch hier ein Schlüsselwort mit der Typ-URI, dessen Unterelement wiederum ein JSON-Objekt ist. Dieses JSON-Objekt definiert aber, im Gegensatz zu normalen Objekten, immer nur genau ein Feld `value`, dessen Inhalt eine sammlungsspezifische JSON-Repräsentation der eigentlichen Sammlung ist.

```
1 {
2   "class" : "/db/_native.collection/Set</db/_native/Integer>",
3   "/db/_native.collection/Set</db/_native/Integer>" : {
4     "value" : [
5       "3",
6       "14159"
7     ]
8   }
9 }
```

Formatbeispiel 3 Ein Set-Sammlungsobjekt mit Integers als Werte

In Formatbeispiel 3 ist ein Set-Sammlungstyp abgebildet, dessen Elemente vom Typ `Integer` sind, der durch den generischen Parameter zwischen den spitzen Klammern angegeben ist. Die grobe Struktur ähnelt dem eines Objektes stark, lediglich das `value` Feld, das bei einem normalen Objekt durch das Schema definiert wird, ist hier fest definiert. Dessen Inhalt

ist ein JSON-Array, dessen Elemente genau die Elemente des eigentlichen Sets repräsentieren.

```
1  {
2    "class": "/db/_native.collection/Map</db/…/MüllBox, /db/_native/String>",
3    "/db/_native.collection/Map</db/…/MüllBox, /db/_native/String>" : {
4      "value" : {
5        "/db/ef8346fac98b"  : "Käsebrot",
6        "/db/8347efa98de2"  : "Brotdose",
7        "/db/02fa27731bc9"  : "Kaffeefilter"
8      }
9    }
10 }
```

Formatbeispiel 4 Ein Map-Sammlungsobjekt mit MüllBox Instanzen als Schlüssel und Strings als Werte

Auch Listen und Arrays werden genauso übertragen, nur die URI des `class` Schlüsselwortes wird auf dem entsprechenden zugrundeliegenden Datentyp geändert. Die Werte der Felder können auch Referenzen auf andere Objekte sein, indem dann ein JSON-Array von Objekt-URIs angegeben wird. Dies ist auch in Formatbeispiel 3 zu sehen, denn dort sind die Schlüssel der zu übertragenden Map Instanzen der `MüllBox`-Klasse. Dies ist auch dem ersten generischen Parameter der Map-Deklaration zu entnehmen ist. Der zweite generische Parameter gibt hier den Typ `String` an, womit, wie in Java oder C# üblich, der Typ des korrespondierenden Wertes eines Schlüssel definiert wird. Zudem ist hier der Wert des `value` Feldes nicht wie im vorherigen Beispiel ein JSON-Array, sondern ein JSON-Objekt, um die Schlüssel-Wert-Beziehung einer Map abzubilden.

3.4.3 Versionsverwaltung von Objekten

Damit Web-Caches ebenfalls in der Lage sind, die Version eines Objektes zu erkennen und Objekte mit bestimmten Version zu revalidieren, wird die Version nicht im Message-Body sondern als String in dem `ETag`-Header übertragen. Dadurch können Caches sehr effizient prüfen, ob sie noch eine aktuelle Version eines Objektes besitzen und dieses gegebenenfalls aktualisieren. Orestes benutzt für die ETags die echten Version, die das Datenbanksystem zur Verfügung stellt. So kann auch der Datenbank-Client jederzeit die Version eines Objektes erfragen und die Gültigkeit eines Objektes, das er evtl. aus einem Cache erhalten hat, verifizieren.

3.4.4 Erstellung von Objekten

Wenn der Orestes-Client ein neues Objekt in der Datenbank anlegen möchte, so sendet er ein POST-Request an die „db“ Ressource. In den Content des Requests sendet er das komplette anzulegende Objekt mit. Wenn der Server das Schema für das anzulegende Objekt kennt,

Abbildung 41 Anlegen eines Box Objektes

akzeptiert er den Request mit einem `201 Created` Statuscode und schickt zusätzlich in einem `Location`-Header die generierte Objekt-ID, in Form einer Objekt-URI der angelegten Objektressource, zurück. Zudem wird ein `ETag`-Header mit zurückgeschickt, indem die aktuelle Versionsnummer des erstellten Objektes dem Client mitgeteilt wird. Beide Header sind an dieser Stelle völlig legitim und mit der Semantik von HTTP vereinbar.

In Abbildung 41 wird ein Objekt der `Box`-Klasse angelegt, hierfür schickt der Client ein POST-Request an die „db"-Ressource. Der Server akzeptiert das neue Objekt und weist dem neu anzulegenden Objekt eine eindeutige Objekt-ID zu. Aus der Objekt-ID und dem „db"-Präfix ergibt sich nun die URI der Objektressource, unter dem das angelegte Objekt in Zukunft auffindbar ist. Diesem Pfad teilt der Server dem Client im `Locaion`-Header des Response mit, damit dieser in Zukunft aus anderen Objekten heraus auf dieses Objekt referenzieren kann.

3.4.5 Bearbeitung und Löschung von Objekten

Damit ein Client ein Objekt in der Datenbank ändern kann, muss er dieses zunächst geladen haben. Hierfür reicht ein einfacher GET-Request an die Objektressource des Objektes, das geändert werden soll. Der Client erhält als Response das angefragte Objekt mit einer zugehörigen Version im `ETag`-Header. Der Client kann nun seine Änderungen an dem Objekt vornehmen und diese anschließend dem Server mit einem POST-Request (bei einer partiellen Änderungen) oder einem PUT-Request (bei einer vollständigen Änderung) mitteilen. In beiden Fällen wird die Version des zu ändernden Objektes auf Aktualität geprüft. Hierzu schickt der Client im Request einen `If-Match`-Header, in dem der Client die Version mitteilt, die er zuvor erhalten hat. Ist diese Version immer noch aktuell, wird das Update durchgeführt und der Server antwortet mit dem Statuscode `204 No Content`. Alternativ kann der Client eine Wildcard (*) in dem `If-Match`-Header angeben, um auszudrücken, dass die aktuelle Version des Objektes für den Aktualisierungsvorgang nicht relevant ist. In diesem Fall übernimmt der Server die Änderung ohne das Objekt vorher auf Aktualität zu prüfen. Nach der erfolgreichen Änderung schickt der Server im Response, in dem `ETag`-Header, die aktuelle Version des geänderten Objektes zurück.

In Abbildung 42 wird gezeigt, wie ein Objekt der `Box`-Klasse geändert werden kann. Dafür wird zunächst eine Version des Objektes mit Hilfe eines GET-Request vom Server abgerufen. Nachdem der Client seine Änderungen an dem Objekt durchgeführt hat, schickt er dieses mithilfe eines POST-Request an den Server zurück, indem er in den `If-Match`-Header die Version einträgt, die er von dem Objekt kennt. Der Client macht hier eine partielle Aktualisierung und kann deswegen das nicht geänderte Feld `boxName` aus dem Objekt weglassen. Der Server testet zunächst die Aktualität des Objektes.

- Ist das Objekt noch aktuell, antwortet der Server mit einem `204 No Content` Statuscode und teilt dem Client die neue Version des Objektes mit Hilfe des ETags im Response mit.

- Ist das Objekt hingegen nicht mehr aktuell, weil es z.B. in der Zwischenzeit von einem anderen Client geändert wurde oder der Client mit einem veralteten Objekt gearbeitet hat, antwortet der Server mit dem Statuscode `412 Precondition Failed`. Der Client könnte nun das Objekt erneut mit einem `Cache-Control: max-age=0` Header anfragen, der verhindert, dass er eine veraltete Version des Objektes aus einem Cache erhält.

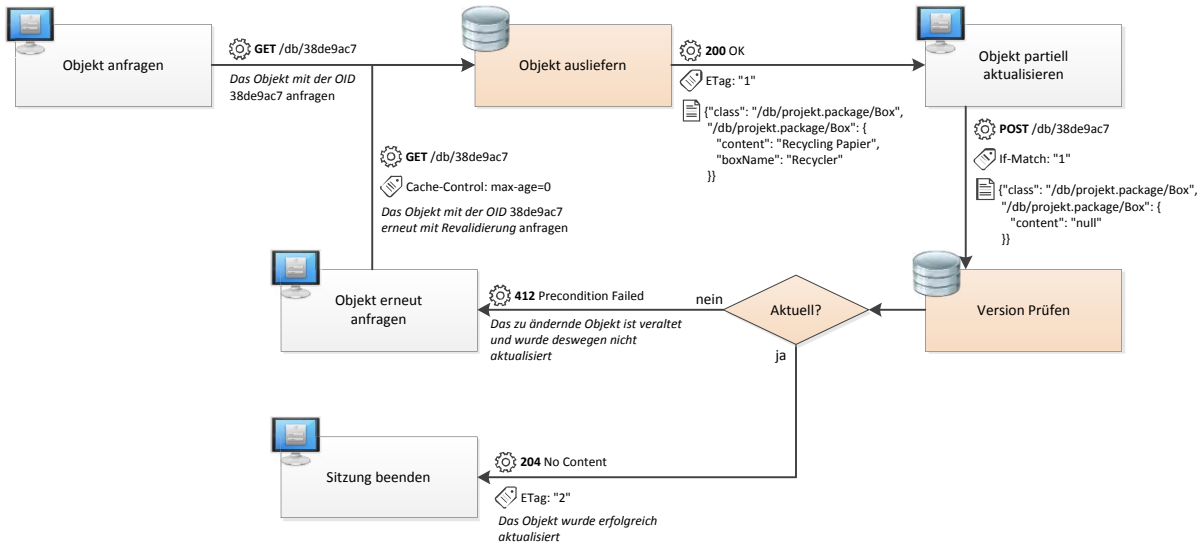


Abbildung 42 Partielle Änderung eines Box Objektes

Da jede Objektänderung mit einem POST- oder PUT-Request dem Orestes-Server mitgeteilt wird, sorgt diese gleichzeitig dafür, dass die Caches, die auf dem Weg zwischen dem Client und dem Server liegen, ihren Cacheeintrag für das geänderte Objekt invalidieren und bei einem erneuten GET-Request an die geänderte Objektressource ihren Cacheeintrag revalidieren. Somit können Cache-Gültigkeitsdauern für Objekte sehr lang sein und Änderungen von Objekten trotzdem sehr schnell von Caches erkannt werden.

Ein Löschvorgang läuft ganz ähnlich wie ein Bearbeitungsvorgang ab. Der Client muss zunächst eine Version des zu löschenden Objektes abfragen. Im zweiten Schritt verwendet der Client nun kein POST- oder PUT-Request, sondern ein DELETE-Request, bei dem er jedoch keinen Inhalt an den Server sendet. Ebenso wie beim Ändern eines Objektes schickt der Client ein `If-Match`-Header mit, um den Server seine Version des Objektes mitzuteilen. Stimmt die Version noch mit der auf dem Server überein, so wird das Objekt gelöscht und der Request wird mit einem `200 OK` Statuscode beantwortet. Ist das Objekt hingegen veraltet, so antwortet der Server, wie auch bei der Änderung, mit dem Statuscode `412 Precondition Failed`.

3.4.6 Revalidierung von Objekten

In Folgenden wird das Laden eines Objektes noch einmal genauer betrachtet. Wenn der Orestes-Client ein Objekt anfragt, das er zuvor schon einmal geladen hat, kann er dieses mit einem sehr performanten Mechanismus, den das HTTP-Protokoll zur Verfügung stellt, auf

Aktualität prüfen. Dasselbe Prinzip wird auch von den Web-Caches ausgenutzt, um die Gültigkeit einer gecacheten Ressource zu prüfen. Wenn also der Client ein Objekt auf Aktualität prüfen möchte, kann er den normalen GET-Request zusätzlich mit einem `If-None-Match`-Header versehen. In diesem kann er dem Server mitteilen, welche Version des Objektes er bereits kennt. Mit einem zusätzlichem `Cach-Control`-Header kann der Client mit `max-age` angeben, wie alt ein zurückgegebenes Objekt höchstens sein darf, wenn dieses aus einem Cache kommt. Die weiteren Web-Caches, die auf dem Weg zum Server passiert werden können, nun überprüfen, ob sie einen Cacheeintrag besitzen, der jünger als das maximal geforderte Alter ist und die Anfrage ggf. ohne weitere Validierung beantworten. Hat der Web-Cache jedoch nur eine veraltete oder keine Version des Objektes, so schickt er den Request weiter. Veraltet heißt in diesem Fall, dass der Cacheeintrag älter als das geforderte Alter ist oder der ETag mit dem des Clients nicht übereinstimmt, da die Web-Cache nicht ermitteln können welcher, ETag eine neuere Version repräsentiert (ETags sind in HTTP nicht untereinander vergleichbar). Gelangt der Request bis zum Server, kann dieser die Version prüfen und dem Client mitteilen, dass seine Version noch aktuell ist oder ihm andernfalls das aktuelle Objekt zurückschicken.

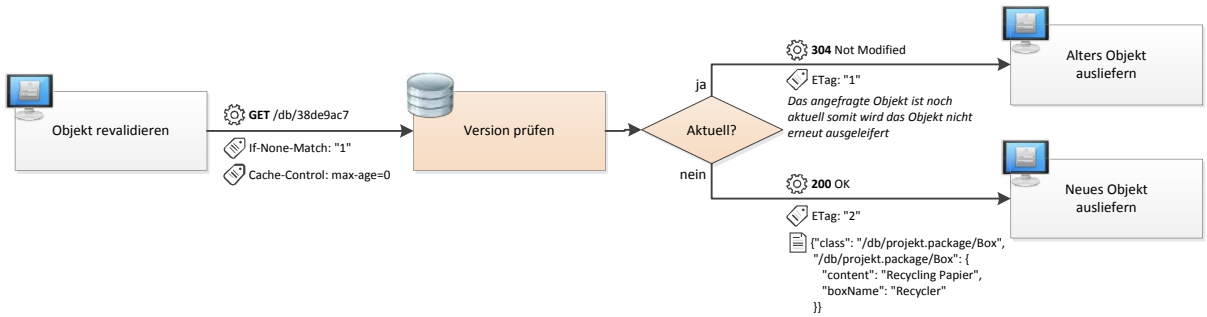


Abbildung 43 Ablauf einer Revalidierung von Objekten

In Abbildung 43 kennt der Orestes-Client bereits eine Version des Objektes mit der OID 7 aus einer vorherigen Datenbankoperation. Um seine Gültigkeit zu verifizieren, schickt er ein Request an die zugehörige Objektressource des Orestes-Servers. Mit Hilfe des `If-None-Match`-Headers drückt der Client in Form eines ETags aus, welche Version des Objektes ihm bekannt ist. Zusätzlich fügt der Client dem Request einen `Cache-Control`-Header mit dem Wert `max-age=0`. Hierdurch verlangt der Client, dass alle Web-Caches, die auf dem Weg zum Server passiert werden, nicht auf sein Request antworten dürfen, ohne das Objekt vorher auf Gültigkeit zu prüfen. Erreicht der Request nun den Server, können hier zwei Fälle auftreten:

- Stimmt die Version, die der Client im `If-None-Match`-Header angegeben hat, nicht mit der aktuellen Version des Servers überein, so schickt der Server das aktuelle Objekt mit dem Statuscode `200 OK` zurück, als ob der Request eine ganz normale Anfrage gewesen wäre.

- Ist das Objekt des Clients hingegen aktuell, so schickt der Server eine Response mit leerem Message-Body und dem Statuscode `304 Not Modified` an den Client zurück.

Der Client erkennt an dem Statuscode, dass sein Objekt noch aktuell ist und kann dieses für die weitere Verarbeitung verwenden. Bekommt er jedoch ein neues Objekt, weiß er, dass seine Version nicht mehr aktuell war und kann somit das zurückgelieferte Objekt nutzen.

Betrachten wir noch eine weitere Art der Objektrevalidierung, die auftreten kann, wenn der Client noch keine Version eines Objektes kennt. Wenn der Client nun einen ganz normalen GET-Request an den Server sendet, um ein Objekt zu laden, durchläuft der Request ebenso die Webcachehierarchie, wie beim vorherigen Beispiel. Besitzt nun ein Cache einen Cacheeintrag für das angefragte Objekt, stellt jedoch fest, dass dieses eventuell veraltet ist, so kann er nach dem gleichen Verfahren vorgehen, wie in Abbildung 43. Der Web-Cache kann dem Request einen `If-None-Match`-Header anfügen, in den er seine Version des Objektes einträgt. Antwortet der Server mit einem `304 Not Modified` Statuscode weiß der Web-Cache, dass sein Eintrag noch gültig ist und kann dem Client mit dem Statuscode `200 OK` antworten. Der Content, der an den Client in diesem Fall zurückgegeben wird, ist dann einfach der Cacheeintrag des Web-Caches.

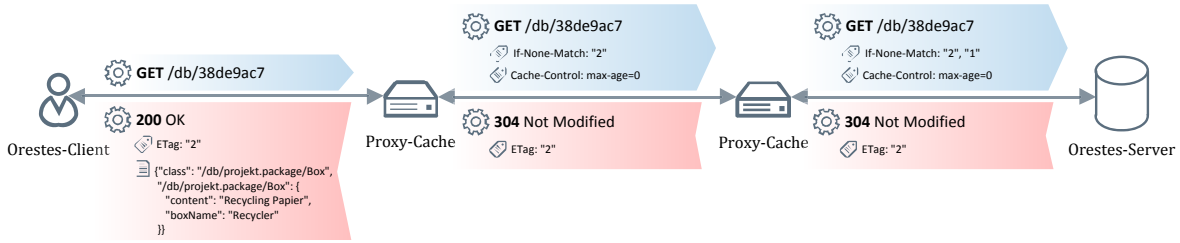


Abbildung 44 Revalidierung eines Objektes in den Web-Caches

Dadurch, dass der `If-None-Match`-Header mehrere ETag Angaben erlaubt, entsteht noch eine weitere Art der Revalidierung, die in Abbildung 44 aufgezeigt ist. Der Client stellt einen gewöhnlichen GET-Request an den Server, um das Objekt mit der OID *38de9ac7* zu laden. Dieser Request erreicht nun den ersten Web-Cache, der bereits eine Version des Objektes kennt, diese muss er jedoch erst revalidieren, bevor er es ausliefern darf. Somit fügt der Web-Cache, wie im vorherigen Beispiel, den Request einen `If-None-Match`-Header mit seiner bekannten Version *2* hinzu. Dieser Request erreicht als nächstes den nächsten Web-Cache in der Webcachehierarchie. Dieser Web-Cache stellt ebenfalls fest, dass er eine Version des angefragten Objektes kennt, diese jedoch nicht mit dem des vorherigen Caches übereinstimmt, da sein ETag nicht mit dem des anderen Web-Caches übereinstimmt. Da für ETags im HTTP-Protokoll keine Größer-Gleich-Relation definiert ist, kann der Web-Cache nicht erkennen, welche der beiden Versionen aktueller ist. Deswegen ist es erlaubt, in dem `If-None-Match`-Header mehrere ETags anzugeben. Somit trägt der zweite Web-Cache seinem ETag, der die Version *1* repräsentiert, ebenfalls in dem `If-None-Match`-Header ein.

Im nächsten Schritt erreicht der Request den Orestes-Server. Dieser prüft ob einer der ETags noch gültig ist und antwortet, wie im vorherigen Beispiel, mit einem `304 Not Modified` Statuscode, da einer der ETags noch eine gültige Version aufweist. Auch hier gibt der Server im Response an, welcher ETag noch gültig ist, denn dieser teilt den Web-Caches mit, welche der bekannten Versionen tatsächlich die gültige Version ist. Der erste Web-Cache, den der Response erreicht, kannte eine Version des Objektes mit der Version *1*. Dieser weiß somit, dass seine Version des Objektes nicht mehr aktuell ist und markiert seinen Cacheeintrag als ungültig, da er keine aktuelle Version des Objektes im Response erhalten hat. Der Response erreicht danach den zweiten Web-Cache, der aus dem `304 Not Modified` Statuscode, wie im vorherigen Beispiel, einen normalen Response aus seinem Cacheeintrag generiert und diesen dem Client zurückschickt.

Der gleiche Ablauf funktioniert auch, wenn der Client, wie im ersten Beispiel, schon eine Version des Objektes kennt und von vornherein einen Request mit einem `If-None-Match`-Header absendet. Auch hier dürfen die Web-Caches ihre bekannten Versionen dem Request im `If-None-Match`-Header anfügen.

3.4.7 Fehlerstituationen

Auch dieses Kapitel wollen wir mit einem Überblick über alle Fehler, die bei Objektoperationen auftreten können, abschließen. Die entstehenden Fehlersituationen, mit ihren zugehörigen Fehlerstatuscodes, sind in der Tabelle 11 aufgezeigt.

Aktion	Fehlercode	Auftreten
Anlegen oder Bearbeiten eines Objektes	404 Not Found	Wenn das Schema für die Klasse des anzulegenden Objektes nicht existiert.
	409 Conflict	Wenn das Schema mit dem Objekt nicht kompatibel ist.
Zusätzlich beim Ändern eines Objektes	404 Not Found	Wenn das zu ändernde Objekt nicht existiert.
	412 Precondition Failed	Wenn das zu speichernde Objekt veraltet ist.
Löschen eines Objektes	404 Not Found	Wenn das zu löschende Objekt bereits gelöscht wurde oder es nicht existiert.
	412 Precondition Failed	Wenn das zu löschende Objekt veraltet ist.

Tabelle 11 Fehler die bei Objektoperationen auftreten können

3.5 Anfrageoperationen

Orestes unterstützt, neben den normalen (OID-basierten) Punktzugriffen auf Objekte, auch das Konzept von Querys an. Da diese aber gerade sehr stark von dem OODBMS hinter dem Orestes-Server abhängen, definiert Orestes kein eigenes Queryformat. Orestes kapselt lediglich die datenbankspezifischen Queryarten, sodass diese ebenfalls über HTTP übertragen und Result-Sets der ausgeführten Querys gecachet werden können. Das Format von Result-Sets ist stets gleich aufgebaut und beinhaltet immer die Objekt-URI und dessen Version. Im Formatbeispiel 5 ist eine JSON-Repräsentation eines solchen Result-Sets zu sehen. Die Schlüssel des JSON-Objektes sind die Objekt-URIs und der jeweilige Wert dessen Version. Mit Hilfe dieses Result-Sets kann der Client die entsprechenden Objekte abrufen und zugleich nachvollziehen, auf welche Version des Objektes die Query zutraf.

```
1 {
2     "/db/1"  : "4",
3     "/db/2"  : "5",
4     "/db/3"  : "2"
5 }
```

Formatbeispiel 5 Ein Result-Set eines ausgeführten Querys

3.5.1 Kapselung proprietärer Anfragesprachen

Da eine Datenbank durchaus mehrere unterschiedliche Anfragesprachen unterstützen kann, werden die verschiedenen Queryformate mit Hilfe des `Content-Type`-Headers identifiziert. Der Content des Request ist dabei komplett für die Queryrepräsentation vorgesehen und somit völlig von dem Orestes-System gekapselt. Die Repräsentation der Querys kann von einem serialisierten Java-Objekt bis zum einfachen JSON-Objekt alle Formate annehmen. Es kann aber vorkommen, dass ein Server beispielsweise zwei verschiedene Querytypen unterstützt und für beide eine JSON-Repräsentation zur Übertragung verwendet. Der Orestes-Server könnte in diesem Fall nicht alleine am Medientypen „application/json“ erkennen, um welchen Querytypen es sich handelt. Deswegen werden für Querys herstellerspezifische Medientypen verwendet. Wäre einer der beiden verwendeten Querys, z.B. ein MongoDB Query, so kann der Medientyp „application/vnd.mongodb+json“ verwendet werden.

3.5.2 Verzögerte und unmittelbare Querys

In Orestes gibt es zwei Arten, wie eine Query ausgeführt werden kann. Es gibt zum einen die unmittelbaren Querys, bei denen die Query einmal ausgeführt wird und anschließend das Ergebnis zurückgeliefert wird. Das Result-Set solcher Querys ist nicht cachebar, zumal das Ergebnis nicht noch ein weiteres Mal abgerufen werden kann. Eine solche Query wird mit einem POST-Request an die Query-Unterressource „/query/ephemeral“ eingeleitet. Der Server antwortet daraufhin mit einem Statuscode `200 OK` und der Response enthält das Result-Set der ausgeführten Querys.

Die verzögerten Querys besitzen eine längere Lebensdauer. Diese werden mit dem gleichen Request, wie die unmittelbaren Querys, eingeleitet, jedoch direkt an die „/query“-Ressource gesendet. Der Server führt die Query aber zunächst noch nicht aus, sondern legt eine neue Unterressource unter der Query-Ressource an und antwortet mit einem `202 Accepted`

Statuscode. In dieser Unterressource wird nun das eigentliche Result-Set der Querys abgelegt, das der Client anschließend mit einem GET-Request an dieser Unterressource abrufen kann. Da Result-Sets durchaus sehr groß sein können, gibt dies dem Client die Möglichkeit, nur einen Teil des Result-Sets abzurufen und zum späteren Zeitpunkt einen anderen Teil, ohne die Query erneut auszuführen. Hierfür kann der Client in dem Requestquerystring die Parameter „from“ und „to“ verwenden, um den zurückzugebenen Ausschnitt des Result-Sets zu spezifizieren. Zudem können solche Result-Sets auch von Caches gespeichert werden und bei einem erneuten Aufruf ausgeliefert werden. Des Weiteren kann der Client mit einem leeren PUT-Request an die Unterressource eine erneute Auswertung der Query auslösen, so dass die Ergebnismenge des Result-Sets aktualisiert wird.

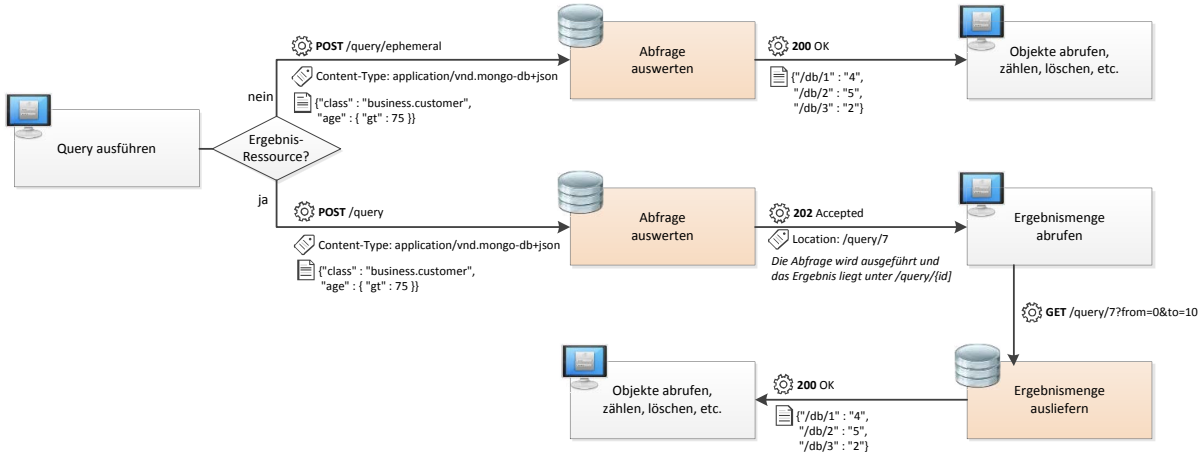


Abbildung 45 Ablauf von verzögerten und unmittelbaren Querys

Betrachten wir abschließend das Beispiel aus Abbildung 45, das die Unterschiede der beiden Queryarten verdeutlicht. Der Client muss sich zunächst entscheiden, ob er eine Ergebnisressource angelegt oder ein einmaliges Ergebnis zurückgegeben werden soll. Der Client möchte eine Mongo-Db Query ausführen. Dazu schickt der Client im Request-Body eine Mongo-Db-Query an den Server. Um den Querytyp zu identifizieren, gibt er im `Content-Type`-Header den Medientyp „application/vnd.mongo-db+json“ an. Mit dem „json“ Suffix kennzeichnet der Client zusätzlich, dass der mongo-db-Query in einer JSON-Repräsentation vorliegt.

- Wenn dem Client ein einfaches Result-Set reicht, so schickt er den POST-Request an die „/query/ephemeral“-Ressource. Der Server verarbeitet daraufhin die Query und schickt im Response ein JSON-Result-Set mit den gefundenen Objekten zurück.
- Wenn der Client die Query jedoch öfter ausführen möchte oder zunächst nur einen Teil des Result-Sets abrufen will, so schickt der Client den POST-Request direkt an die „/query“-Ressource. Der Server legt daraufhin die Query-Unterressource mit der QID *7* an und teilt diese dem Client im `Location`-Header des Response mit. Der Client ruft nun mit einem GET-Request die ersten 10 Objekt-URIs aus dem Result-Set ab. Hierfür gibt er im Request zusätzlich die Requestqueryparameter `from=0` und `to=10` an. Da das Result-Set aber nur drei Objekt-URIs enthält, bekommt der Client im Response nur diese drei zurückgegeben.

3.5.3 Fehlersituationen

Bei Querys können zum einen queryspezifische Fehler bei der Ausführung auftreten, weil z.B. Objekte einer nicht existierenden Klasse abgerufen werden sollen oder die Syntax der Query nicht akzeptabel ist. Den einzigen Fehler, den Orestes aber direkt zurückliefert, ist ein `415 Unsupported Media Type` Statuscode, wenn der Server den Querytyp nicht unterstützt.

3.6 Transaktionen

Eine weitere Schnittstelle, die Orestes zur Verfügung stellt, sind Transaktionen, um, wie in Datenbanken üblich, Objektänderungen an mehreren Objekten atomar als logische Verarbeitungseinheit durchzuführen. Dabei wird von Orestes nur eine optimistische Commitbehandlung unterstützt, da aufgrund der Caches nicht jeder Lesezugriff den Server erreicht und somit gelesene Objekte nicht gesperrt werden können. Auch Transaktionen werden in Form von Ressourcen verwaltet. Hierfür wird, ähnlich wie bei Objekten, eine eindeutige Transaktions-ID verwendet, die zugleich Bestandteil der Transaktions-URI ist. Die URI setzt sich aus dem „transaction“ Präfix, gefolgt von der eigentlich Transaktions-ID zusammen. Also eine Transaktionsressource mit der TID *3* wäre somit unter der URI „/transaction/3“ erreichbar. Grundsätzlich werden alle transaktionsspezifischen Operationen an der „/transaction“ Ressource oder an einer ihrer Unterressourcen ausgeführt, die auch in Abbildung 37 gezeigt werden. Aktive Transaktionen sind dabei grundsätzlich nicht an einen bestimmten Client gebunden und sind somit von jedem Client über die entsprechende Transaktionsressource ansprechbar. Dies ist ein Resultat der zustandslosen Kopplung vom HTTP-Client und dem HTTP-Server. Es ist aber durchaus denkbar, durch eine Authentifizierungsschicht, eine Transaktion nur einem bestimmten Client zugänglich zu machen.

3.6.1 Beginn einer Transaktion

Eine Transaktion wird von einem Client, wie in Abbildung 46 aufgezeigt ist, mit einem POST-Request an die Transaktionsressource „/transaction“ initiiert. Dabei darf ein Client durchaus auch mehrere parallele Transaktionen gleichzeitig starten und nutzen. Das Initiieren einer neuen Transaktion bestätigt der Server mit dem Statuscode `201 Created` im Response. Zusätzlich gibt der Server, ganz ähnlich wie beim Anlegen neuer Objekte, im `Location`-Header des Response, die URI der neu angelegte Transaktionsressource zurück. In dem Beispiel wurde also eine Transaktion mit der TID *4* für den Client angelegt. Mit Hilfe dieser Transaktionsressource kann der Client nun Änderungen an dem Datenbestand im Rahmen dieser Transaktion durchführen.

Abbildung 46 Initiieren einer neuen Transaktion

3.6.2 Datenoperationen im Rahmen einer Transaktion

In einer Transaktion können nur Objekte gelesen, erstellt, geändert und gelöscht werden, und Querys ausgeführt werden. Es ist also nicht möglich, Schemata in einer Transaktion atomar zu aktualisieren, oder zu erstellen. Damit die Änderungen an Objekten im Rahmen einer Transaktion auch von den Caches möglichst schnell erkannt werden, müssen die Än-

derungen an den Objektressourcen durchgeführt werden, obwohl die Transaktion evtl. gar nicht erfolgreich abgeschlossen wird.

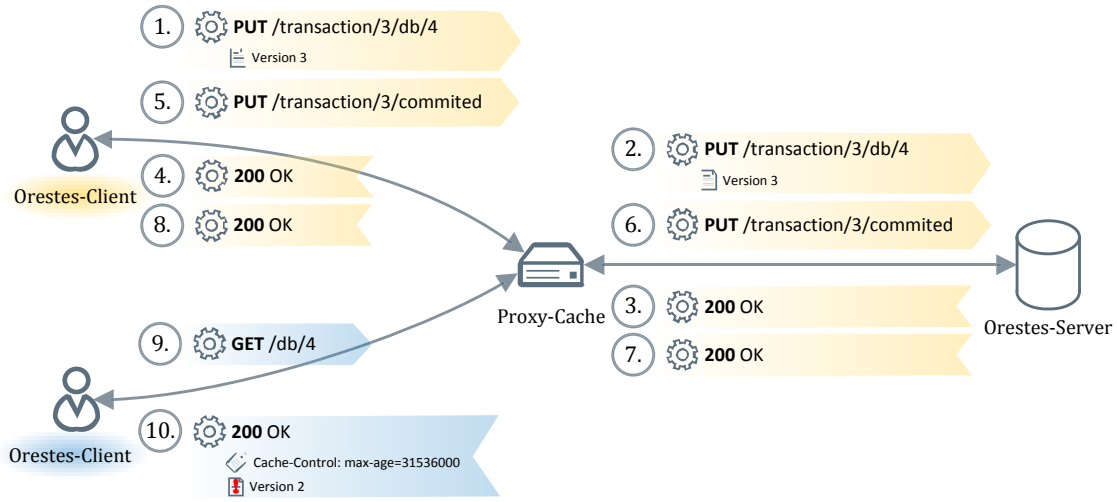


Abbildung 47 Fehlverhalten des Caches, wenn in Transaktionen eigene Objektressourcen verwendet werden

Das Beispiel in Abbildung 47 soll dies verdeutlichen. Wenn also ein Objekt mit der OID *4* in Rahmen einer Transaktion mit der TID *3* geändert werden soll, so wäre es theoretisch möglich dies mit Hilfe der Ressource „/transaction/4/db/3“ auszudrücken. Da das Ziel von Orestes jedoch ist, die Cachezeiten von Objekten möglichst lang zu halten, Änderungen jedoch möglichst schnell zu erkennen, führt dies zu einem Problem. Caches können die Beziehung zwischen der ursprünglichen Ressource „/db/4“ und der Transaktionsressource „/transaction/3/db/4“ nicht herstellen. Betrachten wir hierfür zunächst den Zustand des Systems, nachdem der achte Schritt abgeschlossen ist. Der erste Client hat die Transaktion mit der TID 3 erfolgreich abgeschlossen und der Server hat die Änderung auf der Objektressource „/db/4“ angewendet. Da der Web-Cache zwischen Client und Server aber eine Version des Objektes zwischengespeichert hat, erhält der zweite Client im zehnten Schritt eine veraltete Version des Objektes, da dieser die Änderung der Objektressource unter „/db/4“ nicht registriert hat. Außerhalb von Transaktionen führt dies zu keinem Problem, da die Änderungen direkt an der Ressource „/db/4“ mit PUT oder POST durchgeführt werden, wodurch die Web-Caches ihre Cacheeinträge invalidieren.

Hieraus lässt sich nun die Konsequenz ziehen, dass Änderungen in Rahmen von Transaktionen auch direkt an den original Ressourcen unter „/db/{oid}“ durchgeführt werden müssen. Somit können Web-Caches Änderungen von Objekten in einer Transaktion schneller registrieren und ihre Cacheeinträge ggf. invalidieren. Natürlich kann hierdurch immer noch nicht vollständig garantiert werden, dass alle Web-Caches ihre Einträge invalidieren. Dies führt aber zu einer Reihe von neuen Problemen, für die eine Lösung gefunden werden muss.

Betrachten wir, unter Berücksichtigung der neu gewonnenen Erkenntnisse, noch einmal die Situation aus Abbildung 47 (ein Objekt wird in einer Transaktion aktualisiert). Da nun die gleiche Ressource aus der Transaktion angesprochen wird, bekommt der Web-Cache durch den PUT oder POST-Request mit, dass sein Cacheeintrag nicht mehr gültig ist. Wenn jetzt

ein anderer Client dieses Objekt anfragen würde, so würde der Cache seinen Eintrag revalidieren. Da die Transaktion, die das Objekt geändert hat, aber evtl. noch gar nicht abgeschlossen ist, würde der Server für den anzufragenden Client wieder das alte Objekt ausliefern. Der Cache würde also seinen Eintrag weiterhin für gültig halten. Würde die Transaktion dann aber erfolgreich abgeschlossen werden, hätten wir die gleiche Situation, wie zuvor in Abbildung 47. Um dies zu vermeiden, wird von dem Moment an, wo das Orestes-System von einer evtl. bevorstehenden Änderung eines Objektes weiß, bis zum Abschluss der ändernden Transaktion, das Objekt mit einem `Cache-Control: must-revalidate` Header, ausgeliefert. Dieser sorgt dafür, dass Caches das Objekt bis zum Abschluss der Transaktion revalidieren und anschließend die Änderung mit der ersten Folgeanfrage übernehmen. Das bedeutet also, dass alle Objekte, die während einer Transaktion geändert werden, nicht ohne Revalidierung augeliefert werden dürfen, bis diese abgeschlossen ist.

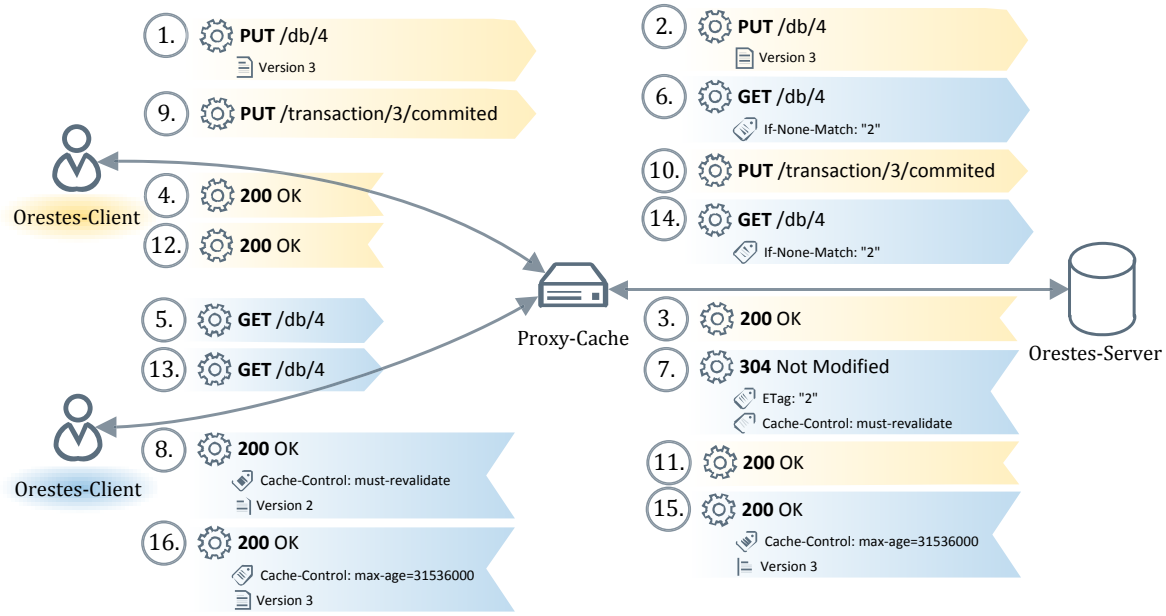


Abbildung 48 Behandlung eines Objektes, nachdem es in einer aktiven Transaktion geändert wurde

Betrachten wir zur Verdeutlichung Abbildung 48, die die Situation des vorherigen Beispiels mit den neu gewonnenen Erkenntnissen darlegt. Wir sehen, dass der Web-Cache, im Gegensatz zum vorherigen Beispiel aus Abbildung 47, in Schritt 6 das angefragte Objekt erst revalidiert, bevor er es an den unteren Client ausliefert. Dies liegt daran, dass der Web-Cache seinen Cacheeintrag durch den vorangegangenen PUT-Request, des oberen Clients (Schritt 1), invalidiert hat. Im Response (Schritt 7) wird dem Cache mitgeteilt, dass das Objekt noch aktuell ist, aber da dieses sich voraussichtlich bald ändern wird, enthält er zusätzlich ein `Cache-Control: must-revalidate` Header, der den Cache dazu auffordert, die Revalidierung auch weiterhin durchzuführen. Nachdem der obere Client im neunten Schritt die Transaktion abgeschlossen hat, übernimmt der Server die Änderung des Objektes. Der untere Client der im 13. Schritt erneut das Objekt abruft, erhält nun die neue Version des Objektes (Schritt 16), da der Web-Cache zuvor instruiert wurde, das Objekt immer erst zu revalidieren. Außerdem wird in Schritt 15 dem Web-Cache, mit dem neuen `Cache-Control-`

Header im Response mitgeteilt, dass dieser das Objekt ab sofort wieder ohne Revalidierung ausliefern darf, da zurzeit keine weiteren aktiven Transaktionen das Objekt ändern werden.

Ein weiteres Problem entsteht dadurch, dass der Client nicht mit Hilfe der URI ausdrücken kann, ob und zu welcher Transaktion eine Objektänderung gehört. Deswegen wird die zugehörige Transaktion, mithilfe des bald standardisierten `Link`-HTTP-Headers [Not10c], im Request angegeben, wie in Abbildung 49 im Request des Client zu sehen ist. Durch ihn können Beziehungen zwischen verschiedenen Ressourcen ausgedrückt werden. Hierdurch kann der Server erkennen, zu welcher Transaktion eine Objektänderung gehört und die Objektänderung im Rahmen dieser Transaktion durchführen. Zudem bekommen die Web-Caches die Änderung der Ressource ebenfalls mit und können ihren Cacheeintrag invalidieren. Auch beim Absetzen eines Querys gibt der Client im `Link`-Header die Transaktion an, um einen Query im Rahmen einer Transaktion auszuführen.

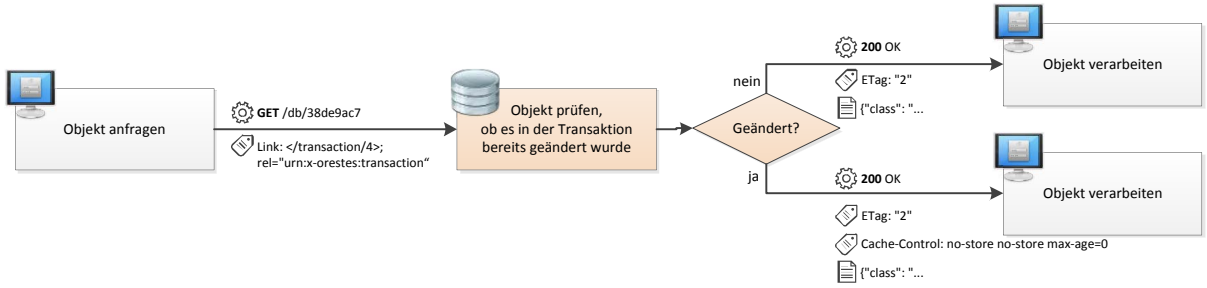


Abbildung 49 Objekt innerhalb einer Transaktion abfragen

In Abbildung 49 ist zu sehen, wie ein Client innerhalb einer Transaktion Objekte abrufen kann. Dabei muss berücksichtigt werden, dass auch Objekte, die in der Transaktion bereits geändert wurden, abgerufen werden können. Da die Versionen von geänderten Objekten aber erst zum Abschluss einer Transaktion geändert werden, tragen die geänderten Objekte immer noch den gleichen ETag wie vor der Änderung. Weil die Transaktion aber noch nicht abgeschlossen ist, dürfen andere Clients die Änderungen aus der Transaktion nicht sehen. Also muss das Objekt an den Client so ausgeliefert werden, dass die Web-Caches die transaktionalen Änderungen eines Objektes nicht cachen. Hierfür wird das Objekt mit einem `Cache-Control: no-store no-store max-age=0` Header versehen, der dafür sorgt, dass Caches den Response nicht speichern. Wie in Abbildung 49 zu sehen ist, prüft der Server, ob das Objekt bereits in der Transaktion geändert wurde und liefert das Objekt ggf. mit dem entsprechenden `Cache-Control`-Header aus.

Das Ändern und Löschen von Objekten läuft nach dem gleichen Schema ab, wie eine Änderung, die außerhalb einer Transaktion durchgeführt wird. Auch hier fügt der Client dem PUT- oder POST Request einen `ETag`-Header mit der Version hinzu und zusätzlich den `Link`-Header mit der Transaktions-ID. Jedoch vereinfacht sich die Verarbeitung auf Serverseite zunächst, denn der Server überprüft zunächst nicht, ob das geänderte Objekt noch aktuell ist, denn dies wird erst zum Abschluss der Transaktion durchgeführt. Wurde zuvor jedoch bereits innerhalb der aktuellen Transaktion eine Änderung auf dem Objekt durchge-

führt, so muss diese auf einer mindestens so neuen Version basieren, wie die vorherige Änderung. Ist die Version jedoch älter, so Antwortet der Server wie bei einer normalen Änderung mit dem Statuscode `412 Precondition Failed`. Ansonsten merkt er sich lediglich die Änderungen und die Version des Objektes, auf dem die Änderungen basieren und akzeptiert mit einem `200 OK` Statuscode. Bei der Objekterstellung mit Hilfe eines POST-Request an die „/db“ Ressource wird jedoch weiterhin sofort eine OID für das Objekt erzeugt, damit dieses im weiteren Verlauf der Transaktion bereits für Referenzen eingesetzt werden kann.

3.6.3 Behandlung von Objekt-IDs

Wenn eine Transaktion sehr viele Objekte gleichzeitig erstellen möchte, muss sie für jedes einzelne Objekt, das sie anlegen möchte, einen POST-Request an die „/db“-Ressource ausführen, damit das neue Objekt eine OID erhält. Denn erst dann kann die Transaktion dieses Objekt in anderen Objekten referenzieren. Dies würde gerade bei großen Objektmengen, die sehr stark untereinander verknüpft sind, zu sehr großen Latenzzeiten führen, da alle Objekte zunächst angelegt und anschließend ihre Referenzen noch einmal aktualisiert werden müssten. Deswegen gibt es bei Orestes in Transaktionen eine weitere Möglichkeit, neue Objekte anzulegen. Dabei allokiert der Client zunächst eindeutige neue OIDs, mithilfe eines POST-Request an die „/transaction/{tid}/oids“-Ressource. Der Client erhält daraufhin eine Liste, aller für ihn eindeutig neu generierten Objekt-URIs, an die er nun die neuen Objekte per PUT-Request schicken und zugleich in anderen Objekten referenzieren kann. Nachdem der Client eine Transaktion abgeschlossen hat, verfallen die nicht verwendeten Objekt-IDs und können in der Zukunft vom Server erneut verwendet werden.

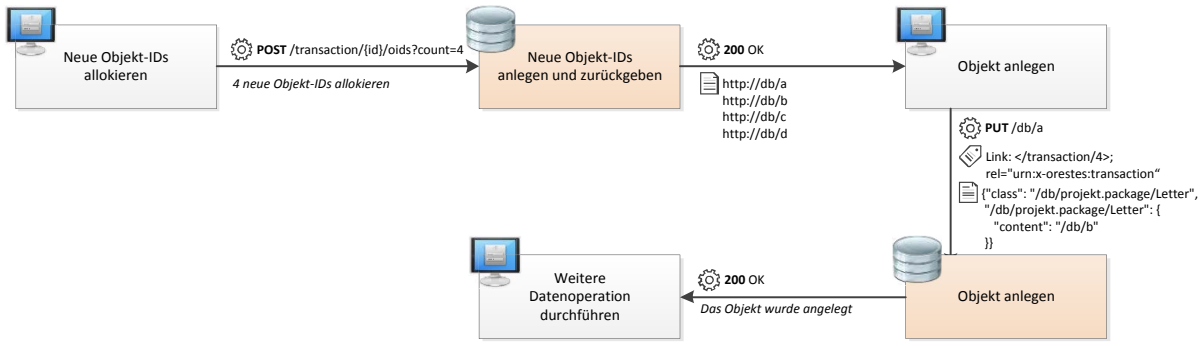


Abbildung 50 Allokieren von OIDs in einer Transaktion

In Abbildung 50 allokiert der Client nach dem Beginn einer Transaktion neue OIDs. Im Request gibt der Client mit dem „count“ Query-Parameter zusätzlich an, wie viele neue OIDs der Client erhalten möchte, hier also *4*. Wird dieser Parameter weggelassen, erhält er eine serverspezifische Anzahl an neuen OIDs. Im Response erhält der Client nun eine URI-Liste mit Objekt-URIs, die die neu angelegten Objektressourcen angeben. Der Client kann nun das neue Objekt mit Hilfe eines PUT-Request an die „/db/a“-Ressource anlegen und zugleich in dem Objekt auf ein anderes neues Objekt mit der OID *b* referenzieren. Der Request wird hier aber mit einem Statuscode `200 OK` beantwortet, da die Objektressource schon erstellt wurde

(Im Gegensatz zum normalen Erstellvorgang, bei dem der Statuscode `201 Created` zurückgegeben wird).

3.6.4 Ende einer Transaktion

Eine Transaktion kann erfolgreich abgeschlossen (*commited*) oder abgebrochen (*aborted*) werden. Genau diese beiden Ressourcen sind als Unterressourcen unter einer Transaktionsressource vorgesehen. Das Ende einer Transaktion wird mit Hilfe eines PUT-Requests an die jeweilige Ressource, „/transaction/{tid}/commited“ oder „/transaction/{tid}/aborted“ eingeleitet. Nachdem eine Transaktion abgeschlossen ist, können in dieser Transaktion keine weiteren Datenoperationen an der Datenbank ausgeführt werden. Jede Datenoperation, die dennoch außerhalb einer aktiven Transaktion ausgeführt wird, wird von dem Orestes-Server mit einem `409 Conflict` Fehlerstatuscode im Response beantwortet.

Wird eine Transaktion abgebrochen, so werden keinerlei Änderungen, die in der Transaktion an den Daten vorgenommen wurden in den Datenbestand übernommen. In dem PUT-Request, den der Client an die Datenbank sendet, werden keinerlei Daten im Message-Body übertragen. Der Server antwortet mit dem Statuscode `201 No Content`, wenn die Transaktion nicht bereits abgeschlossen wurde. Ein Abbruch einer Transaktion bedeutet also das gleiche, als ob der Datenbestand niemals geändert wurde.

Der erfolgreiche Abschluss einer Transaktion ist ein wenig komplizierter, da der Server nicht genau nachvollziehen kann, welche Objekte in der Transaktion von dem Client verwendet wurden (*Read-Set*). Dies liegt daran, dass Leseoperationen von Web-Caches beantwortet werden können, ohne, dass der Server jemals etwas von der Anfrage mitbekommen hat. Der Client muss sich somit jedes gelesene Objekt mit der entsprechenden Version merken, da Objekte, die aus einem Web-Cache stammen, veraltet sein können. Deshalb kann die Aktualitätsprüfung auch nur auf Serverseite stattfinden, da der Client die aktuellen Versionen eines Objektes nicht unbedingt kennt. Objekte, die der Client ändert, müssen jedoch auf Clientseite nicht zusätzlich geloggt werden, da Schreibvorgänge immer vom Server mit einer angegebenen Version verarbeitet werden. Wenn der Client also eine Transaktion commiten möchte, kann er zusätzlich Objekte und die jeweils verwendete Version des Objektes beim Commit mit angeben. Hierfür sendet der Client, beim PUT-Request an die „commited“-Ressource, ein JSON-Objekt mit, das diese Informationen beinhaltet. In Formatbeispiel 6 ist ein solches JSON-Objekt aufgezeigt, es gleicht den Queryresult-Ressourcen. Es ist ein einzelnes Objekt, bei dem die Schlüssel die jeweiligen Objekt-URIs sind und dessen Wert die verwendete Version des Objektes repräsentiert. In diesem Formatbeispiel, hat der Client also die Objekte mit den OIDs *3*, *5*, *67* im Verlauf der Transaktion gelesen und die jeweiligen Versionen 4, 20 und 872

```
1 {
2     "/db/3"   : "4",
3     "/db/5"   : "20",
4     "/db/67"  : "872"
5 }
```

Formatbeispiel 6 Objektrefernzen mit Versionsnummern

dieser Objekte erhalten. Der Server kann nun ermitteln, ob die Transaktion erfolgreich abgeschlossen werden kann, denn dafür müssen alle geschriebenen und angegebenen Objekte mit den jeweiligen aktuellen Versionen übereinstimmen. Wird bei der Überprüfung auf Serverseite festgestellt, dass eines oder mehrere Objekte veraltet sind, wird die Transaktion abgebrochen. Dadurch, dass der Client dem Server explizit mitteilt, welche Objekte geprüft werden sollen, kann er auch Objekte aus einer vorherigen Transaktion mitverwenden, ohne diese in der neuen Transaktion erneut zu laden. Dennoch kann er zum Abschluss der Transaktion sicherstellen, dass diese nur erfolgreich übernommen wird, wenn die mitverwendeten Objekte ebenfalls noch aktuell sind.

Wird die Transaktion erfolgreich abgeschlossen, werden die geänderten Objekte in den Datenbestand übernommen und für diese neue Versionen generiert. Damit der Client weiß, welche Versionen die geänderten Objekte nun haben, erhält der Client im Response des PUT-Request wiederum ein JSON-Objekt. Dieses hat den gleichen Aufbau wie das Formatbeispiel 6, mit dem Unterschied, dass es diesmal alle Objekt-URIs der geänderten Objekte als Schlüssel und die jeweilige neue Version als Wert beinhaltet.

Wird die Transaktion jedoch abgebrochen weil ein Objekt veraltet war, antwortet der Server hier mit einem `412 Precondition Failed` Fehlerstatuscode. Auch hier schickt der Server im Response-Body ein JSON-Objekt, das dem aus Formatbeispiel 6 gleicht, zurück. In diesem Fall zeigt das JSON-Objekt jedoch genau die Objekte auf, die in der Transaktion veraltet waren. Die Version der jeweiligen Objekt-URIs entsprechen der aktuellen Version des Objektes in der Datenbank, damit der Client, falls er die Transaktion beispielsweise neu startet, genau diese Objekte von den Caches revalidieren lassen kann.

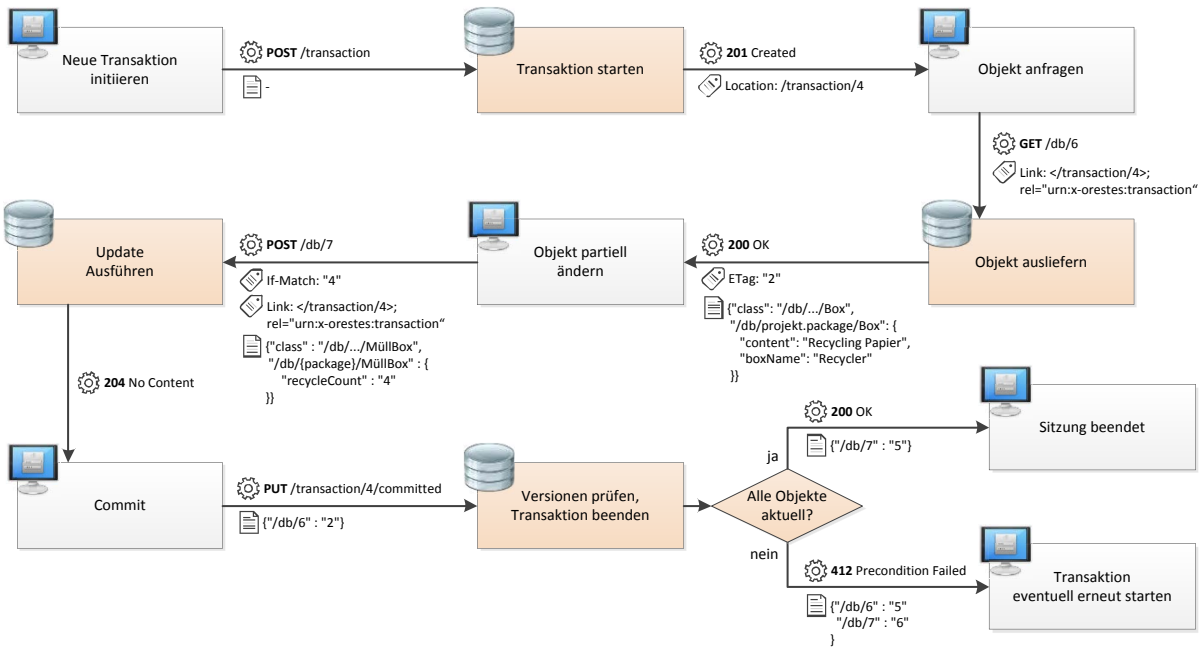


Abbildung 51 Beispielhafter Ablauf einer Transaktion

Dieses Kapitel wollen wir mit einem etwas umfangreicheren Beispiel eines Transaktionsablaufs abschließen (Abbildung 51). Zunächst startet der Client eine neue Transaktion mit einem POST-Request an die „/transaction“-Ressource. Wie zuvor beschrieben, erhält der Client im Response die spezifische Transaktionsressource, mit deren Hilfe er die gestartete Transaktion verwalten kann. Der Client fragt nun zunächst das Objekt mit der Objekt-ID *6* ab und erhält dieses Objekt in der Version *2*. Dabei ist zu beachten, dass in diesem Fall auch ein Web-Cache den Request beantwortet haben könnte, also das Objekt eventuell nicht mehr aktuell ist. Im weiteren Verlauf ändert der Client das Objekt mit der Objekt-ID *7* partiell. Die Änderungen basieren dabei auf der Version *4* des Objektes. Abschließend commitet der Client die Transaktion mit einem PUT-Request an die „/transaction/4/commited“-Ressource. Hierbei sendet der Client im Request die Version des zuvor gelesenen Objektes mit, um kenntlich zu machen, dass dieses Objekt Teil der Transaktion ist. Der Server überprüft nun die Versionen des gelesenen Objektes mit der OID *6* und des geschriebenen Objektes mit der OID *7* auf Aktualität.

- Sind die beiden Objekte immer noch aktuell, so übernimmt er die Änderung des Objektes mit der OID *7* in den Datenbestand und antwortet mit dem Statuscode `200 OK`. Im Response-Body schickt der Server zusätzlich die aktuelle Version des geänderten Objektes mit, die in diesem Beispiel von *4* auf *5* erhöht wurde.
- Ist eines der Objekte jedoch nicht mehr aktuell, so antwortet der Server mit einem `412 Precondition Failed` Statuscode und schickt im Response-Body die aktuellen Versionen der nicht mehr aktuellen Objekte. In diesem Beispiel hat das gelesene Objekt bereits die Version *5* und das geänderte Objekt die Version *6*.

3.7 Verwaltung

In Orestes gibt es drei Einstellungsressourcen, um den Orestes-Server clientseitig zu konfigurieren. Diese können per GET-Request abgefragt werden, mit PUT komplett und mit POST partiell geändert werden. Es gibt die „/settings“ Ressource, mit der allgemeine Einstellungen für den Orestes-Server abgefragt und geändert werden können. Vor allem serverspezifische Einstellungen der zugrundeliegenden Datenbank können hier konfiguriert werden. Außerdem gibt es die „/transaction/settings“-Ressource, an der transaktionsspezifische und die „/query/settings“-Ressource, an der queryspezifische Einstellungen gesetzt werden können. Alle drei Ressourcen werden durch ein einfaches JSON-Objekt repräsentiert, dass als Schlüssel den jeweiligen Einstellungsnamen und als Wert den jeweiligen Wert der Einstellung beinhaltet.

Die Einstellungsobjekte stellen zu diesem Zeitpunkt wenige orestesspezifische Einstellungsmöglichkeiten zur Verfügung, da diese voraussichtlich erst im weiteren Entwicklungsprozess des Orestes-Systems entstehen werden. Ein Beispiel für eine bereits existierende Einstellungsmöglichkeit ist die „lifetime“-Einstellung, die für die Transaktionen angibt, wie lange diese insgesamt laufen dürfen und für Queryressourcen, wie lange diese auf dem Server maximal verbleiben bevor sie gelöscht werden.

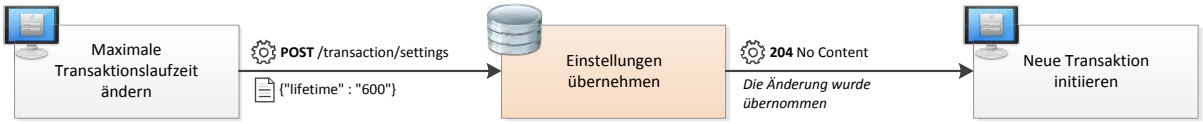


Abbildung 52 Ein Client der die maximale Transaktionslaufzeit auf 600 Sekunden einstellt

Der Client aus Abbildung 52 ändert die Transaktionslaufzeit auf 10 Minuten, indem er ein POST-Request an die „/transaction/settings“ Ressource sendet. Er verwendet ein POST-Request, da er ansonsten zunächst alle Einstellungen abrufen müsste, bevor er mit einem PUT-Request diese Einstellung ändern könnte. Der Server antwortet in jedem Fall mit dem Statuscode `204 No Content`, um zu signalisieren, dass die Einstellungen erfolgreich übernommen wurden.

Des Weiteren gibt es die „/info“-Ressource, mit deren Hilfe der Client datenbankspezifische Informationen per GET-Request abrufen kann. Die Repräsentation dieser Ressource ist dabei komplett der datenbankspezifischen Implementierung überlassen. Das bedeutet, dass ein Client den Inhalt dieser Ressource nur verstehen kann, wenn er eine datenbankspezifische Implementierung zur Interpretation des Inhaltes besitzt.

3.8 API- und Formatumstellungen

Damit Orestes auch in Zukunft an Funktionalität erweitert oder das Übertragungsformat geändert werden kann, ist Orestes jetzt schon so konzipiert, dass dies sehr leicht möglich ist.

Es gibt in Orestes die „/version“-Ressource, die die Versionsnummer des Orestes-Servers angibt. Mit ihr kann ein Client, vor dem eigentlichen Kommunikationsprozess mit dem Server, feststellen, ob er zu dem Orestes-Server kompatibel ist. Damit das Auslesen der Versionsnummer stets funktioniert, besitzt diese Ressource eine textuelle Repräsentation (`text/plain`) der Version des Servers. Der Client kann nun mit einem GET-Request diese Ressource abrufen und testen, ob er mit der Version des Servers kompatibel ist.

Die Formatumstellung ist in Orestes über Content Negotiation gelöst. Das bedeutet, dass wenn der Client eine Ressource anfragt, mithilfe des `Accept`-Headers angibt welche Formate er alle unterstützt und welche er davon favorisiert. Der Server kann nun eines dieser Formate auswählen, das er ebenfalls unterstützt und in diesem Format antworten. Der Client erkennt dann anhand des `Content-Type`-Headers im Response das Format, das der Server ausgewählt hat. Dies bedeutet also, dass eine Formatumstellung als Formaterweiterung vorgenommen werden kann, wodurch die alten Formate nicht ersetzt werden. Somit ist es sehr wahrscheinlich, dass Server und Client mindestens ein gemeinsames Format haben, das beide Seiten verstehen.

3.9 Zusammenfassung: Ressourcen und Operationen

Die in Tabelle 12 als Operationen und Ressourcen zusammengefasste REST-Architektur von Orestes erfüllt alle angestrebten REST-Prinzipien (siehe Kapitel „Zusammenfassung als REST-Prinzipien“):

Eindeutig identifizierbare Ressourcen

- Die Instanzen der Abstraktionen von Orestes sind ausnahmslos als adressierbare, eindeutig identifizierbare, hierarchisch organisierte Ressourcen modelliert: Objekte, Klassen, Schemata, Namensräume, Querys, Transaktionen, OID-Listen, Zustände, Einstellungen und Verwaltungsobjekte.

Verknüpfung/Hypermedia

- Orestes macht von Hypermedia auf verschiedene Weisen gebrauch:
  - HTTP-Nachrichten und Repräsentationen enthalten Verweise auf Ressourcen (z.B. Objekte die als Feld eine Referenz auf ein anderes Objekt halten)
  - Oberressourcen verweisen auf alle verfügbaren Unterresourcen → explorative Schnittstellenerschließung und -Verwendung

Standardmethoden

- Die Operationen die an Orestes-Ressourcen durchführbar sind, werden gemäß ihrer jeweiligen Semantik auf standardisierte HTTP-Methoden abgebildet. Eine Migration zu neuen Methoden (z.B. PATCH oder BATCH) ist jederzeit möglich.

Unterschiedliche Repräsentationen

- Resourcen und Repräsenationen sind in Orestes streng entkoppelt. Ein Objekt oder Schema kann beispielsweise jederzeit parallel als JSON-, XML- und HTML-Repräsentation verfügbar sein und dynamisch durch HTTP-Content-Negotiation ausgehandelt werden.

Statuslose Kommunikation

- Serverseitiger Zustand wird entweder vom Client gehalten (verwendete Transaktion und Autorisierung) oder in den Zustand einer Ressource verwandelt (ausgewertete Querys, laufende Transaktionen, allokierte OIDs, Transaktionsstatus). Cookies und Sessions werden aus diesem Grund nicht verwendet.

Caching

- Das Caching ist tief in Orestes verankert: Objekte werden durch Web-Caches für lange Zeitspannen gespeichert, abhängig von ihrem Zustand (z.B. Gegenstand laufender Transaktion) und änderungsinduzierten Invalidierungen. Auch die übrigen Ressourcen sind durch Hashwerte und Revalidierungsaufforderungen cachbar.

Abbildung 53 Rest-Prinzipien in der Orestes-Architektur

Die Konzeption von Orestes ist angelehnt an die Umsetzung der in Kapitel 2.5 beschriebenen NoSQL-Datenbank-Schnittstellen. Orestes führt jedoch ein auf dem Gebiet der webzentrierten Persistenztechniken neues Element ein: die umfassende Nutzung der Caching-Hierarchie zur Speicherung von Datenelementen. Die Nutzung der Web-Caches hat jedoch Konsequenzen auf das Maß an Konsistenz, die durch das System aus Sicht eines Benutzers sichergestellt wird.

Anhand der Konsistenzbedingungen aus Kapitel 2.5 kann Orestes wie folgt klassifiziert werden:

- **Monotonic Read:** ein Leseoperation über HTTP kann im Regelfall keine älteren Objekte zurückgeben als ein früherer, da Einträge in Web-Caches nicht durch ältere ersetzt werden (sie können nur expirieren oder invalidiert werden). Werden allerdings aufgrund von Routingentscheidungen andere Web-Caches ältere Objekte als zuvor empfangene ausliefern, so kann dies anhand eines Vergleichs mit protokollierten, gelesenen Versionen ermittelt werden. Anschließend wird mit einer Revalidierungsaufforderung das aktuelle Objekt angefragt.
- **Read Your Writes:** garantiert Orestes dadurch, dass einerseits Schreibzugriffe vorhandene Web-Cache-Einträge invalidieren. Des Weiteren wird durch das Protokollieren geschriebener Versionen das Laden älterer Versionen erkannt und durch eine Revalidierung behoben.
- **Write follows Reads:** Orestes garantiert, dass ein Schreibzugriff, der auf einen Lesezugriff folgt, auf einer mindestens so aktuellen Version des Objekts ausgeführt wird wie ein vorangehender Lesezugriff. Diese Schreiboperation erfolgt entweder lokal, wo die *Monotonic Read*-Bedingung eine Verletzung von Write follow Reads verhindert oder auf dem Server, der stets die aktuellste Version besitzt.
- Betrachtet man die Web-Caches als Teil des (Datenbank-)Systems, so wird von Orestes keine **Immediate** oder **Strong Consistency** gewährt, da Änderungen nicht sofort in Web-Caches sichtbar werden. Die Verwendung von Transaktionen mit einer optimistischen Commitbehandlung kann bei Orestes dennoch die ACID Kriterien garantieren. Diese werden jedoch nicht durch Orestes zugesichert oder beeinflusst, sondern durch die serverseitige Implementierung und das OODBMS:
    - **Atomicity**: Die Änderungen einer Transaktion werden durch das OODBMS nur als Ganzes angenommen oder zurückgewiesen.
    - **Consistency**: Die Datenbasis (nicht das Gesamtsystem) befindet sich nach einer Transaktion in einem konsistenten Zustand (so wird z.B. referentielle Integrität sichergestellt, nicht aber der gleiche Zustand aller Web-Caches).
    - **Isolation**: Die Wahl und Implementierung eines *Isolation Levels* obliegt ausschließlich dem OODBMS. Web-Caches haben keinen Einfluss darauf:
        - **Dirty Reads**: Objekte die Gegenstand laufender Transaktionen sind, werden bei Orestes nicht in Web-Caches gespeichert.
        - **Lost Updates**: werden unabhängig von Caches bei der Commitbehandlung identifiziert und durch einen Transaktionsrollback behoben.
        - **Non-Repeatable** und **Phantom Reads**: Die Web-Caches haben keinen Einfluss darauf, wie isoliert Transaktionen und Querys in einem OODBMS ausgeführt werden.
    - **Durability**: Für die Dauerhaftigkeit einer Änderung garantiert allein das OODBMS, gespeicherte Einträge von Web-Caches haben keinen Anspruch auf Dauerhaftigkeit.

Die Ressourcen und Operationen von Orestes sind in Tabelle 12 zusammengefasst. Für jede Ressource sind die verwendbaren HTTP-Methoden, Repräsentationsformate bei Orion (Referenzimplementierung), Statuscodes und die Bedeutung des Aufrufs aufgeführt.

Verb	Format	Statuscode	Bedeutung
/			
GET	←URI-List	*Multiple Choices*	Alle Subressourcen abrufen
/db			
POST	→JSON ←URI-List	*OK*	Ein Objekt mit servergenerierter OID anlegen
/db/{oid}			
GET	←JSON	*OK*	Objekt abrufen
PUT	→JSON	*Created*	Objekt aktualisieren oder anlegen
POST	→JSON	*No Content*	Objekt partiell aktualisieren
DELETE		*No Content*	Objekt löschen
/db/{namespace}/{class}			
GET	←URI-List	*OK*	Alle Subressourcen ausgeben
PUT		*Created*	Neue Klasse anlegen
DELETE		*No Content*	Klasse löschen, falls kein Schema definiert ist und keine referentiellen Abhängigkeiten bestehen
/db/{namespace}/all_classes			
GET	←URI-List	*OK*	Alle Klassen eines Namespaces abrufen
/db/{namespace}/{class}/all_objects			
GET	←URI-List	*OK*	Alle Objekte einer Klasse abrufen
/db/{namespace}			
GET	←URI-List	*OK*	Alle Subressourcen (Klassen) abrufen
/db/all_namespaces			
GET	←URI-List	*OK*	Alle definierten Namespaces abrufen
/db/all_classes			
GET	←URI-List	*OK*	Alle definierten Klassen und ihre zugehörigen Namespaces abrufen
/db/all_objects			
GET	←URI-List	*OK*	Alle Objekte der Datenbank abrufen
/transaction			
POST		*Created*	Transaktion starten
/transaction/settings			
GET	←JSON	*OK*	Transaktionseinstellungen abrufen
PUT	→JSON	*No Content*	Transaktionseinstellungen setzen
POST	→JSON	*No Content*	Transaktionseinstellungen partiell setzen
/transaction/{id}/			
GET	←URI-List	*Multiple Choices*	Alle Transaktions-Subressourcen ausgeben
/transaction/{id}/status			
GET	←JSON	*OK*	Status der Transaktion abfragen
/transaction/{id}/changeset			
GET	←URI-List	*OK*	Alle OIDs geänderter Objekte ausgeben

Verb	Format	Statuscode	Bedeutung
/transaction/{id}/commited			
PUT	→JSON ←JSON	*OK*	Alle gelesenen und geschriebenen Objekte als OID für Commit der laufenden Transaktion übergeben
/transaction/{id}/aborted			
PUT		*No Content*	Rollback der laufenden Transaktion
/transaction/{id}/oids			
POST	←URI-List	*OK*	Neue OIDs allokieren und zurückgeben
GET	←URI-List	*OK*	Alle generierten OIDs abfragen
/query			
POST	→JSON ←URI-List	*OK* / *Accepted*	Eine benannte, wiederverwendbare Abfrage absetzen
/query/ephemeral			
POST	→JSON ←URI-List	*OK* / *Accepted*	Eine einmalige Abfrage absetzen
/query/settings			
GET	←JSON	*OK*	Query-Einstellungen lesen
POST	→JSON	*No Content*	Query-Einstellungen partiell ändern
PUT	→JSON	*No Content*	Query-Einstellungen setzen
/query/{id}			
GET	←URI-List	*OK*	Ergebnis einer zuvor abgesetzten Query abrufen
PUT		*No Content*	Eine abgesetzten Query erneut ausführen
/settings			
GET	←JSON	*OK*	Globale Servereinstellungen lesen
POST	→JSON	*No Content*	Globale Servereinstellungen partiell ändern
PUT	→JSON	*No Content*	Globale Servereinstellungen setzen
/version			
GET	←JSON	*OK*	Versionsnummer des OODBMS
/status			
GET	←JSON	*OK*	Laufzeitinformationen über OODBMS
/info			
GET	←JSON	*OK*	Serverinformationen abrufen (Name, Hersteller, etc.)

Tabelle 12 Ressourcen und Operationen von Orestes

4 Implementierung

In dem folgenden Kapitel werden wir unsere Implementierung der Orestes-Übertragungsschicht (Orion, „Orestes-Implementation On Networklevel“) vorstellen und ausführen. Die Architektur der Implementierung mit ihren Komponenten ist in Abbildung 52 gezeigt:

- **Orestes-Schnittstelle:** Die auf Client und Server identische Schnittstelle bildet die Operationen von OODBMS auf Netzwerkebene ab.
- **Orestes-Client:** Nimmt die Aufrufe aus einer Client-Persistenz-API an der Orestes-Schnittstelle entgegen und führt die notwendigen Schritte aus, um eine Anfrage über HTTP zu versenden.
- **Orestes-Server:** Empfängt die HTTP-Anfragen, verarbeitet sie und ruft über die Orestes-Schnittstelle die Serverimplementierung der jeweiligen Operation auf.

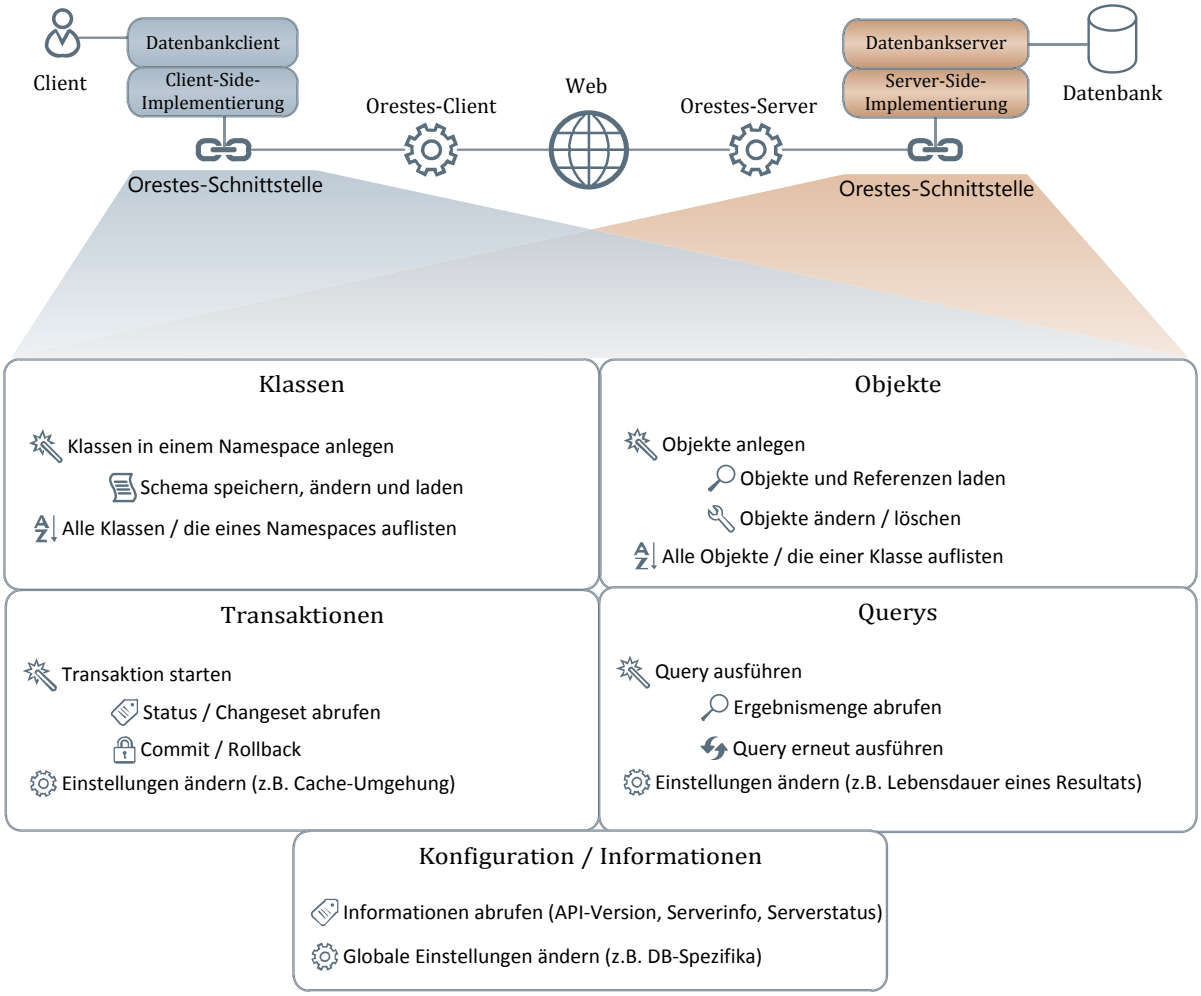


Abbildung 54 Die Orestes-Schnittstelle aus High-Level-Sicht

Die Client-Side und Server-Side-Implementierung für die Aufrufe und Entgegennahme der Orestes-Schnittstellen-Operationen sind nicht Teil dieser Arbeit. Wir haben jedoch die Orestes-Schnittstelle mit der Kenntnis einer Einbettungsmöglichkeit in die VERSANT OBJECT

DATABASE entwickelt. Eine Implementierung dieser beiden Komponenten beschränkt sich bei Verfügbarkeit von Orion auf die Generierung von Schnittstellen-Aufrufen und entsprechende Weiterverarbeitung dieser Aufrufe auf Serverseite. So entsteht bei einem ansonsten unveränderten System eine Zwischenschicht mit speziellen (orestesspezifischen) Datenstrukturen, die jedoch die Leistungsparameter des Systems beeinflusst.

Orion haben wir mit Java umgesetzt, so dass ein Zugriff auf die Orestes-Schnittstelle von Orion aus jeder JVM-konformen Sprache erfolgen kann.

4.1 Die Orestes-Schnittstelle von Orion

Wie in Abbildung 54 zu sehen ist, besitzen der Orestes-Client und der Orestes-Server eine identische Orestes-Schnittstelle. Der Orestes-Client implementiert diese Schnittstelle, die von dem spezifischen Client genutzt werden kann, um über HTTP Operationen an den Orestes-Server zu senden. Auf Serverseite muss die datenbankspezifische Anbindung diese Schnittstelle implementieren, die dann von dem Orestes-Server verwendet wird, um die HTTP Aufrufe an diese weiterzugeben. Dieses Interface spiegelt somit die HTTP-Schnittstelle von Orestes in Java wieder.

Die Orestes-Schnittstelle verwendet dabei verschiedenen Datenstrukturen, die zwischen Client und Server übertragen werden. Diese werden durch Java-Interfaces abstrahiert, damit die client- oder serverspezifische Implementierung eigene Repräsentationen dieser Datenstrukturen verwenden kann. In Tabelle 13 werden diese Datenstrukturen aufgelistet und ihre Funktionalität innerhalb von Orion erklärt. Für fast alle Datenstrukturen gibt es bereits eine Standardimplementierung, außer für jene, die datenbankspezifische Daten bereitstellen.

Interface	Erklärung	Abbildung auf HTTP
IVersion	Repräsentiert eine Version eines Objektes innerhalb von Orion. Dieses Interface muss von jeder Datenbank implementiert werden, da die Versionsnummer von Objekten in verschiedenen Datenbanksystemen unterschiedlich behandelt werden.	Wird in dem `If-Match`- und `If-None-Match`-Header des Request, sowie im `ETag`-Header des Response als String-Repräsentation übertragen.
IType	Repräsentiert die verschiedenen Typen und Klassen, die in Orestes verwaltet werden können, einschließlich nativer Typen.	Wird durch eine Typ-URI repräsentiert und im Request meist als Teil der Request-URI übertragen.
ISchema	Ein `ISchema` wird für die Schemadeklaration einer Klasse übertragen.	Wird im Body in verschiedenen Medientypformaten übertragen. Das beste Übertragungsformat wird mit Hilfe von Content Negotiation ermittelt.
IObject	Beinhaltet die Werte der Objektfelder eines Objektes. Ein `IObject` hat immer eine OID und eine `IVersion`, die das Objekt identifizieren. Auch Sammlungen sind vom Typ `IObject`.	Wird im Content in verschiedenen Medientypformaten übertragen. Das beste Übertragungsformat wird mit Hilfe von Content Negotiation ermittelt.
IQuery	Mithilfe dieses Interfaces werden Querys in Orestes identifiziert. Die datenbankspezifische Implementierung kann dieses Interface mehrmals implementieren, um verschiedene Querytypen über HTTP an die Datenbank zu schicken.	Wird im Content eines Query-Requests an den Server geschickt. Mit Hilfe des Medientyps im `Content-Type`-Header wird der Querytyp auf Serverseite identifiziert und die zugehörige `IQuery`-Implementierung instanziiert.
OIDs	`OIDs` sind eine Liste von Objekt-IDs und werden in Request und Response verwendet, um Objektmengen abzubilden, z.B. als Response auf ein „/db/all_objects“-Request.	Wird im Content in verschiedenen Medientypformaten übertragen. Das beste Übertragungsformat wird mit Hilfe von Content Negotiation ermittelt.
OIDVersionTuples	Dies ist eine Erweiterung der `OID`-Liste, in der jeder OID eines Objektes zusätzlich noch dessen Version zugeordnet ist. Die Datenstruktur wird durch eine Map gebildet.	Wird im Content in verschiedenen Medientypformaten übertragen. Das beste Übertragungsformat wird mit Hilfe von Content Negotiation ermittelt.

Tabelle 13 Datenstrukturen in Orion und ihre Funktionalität

Die Orestes-Schnittstelle ist in Orion in vier Interfaces aufgeteilt. Es gibt das `IDataOperation` Interface, dass alle Objektoperationen bereitstellt, die außerhalb und innerhalb von Transaktionen ausgeführt werden können. Dieses Interface ist ein Superinterface und stellt

nur die Basisoperationen der zwei anderen Interfaces dar. Von diesem erbt einmal das `ITransaction` Interface, das die Schnittstelle für den transaktionalen Zugriff bereitstellt. Außerdem gibt es noch das `IOrestes` Interface, das ebenfalls von dem `IDataOperation` Interface erbt, hier aber für den nicht transaktionalen Zugriff auf Objekteoperationen zuständig ist (Autocommit-Modus). Zudem enthält es noch alle Operationen für die Metadatenverwaltung, wie das Abrufen und Ändern von Klassen- und Schemata, Einstellungen und Serverinformationen. Das vierte Interface `IReExecuteableQuery` kapselt den Zugriff auf Queryressourcen, mit dessen Hilfe Querys öfter ausgeführt, das Queryresult auf Serverseite gespeichert und nur Teile des Queryresults abgerufen werden können. In Tabelle 14 sind alle Methoden und die korrespondierenden HTTP-Schnittstellen aufgezeigt, die diese Interfaces bereitstellen,.

Methode des Java-Interfaces	Methode der HTTP-Schnittstelle
IDataOperation	
load(String, IVersion) : IObject	**GET** /db/{oid}
Das erste Argument ist die OID des zu ladenden Objektes. Wenn der zweite Parameter angegeben ist, so wird das Objekt nur geladen, wenn es die angegebene Version hat. So kann z.B., wenn ein Objekt aus einem Queryresult geladen wird, sichergestellt werden, dass es noch die Version hat, die beim Ausführen des Querys verwendet wurde.	
loadNewest(String, IVersion) : IObject	**GET** /db/{oid}
Lädt das angegebene Objekt erneut, indem ein Revalidierungs-Request durchgeführt wird. Ist das Objekt veraltet, so wird das neue Objekt zurückgegeben, ansonsten `null`*.*	
store(IObject, IVersion) : void	**POST** /db/{oid} **PUT** /db/{oid}
Speichert das angegebene Objekt in der Datenbank. Hat das Objekt noch keine OID, wird dieses per POST-Request an den Server gesendet und nach seiner Erstellung wird diesem seine OID zugewiesen. Wenn es sich um eine Änderung handelt, so wird das Objekt, wenn es sich um eine partielle Änderung handelt, per POST-Request und ansonsten per PUT-Request gesendet. Ist der zweite Parameter dabei angegeben, so muss das Objekt auf Serverseite die angegebene Version haben, damit die Änderung übernommen wird. Nachdem das Objekt geändert wurde, wird die Version des übergebenen Objektes aktualisiert.	
delete(String, IVersion) : void	**DELETE** /db/{oid}
Der erste Parameter gibt die OID des Objektes an, das gelöscht werden soll. Ist der zweite Parameter angegeben, so wird das Objekt nur gelöscht, wenn es auf Serverseite noch die angegebene Version hat.	
getAllObjects([IType]) : OIDs	**GET** /db/all_objects **GET** /db/{namespace}/{class}/all_objects
Gibt die OIDs aller Objekte der Datenbank zurück, wenn die Methode ohne den IType Parameter aufgerufen wird, ansonsten alle OIDs der Objekte des angegebenen Typs.	
executeOnce(IQuery) : OIDVersionTuples	**POST** /query/ephemeral
Führt den angegebenen Query aus, und gibt das Ergebnis als Map zurück. Dabei sind die Schlüssel dieser Map OIDs und dessen Werte die Versionen der gefundenen Objekte.	
execute(IQuery) : IReExecuteableQuery	**POST** /query
Führt den Query auf dem Server aus und gibt ein IReExecuteableQuery zurück, an dem das Ergebnis des Querys abgerufen, aber auch der ursprüngliche Query erneut ausgeführt werden kann.	

IOrestes : IDataOperation	
getAllNamespaces() : ResourceList	**GET** /db/all_namespaces
Gibt die URIs aller bekannten Namespaces der Datenbank als Liste zurück.	
getAllClasses([String]) : Types	**GET** /db/all_classes **GET** /db/{namespace}/all_classes
Gibt die Typen aller benutzerdefinierten Klassen, die die Datenbank kennt zurück, wenn der Parameter zusätzlich einen Namespace angibt, werden nur die Klassen aus diesem Namespace zurückgegeben.	
createClass(IType) : void	**PUT** /db/{namespace}/{class}
Legt die angegebene Klasse an, wenn diese nicht schon existiert.	
deleteClass(IType) : void	**DELETE** /db/{namespace}/{class}
Löscht die angegebene Klasse, wenn alle Vorrausetzungen erfüllt sind.	
exists(IType) : boolean	**HEAD** /db/{namespace}/{class}
Testet, ob die angegebene Klasse in der Datenbank existiert.	
load(IType) : ISchema	**GET** /db/{namespace}/{class}/schema
Lädt das Schema für die angegebene Klasse und gibt es zurück.	
store(ISchema) : void	**PUT** /db/{namespace}/{class}/schema
Speichert das Klassenschema, wenn alle Vorrausetzungen erfüllt sind.	
delete(IType) : void	**DELETE** /db/{namespace}/{class}/schema
Löscht das Schema der angegebenen Klasse, wenn alle Voraussetzungen erfüllt sind.	
beginTransaction() : ITransaction	**POST** /transaction
Startet eine neue Transaktion und gibt diese zurück.	
getGeneralSettings() : Properties getTransactionSettings() : Properties getQuerySettings() : Properties	**GET** /settings **GET** /transaction/settings **GET** /query/settings
Gibt die jeweiligen Einstellungen als Property-Map zurück.	
setGeneralSettings(Properties) : void setTransactionSettings(Properties) : void setQuerySettings(Properties) : void	**PUT** /settings **PUT** /transaction/settings **PUT** /query/settings
Ändert alle Einstellungen auf die neuen angegebenen Werte aus der Property-Map	
updateGeneralSettings(Properties) : void updateTransactionSettings(Properties) : void updateQuerySettings(Properties) : void	**POST** /settings **POST** /transaction/settings **POST** /query/settings
Ändert gewählte Einstellungen auf die neuen angegebenen Werte aus der Property-Map	
getServerInfo() : IServerInfo	**GET** /info
Gibt das serverspezifische Informationsobjekt zurück, für das der Client eine Implementierung benötig.	
getStatus() : IStatus	**GET** /status
Gibt das serverspezifische Statusobjekt, für das der Client eine Implementierung benötigt, zurück.	
getOrestesVersion() : String	**GET** /version
Gibt die Version des Orestesservers zurück.	

ITransaction : IDataOperation	
isActive() : boolean isAborted() : boolean isRollbacked() : boolean	**HEAD** /transaction/{tid}
Testet den Status der Transaktion. Eine Transaktion ist solange aktiv, bis sie „commitet“ oder „abortet“ wird. Der Status der Transaktion wird außerdem auf „aborted“ gesetzt, sobald in der Transaktion ein Fehler auftrat. Sobald eine Transaktion abgeschlossen wurde, existiert seine Ressource nicht mehr, wodurch der Test auf Existenz der Ressource reicht, um den Aktivstatus zu verifizieren.	
allocateOIDs([int]) : OIDs	**POST** /transaction/{id}/oids
Allokiert neue OIDs und gibt diese als Liste zurück. Sie können anschließend zu erstellenden Objekten zugewiesen werden. Mit dem optionalen Parameter kann angegeben werden, wie viele neue OIDs generiert werden sollen.	
getAllocatedOIDs() : OIDs	**GET** /transaction/{id}/oids
Gibt alle in dieser Transaktion bereits allokierten OIDs als Liste zurück.	
getChangeSet() : OIDs	**GET** /transaction/{id}/changeset
Gibt alle OIDs der Objekte als Liste zurück, die in der Transaktion bereits geändert wurden.	
commit(OIDVersionTuples) : OIDVersionTuples	**PUT** /transaction/{id}/commited
Commited die Transaktion. Dabei können in der übergebenen Map weitere OIDs und die jeweils verwendete Version von weiteren Objekten, die in der Transaktion gelesen oder verwendet wurden, mit angegeben werden. In der zurückgegebenen Map sind die OIDs der geänderten Objekte und deren aktuelle Versionen enthalten.	
abort()	**PUT** /transaction/{id}/aborted
Bricht die Transaktion ab, alle Änderungen der Transaktion werden rückgängig gemacht.	
IReExecuteableQuery	
getObjects([form, to]) : OIDVersionTuples	**GET** /query/{id}
Ruft die Objekte des Queryresults ab. Wenn die beiden optionalen Parameter mit angegeben sind, kann ein bestimmter Teil des Queryresults selektiert und zurückgegeben werden.	
reExecute() : void	**PUT** /query/{id}
Führt den Query erneut aus und aktualisiert das Queryresult auf Serverseite.	

Tabelle 14 Das Orestes Interface in Java

Neben diesem Interface, das der Server und der Client gleichermaßen verwenden, gibt es auf Clientseite noch ein erweitertes Interface, das zusätzliche Methoden bereitstellt, die unter anderem die Commitbehandlung vereinfachen aber auch das Laden, Speichern, Löschen und Revalidieren von Objekten. Zudem gibt es die *all*-Methoden, die mehrere Objekte zugleich laden, speichern und löschen können, um z.B. alle Objekte eins Query-Results abzurufen. Diese Methoden arbeiten dabei asynchron mit mehreren TCP-Verbindungen zugleich, um weiterhin Web-Caching ausnutzen zu können.

4.2 Projektstruktur von Orion

Orestes ist in drei Projekte unterteilt, um eine klare Trennung zwischen dem Orestes-Client, dem Orestes-Server und den Komponenten die beide Teile benötigen zu schaffen. Somit benötigen beide Seiten jeweils die Klassen aus dem *orestes-common* Projekt, der Client die spezifischen Klassen aus dem *orestes-client* Projekt und der Server die spezifischen Klassen aus dem *orestes-server* Projekt.

Das *orestes-client* Projekt beinhaltet einen Restlet-Client, der das Orestes-Interface implementiert und somit alle Java-Methoden mit den entsprechenden HTTP-Methoden umsetzt. Zudem enthält es noch einige Restlet-spezifische Klassen, die für die Anbindung an das Restlet-Framework verwendet werden, um beispielsweise den zugrundeliegenden HTTP-Client zu konfigurieren. Dieses Projekt ist sehr kompakt und beinhaltet lediglich 7 Klassen.

Das *orestes-server* Projekt umfasst eine umfangreichere Anzahl von ca. 25 Klassen, da jede Orestes-Ressource durch eine `ServerRessource`-Klasse umgesetzt ist, die die jeweiligen HTTP-Methoden verarbeiten. Zusätzlich enthält es noch den eigentlichen Orestes-Server, der den internen Restlet-Server konfiguriert und an dem Restlet-Router die Orestes-Ressourcen registriert.

Das *orestes-common* Projekt beinhaltet zum einem, die Orestes-Interfaces, die einmal vom spezifischen Server und von dem Orestes-Client implementiert werden. Zudem enthält es auch noch die Implementierungen der Datenstrukturen, die Orestes verwendet und alle Konverter, die Restlet benötigt, um die verschiedenen Datenstrukturen zwischen HTTP-Entity und Java-Objekten konvertieren zu können. Insgesamt umfasst dieses Projekt mehr als 70 Klassen.

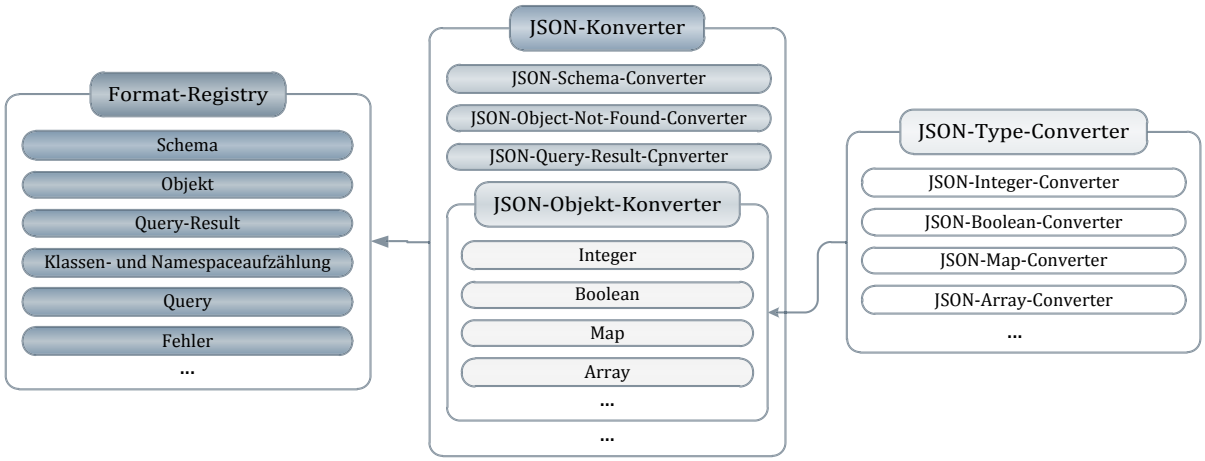


Abbildung 55 Aufbau der Format-Registry mit seinen Konvertern

Diese Konverter werden in Orestes von der *Format-Registry* verwaltet, an der auch zusätzliche Formate für andere Medientypen registriert werden können, um die Austauschbarkeit der Übertragungsformate zu ermöglichen. Dabei wird immer ein Konverter für eine bestimmte Datenstruktur und Medientyp registriert. Abbildung 55 zeigt den schematischen Aufbau der Format-Registry, in der links zunächst die Datenstrukturen aufgelistet sind, die

von Orestes übertragen werden können. Für jeden dieser Datenstrukturen können mehrere Konverter für jeweils einen Mediatyp registriert werden. In der Mitte der Abbildung ist dies beispielhaft für den Medientyp JSON aufgezeigt. Der JSON-Schema-Converter ist also verantwortlich für die Hin- und Rücktransformation von Schemarepräsentationen, zwischen `ISchema` und der JSON-Repräsentation des Schemas. Eine Besonderheit stellt der Objekt-Converter dar, der die Orestes-Objekte konvertiert. Da ein Orestes-Objekt in den Feldern verschiedene Datentypen aufweist, wie beispielsweise eine Zahl oder eine Map, sind Objekt-Converter selbst Registrys. An diesem können für den Medientyp, den der Objekt-Converter konvertieren kann, Typ-Converter registriert werden. Diese sind in der Abbildung rechts zu sehen und stellen die Typ-Converter für den JSON-Objekt-Converter dar. Sie sind also selbst Konverter die JSON konvertieren können.

4.3 Verwendung von Orion

Die Verwendung des Orestes-Clients werden wir an einem Beispiel verdeutlichen. Das Beispiel soll sowohl den Stand der Implementierung verdeutlichen, als auch die Verwendung der Orestes-Schnittstelle von Orion, wie sie aus der Einbettung in eine Client-Persistenz-API erfolgen würde.

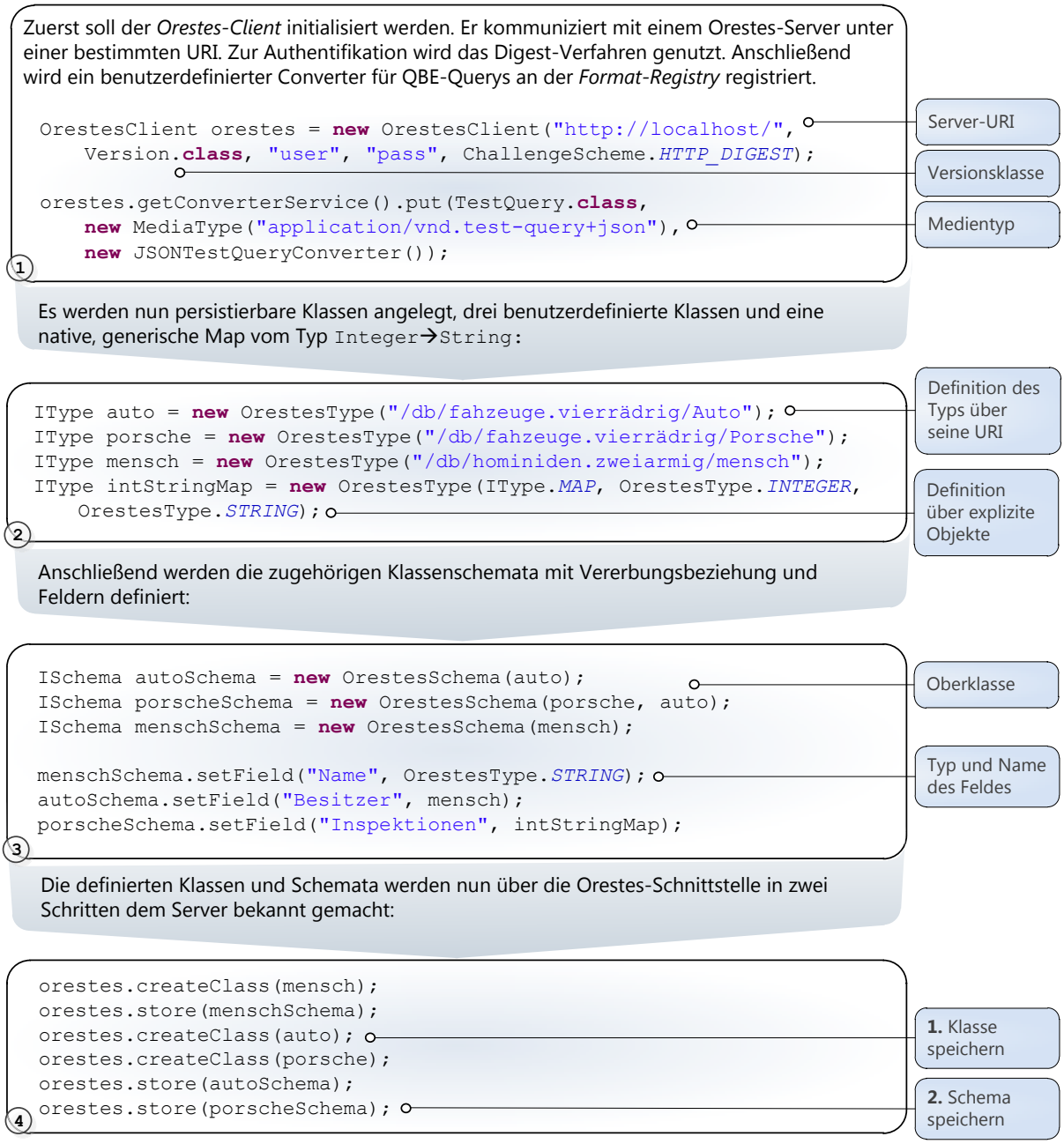


Abbildung 56 Beispiel für die Verwendung der Orestes-Schnittstelle von Orion, Teil 1

Das Beispiel ist in mehrere Schritte unterteilt (Abbildung 56). Zuerst erfolgen die Initialisierung des Orestes-Clients und die Registrierung eines spezifischen Formatkonverters für eine bestimmte (Query-)Klasse①. Ziel von Orion ist „Convention Over Configuration“, Konverter für kontext- und datenbankspezifische Repräsentationsformate müssen nur registriert werden, falls die implementierten Formate (JSON) unerwünscht sind. Anschließend werden

Klassen② und Schemata③ erzeugt. Diese werden vorerst lediglich clientseitig gehalten. Im letzten Schritt erfolgt die Übermittlung der Klassen und Schemata über die Orestes-Schnittstelle④.

Abbildung 57 Beispiel für die Verwendung der Orestes-Schnittstelle von Orion, Teil 2

Im zweiten Teil des Beispiels (Abbildung 57) werden nun Objekte der Klassen erstellt, deren Schemata zuvor der Datenbank übermittelt wurden⑤. An diesen Objekten werden Felder

gesetzt, sowohl Sammlungsdatentypen⑥ als auch primitive Typen⑦. Anschließend werden die Objekte persistiert. In einer Transaktion wird daraufhin ein zuvor gespeichertes Objekt abgerufen⑧. Nachdem dieses Objekt geändert wurde, wird es mit dem Commit der Transaktion dem Server übermittelt⑨. Das Beispiel zeigt, dass die Orestes-Operationen denen einer Persistenz-API auf niedrigerer Abstraktionsebene gleichen.

Das Zusammenspiel der verschiedenen Komponenten von Orion, Datenbank und Persistenz-API ist in Abbildung 58 gezeigt. Das Szenario ist folgendes: Ein Client persistiert aus Java mit der Java-Data-Objects-API (JDO) ein Objekt. Gezeigt ist die Übermittlung des Objekts in fünf Schritten, die nicht aufgeführte Antwort wird analog generiert. Um ein an der Schnittstelle entgegengenommenes Objekt in HTTP repräsentieren zu können, bedienen sich Orestes-Client und Server in Orion einer Format-Registry, an der für zu übertragene Klassen ein Konverter registriert werden kann. Konverter leisten auch die Rückkonvertierung eines Medientyps in eine korrespondierende Java-Klasse (hier `OrestesObject`). Die Konverter für Objekte sind zweistufig aufgebaut (Objektkonverter→Konverter für Felddatentypen).

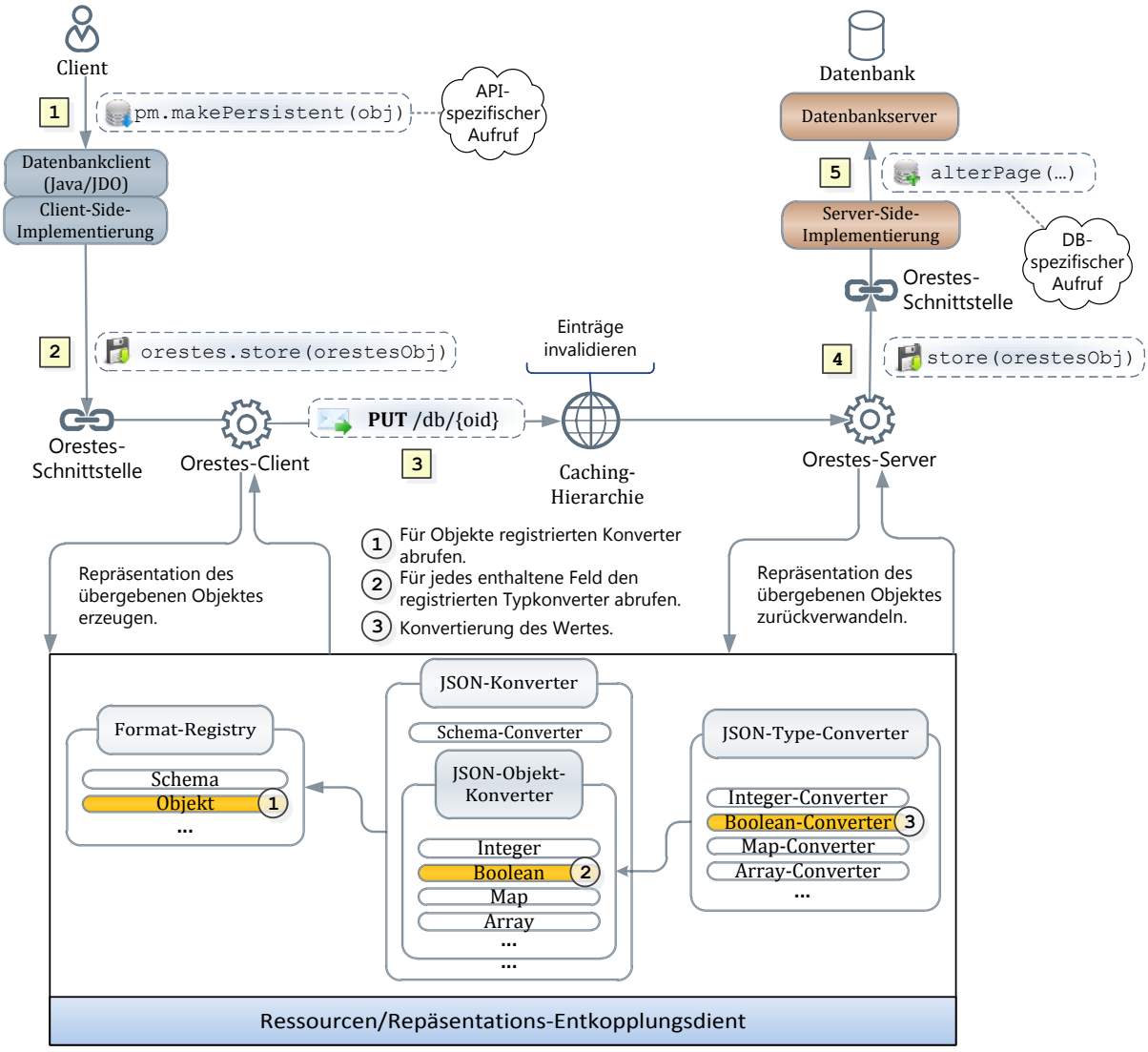


Abbildung 58 Die Rolle ausgewählter Orion-Komponenten bei einer Speicheroperation

4.4 Eingesetzte Frameworks

4.4.1 Restlet

Restlet [LoBo09] ist ein Java-Framework, das Zugriff auf die HTTP-Schnittstelle und REST-Konzepte auf Objektebene anbietet. Restlet war dabei zunächst als Erweiterung für Java-Servlets gedacht, um Aufbauend auf diesen REST-Archietekturen umsetzen zu können. Im weiteren Verlauf der Entwicklung haben sich aber immer mehr Anwendungsfelder ergeben, so dass Restlet nicht mehr ausschließlich auf Servlets aufbaut, sondern weitere HTTP-Connectoren anbietet, die die HTTP-Kommunikation auch über anderen Schnittstellen abwickeln können. Einer der neuern Errungenschaften von Restlet ist die Einsetzbarkeit in der Google-Cloud „Google App Engine“ [San09] und dem Google Handybetriebssystem „Android“ [RLM+09] auf dem ebenfalls Java läuft. Folglich könnte auch Orestes in diesen Umgebungen zum Einsatz kommen.

Restlets Architektur lehnt sich sehr stark an die JAX-RS-Spezifikation JSR-311 [HaSa09] an, die eine Java API für RESTfull Webservices spezifiziert. Diese wird von Sun Microsystems in dem eigenen REST-Framework, Jersy [Sun09] umgesetzt und weiterentwickelt. Eine Umstellung von Orestes auf das Jersy-Framework wäre somit ebenfalls realisierbar.

Wir haben die Version 2 des Restlet-Frameworks eingesetzt, da dieses eine große Reihe an Neuerungen mit sich bringt, unter anderem die Unterstützung von Request Pipelining und der asynchronen HTTP-Request Behandlung. Da die aktuelle Version sich aber noch im Testing-Stadium befindet, konnten wir leider noch nicht alle Neuerungen im vollen Maße ausnutzen, sodass z.B. die performante Umsetzung der *all*-Methoden noch nicht möglich war.

4.4.1.1 Abstraktion von Ressourcen

In Restlet werden Ressourcen sowohl auf Client- als auch auf Serverseite durch Klassen modelliert. Diese Ressourcen besitzen jeweils ein Request- und Response-Objekt, an dem über verschiedene Methoden die jeweiligen HTTP-Header ausgelesen und gesetzt werden können. Dabei sind alle standardkonformen Header durch entsprechende Objektstrukturen abstrahiert und können somit sehr einfach ausgelesen und gesetzt werden. Restlet wandelt die jeweiligen Objektstrukturen beim Absenden in die Textrepräsentation der HTTP-Header um und beim Empfangen wieder zurück in die jeweiligen Objektstrukturen. Auch der Content, der übertragen wird, kann durch Objekte abstrahiert werden und mithilfe entsprechender Typ-Konverter in die jeweilige Textrepräsentation und zurück konvertiert werden.

4.4.1.2 Aufbau des Restlet Servers

Der Restlet-Server verarbeitet die Requests, über ein komplexes Routingsystem. Dabei können mit Hilfe von URI-Templates verschiedene URIs auf gleiche Ressourcen-Implementierungen geroutet werden. Um eine solche Route zu definieren, gibt es in Restlet den *Router*, an dem URI-Templates registriert werden können. In Orestes ist die „oid“-Ressource unter einer konkreten Transaktion immer der gleiche Ressourcen-Typ. An diesem können per POST-Request neue OIDs allokiert werden und per GET-Request alle allokierten OIDs abgerufen werden. Diese Ressource wird am *Router* mit der Template-URI

„/transaction/{tid}/oids“ registriert, wobei *tid* ein Platzhalter für die spezifische Transaktions-ID ist. Wird nun ein GET-Request an die Ressource „/transaction/3/oids“ gesendet, so routet der *Router* den Request an den spezifischen Ressourcen-Typ weiter und stellt in dem „tid“-Argument den Wert *3* zur Verfügung. Ein Ressourcen-Typ wird in Restlet durch eine Klasse abstrahiert, die von der `ServerRessource`-Klasse erben muss. In Orestes heißt diese Klasse für die „oid“-Ressource `TransactionOids`. Diese Klasse kann nun beliebige Methoden deklarieren, die die verschiedenen HTTP-Methoden verarbeiten. Mithilfe von Annotations kann an der jeweiligen Java-Methode deklariert werden, welche HTTP-Methode diese verarbeitet. In der besagten `TransactionOids`-Klasse wird eine Methode `getAllocatedOIDs` deklariert, die mit der Annotation `@Get` versehen ist. Diese Methode verarbeitet somit alle GET-Requests für diesen Ressourcen-Typ. Dieses Annotationsprinzip ist das Kernkonzept der JSR-311 Spezifikation.

Die Methoden können, wenn sie beispielsweise ein POST-Request verarbeiten, noch einen Methodenparameter aufweisen, wobei der Content des Request dann mit Hilfe eines Konverters in eine Instanz des angegebenen Parametertyps konvertiert wird. Gibt die Methode noch ein Objekt zurück, so wird diese Instanz ebenfalls wieder mit einem entsprechenden Konverter konvertiert und im Content des Response zurückgeschickt. Die `getAllocatedOIDs` Methode gibt ein `OIDs`-Objekt zurück, das durch einen entsprechenden Konverter beispielsweise in eine URI-Liste konvertiert wird.

4.4.1.3 Aufbau des Restlet-Clients

Der Restlet-Client wird, im Gegensatz zum Server, durch eine generische Ressource abstrahiert, die jede beliebige Ressource eines Servers repräsentieren kann. Diese generische Ressource wird durch die `ClientRessource`-Klasse abstrahiert und bei der Instanziierung dieser Klasse wird eine URI mit angegeben. Diese gibt an, welche Ressource diese repräsentieren soll. Die `ClientRessource`-Klasse bietet für jede HTTP-Methode eine äquivalente Java-Methode an, mit deren Hilfe der jeweilige Request gesendet werden kann. Wenn also ein Client alle bis dahin allokierten Objekt-IDs der Transaktion *3* anfragen will, so instanziiert Orion eine neue `ClientRessource` mit der URI „/transaction/3/oids“. Anschließend ruft er an der Instanz die Methode `get(OIDs.class)` auf, um ein GET-Request abzusetzen und eine Instanz der übergebenen Klasse zu erhalten. Diese Instanz wird dabei mit Hilfe eines Konverters in Restlet erzeugt, der den Content der Response in die Instanz der angegebenen Klasse umwandelt. Hier würde dieser Konverter also die vom Server empfangene URI-Liste wieder in ein `OIDs`-Objekt zurück konvertieren.

4.4.2 Apache Commons Lang

Apache Commons Lang, ist ein Utility-Framework, das Schwachstellen der internen Java-API auszubessern versucht. Es stellt eine große Anzahl an Hilfsmethoden zur Verfügung, um native Datentypen zu verarbeiten. Der Einsatz dieses Frameworks spielte bis jetzt noch eine nicht sehr tragende Rolle für die Entwicklung von Orestes und der weitere Entwicklungsprozess wird zeigen, ob diesem Framework noch eine größere Bedeutung gewidmet wird oder es sich evtl. sogar erübrigt.

4.5 Performance-Aspekte

Neben der Beschleunigung der Kommunikation durch die Involvierung von Web-Caches kommen in Orion weitere Aspekte performanter Netzwerknutzung zum Einsatz. Dies ist erforderlich, um die Datenraten von proprietären auf TCP-basierenden Protokollen von OODBMS-Schnittstellen zu erreichen. Die Techniken, die dazu genutzt bzw. vorgesehen sind, werden wir im Folgenden schildern.

4.5.1 Persistent Connections

Die HTTP-Kommunikation läuft im WWW und bei Orion auf Basis von TCP (*Transport Control Protocol*) ab. Abbildung 59 zeigt den Austausch eines Request/Response-Paares aus Sicht von TCP als Sequenzdiagramm. In dem Beispiel wird die TCP-Verbindung nach Übertragung des Request/Response-Paares abgebaut. Dieses Vorgehen ist aus mehrerlei Gründen unvorteilhaft [Gri02, S.16]:

- Durch den Auf- und Abbau der TCP-Verbindungen verzögert sich die Übertragung um jeweils mindestens zwei Round-Trip-Times (RTT). Zusätzlich müssen bei Aufbau der Verbindung durch das Betriebssystem neue Ressourcen (Puffer, Zustandsvariablen, etc.) allokiert werden.
- Bei Paketverlusten im Verbindungsaufbau entstehen signifikante Verzögerungen.
- Durch seinen Slow-Start-Algorithmus passt sich TCP inkrementell und nur langsam der maximal verfügbaren Datenrate an, um verstopfte (engl. congested) Netzwerke nicht zu überlasten. Bei jedem neuen Verbindungsaufbau wird die Übertragungsrate mit einem konservativen Wert initialisiert und schrittweise erhöht.

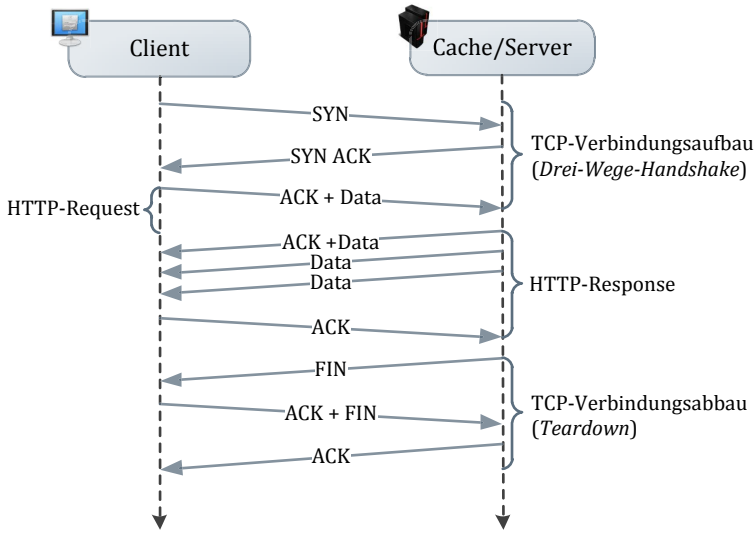


Abbildung 59 Übertragung eines Request/Response-Paares über TCP nach [Gri02]

Die Spezifikation von HTTP nimmt Rücksicht auf die genannten Aspekte und führt mit Persistent Connections einen HTTP-Header (`Connection`) ein, um Verbindungen explizit über die Grenzen eines Request/Response-Paares hinweg offen zu halten.

In Orion setzen wir persistente Verbindungen ein, so dass die Datenrate bei Zugriffen über die Orestes-Schnittstelle mit einer steigenden Anzahl von Zugriffen anwächst. Anders als in vielen Web-Servern umgesetzt, wird die offene TCP-Verbindung jedoch nicht nach einer kurzen Zeitspanne geschlossen (bei Web-Servern z.B. nach Laden aller in HTML-Seiten eingebetteten Elemente) sondern langfristig offen gehalten. Dies garantiert, dass sich die Datenrate der vorliegenden Netzwerksituation anpasst und Anfragen an den Orestes-Server mit hoher Übertragungsgeschwindigkeit abgewickelt werden.

4.5.2 Kompression

Kompression ist ein wichtiges Verfahren, um die Größe einer über HTTP versendeten Repräsentation zu minimieren. Der Orestes-Server versendet mithilfe der Server-driven Content-Negotiation eine Repräsentation stets in einer komprimierenden Codierung, die der Client zu empfangen bereit ist. Die Codierung erstreckt sich in HTTP ausschließlich über den Body der Nachricht. Die in HTTP verwendeten Kompressionsstandards *gzip* [Deu96a] und *deflate* [Deu96b] sind eine Kombination einer wörterbuchbasierten Kompression mithilfe des Lempel-Ziv-Storer-Szymanski-Algorithmus und der Huffman-Entropiekodierung. Für textbasierte Inhalte, wie sie auch bei Orestes primär transportiert werden, haben Destounis et al. eine durchschnittliche Kompressionsrate von 75,2% ermittelt [DGK+01].

4.5.3 Caching aller Ressourcen

In Orion ist jede Ressource bar.cachebar. Da nicht für jede Ressource eine exakte Gültigkeitsdauer vorhergesagt oder die Auslieferung veralteter Ressourcen kompensiert werden kann, werden derzeit für alle Nicht-Objekt-Ressourcen MD5-Hashwerte über der Repräsentation der Ressource gebildet. Diese werden als `ETag` zusammen mit einer `must-revalidate` Direktive übermittelt. Dies gestattet den Web-Caches, die entsprechende Ressource zu speichern. Bei jeder Anfrage bezüglich der gespeicherten Ressource validiert der Cache durch einen konditionalen Request die Gültigkeit der gespeicherten Repräsentation. Dies spart einerseits Netzwerklast ein und kann andererseits den Orestes-Server dadurch entlasten, dass komplexe Neugenerierungen von Repräsentation entfallen.

4.5.4 Vorgesehene beschleunigende Techniken

Bei einigen Operationen, die der Orestes-Client bereitstellt, sind als Abbildung auf die HTTP-Ebene mehrere Anfragen notwendig. Solche parallel verarbeitbaren Requests können in Orestes an mehreren Stellen genutzt werden:

- Bei Anfragen der Objekte eines Queryergebnisses wird jedes Objekt dediziert angefragt. Dies kann parallel geschehen.
- Bei dem Austausch von Metadaten oder dem optionalen, initialen Laden aller Klassen-Schemata, können diese nicht aufeinander aufbauenden Anfragen parallelisiert werden.
- Auch bei Lösch- und Änderungsoperationen ist eine gebündelte Übertragung in einigen Fällen sinnvoll.

Bei Orestes werden derartige Anfragen nicht durch eine Sammlungsressource modelliert, da nur durch das separate Anfragen und Ändern jeder Ressource die Web-Caches einbezogen werden. Zur performanten Umsetzung solcher gebündelten Anfragen können zwei Techniken eingesetzt werden, die in Orion schon vorbereitet und implementiert, aber derzeit noch nicht für konkrete Operationen nutzbar sind:

Multiple Simultaneous Connections
Hierbei wird eine größere Anzahl paralleler TCP-Verbindungen geöffnet, über die gleichzeitig HTTP-Anfragen abgesetzt werden. Diese Technik erfordert keine Unterstützung durch die Infrastruktur und verkürzt durch Nebenläufigkeit die Latenzzeit einer Anfrage.

Request Pipelining
Eine mit HTTP/1.1 eingeführte Erweiterung des Request/Response-Modells ist das sogenannte Request-Pipelining [Not10b]. Hierbei versendet der Client eine Reihe von idempotenten HTTP-Anfragen (z.B. GETs) über eine TCP-Verbindung (persistente HTTP-Connection), ohne zuvor die Antwort des Servers abzuwarten (Abbildung 60).

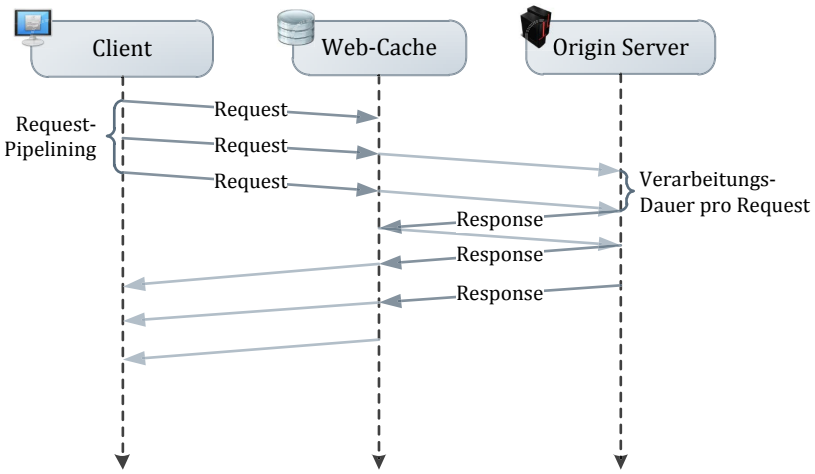


Abbildung 60 Request-Pipeling in HTTP

Die beteiligten Intermediaries leiten die Anfragen an den Server weiter. Falls dieser wie der Orestes-Server Request Pipelining beherrscht, sendet er sukzessive alle Antworten in der Reihenfolge der empfangenen Anfragen. Kommt ist bei einen Intermediary-Cache zu einem Cache-Hit, sendet er die Antwort ebenfalls in der Reihenfolge der erhaltenen Anfragen (Vermeidung von clientseitigen Fehlzuordnungen). Dabei ist die Länge der Antwort (`Content-Length`) ein unverzichtbarer Header, um in dem TCP-Stream die Grenzen der einzelnen Responses ermitteln zu können. Request Pipelining ist, obwohl seit 1999 Bestandteil der Spezifikation, bisher unzureichend implementiert (vor in Web-Browsern und gängigen Web-Servern) [Not10b, K.3]. Ein Problems, dass durch Request Pipelining bei entsprechender Unterstützung durch die Infrastruktur gelöst werden kann, ist das *Head-Of-Line Blocking*: Ein HTTP-Request in einer Sendewarteschlange kann erst gesendet werden, sobald der erste vorherige Request aus der Wartschlange (Head-Of-Line) durch den Server beantwortet wurde.

Für Orestes ist jedoch durch eine Kombination aus Request-Pipelining und parallelen Verbindungen ein hoher Performancegewinn für Anfragen an mehrere Ressourcen zu erwarten. Bei sehr großen Datenmengen (z.B. Query-Results aus vielen tausend Ergebnissen) kann als Einzelfall die Übertragung als aggregierte Repräsentation sinnvoll sein. Für ihre Umsetzung planen wir, das diesbezügliche Verhalten des Orestes-Servers durch URI-Parameter zu steuern (z.B. „GET /query/001?depth=2“ für eine bis Ebene zwei instanziierte Query-Ergebnismenge).

Architekturbedingte Performanceaspekte sind in Orion ebenfalls vorhanden, so wird das Leistungsverhalten durch Nebenläufigkeit der Request-verarbeitenden Einheiten des Servers weiter erhöht. Der Orestes-Server skaliert also mit der Anzahl zur Verfügung stehenden Prozessoren und Kerne. Insgesamt trägt neben dem konsequenten Caching aller Ressourcen die Verwendung von persistenten Verbindungen und Datenkompression in Orion dazu bei, Netzwerklast einzusparen und unter der maximal verfügbaren Datenrate zu kommunizieren.

5 Ausblick und Weiterentwicklung

Mit Orestes haben wir einen Ansatz vorgestellt, um Objekte über eine REST-Schnittstelle so bereitzustellen, dass Web-Caches für die Minimierung von Latenzzeit und Netzwerklast genutzt und objektbezogene Operationen standardkonform auf HTTP abgebildet werden. In dieser Arbeit haben wir die Ressourcenstruktur mit ihren Operationen, die Aspekte des Formatdesigns und eine generisch nutzbare Übertragungsschicht entwickelt.

Da das in Literatur und Praxis bisher einzigartige Ziel von Orestes Überschneidungen mit vielen Ansätzen und Technologien aufweist, ist eine vollständige Umsetzung aller im Rahmen von Orestes identifizierten Anforderungen und Konzepte von großem Umfang. Aufgrund der vielseitigen Herausforderungen des Orestes-Ansatzes sehen wir zahlreiche Entwicklungsmöglichkeiten, deren weitere wissenschaftliche Untersuchung und Ausarbeitung wir anstreben. Zu den verbleibenden Punkten zählen:

Gebündelte Abrufe und Änderungen

Die gebündelte Behandlung von Lese- und Schreibvorgängen ist Teil der bestehenden API, dennoch werden diese in Orion nur mangelhaft unterstützt. Eine performante Umsetzung dieser Operationen mithilfe von Request-Pipelining ist eine Möglichkeit, die jedoch bezüglich der mangelhaften Unterstützung durch Web-Caches genauer untersucht werden muss. Auch eine asynchrone Erweiterung der Datenoperations-API wäre denkbar, um mehrere unterschiedliche Operationen im Hintergrund parallel ausführen zu können (z.B. ein nicht blockierender Aufruf von `store(obj)`).

Abruf eines Änderungsprotokolls vom Orestes-Server

Eine Option, um zwischenzeitliche Änderungen von Objekten an Clients zu übermitteln, ist die Bereitstellung eines Änderungsprotokolls als Ressource des Orestes-Servers. Zu der Funktionalität einer solchen Ressource sollte der Abruf von Änderungen in einem bestimmten Zeitraum und einer bestimmten Extension gehören. Dies würde den Client zu vielfältigen Möglichkeiten verhelfen:

- Prüfung, ob von Web-Caches empfangene Objekte in dem vom Server abgerufenem Änderungsprotokoll enthalten sind → Revalidierung initiieren
- Nutzung ungebundener Kapazitäten von Netzwerk und Client: Ausführung invalidierender Aufrufe (z.B. HEAD-Methode mit If-None-Match-Header) für alle im Änderungsprotokoll für einen bestimmten Zeitraum aufgeführten Objekte bestimmter Versionen, in Situationen geringer Auslastung.

Das Änderungsprotokoll kann bei vielen schreibenden Operationen am System große Ausmaße erreichen. Da es jedoch ausschließlich auf Lesezugriffe ausgerichtet ist, können einerseits effiziente Datenstrukturen verwendet werden und andererseits Web-Caches zur Speicherung von in der Vergangenheit liegenden Fragmenten des Änderungsprotokolls eingesetzt werden.

Dezentralisierung des Orestes-Servers

Durch die zustandslose Kommunikation bei dem HTTP-Protokolls ergibt sich die Möglichkeit, den Orestes-Server zu dezentralisieren, sodass die HTTP-Requests von mehreren HTTP-

Servern verarbeitet werden können. Ein gängiger Load-Balancer kann die Last dabei auf die verschiedenen Orestes-Server aufteilen. Da die Datenbank selbst nicht dezentralisiert ist, kann ein Orestes-Server konzipiert werden, der über ein TCP-basiertes Protokoll mit dem eigentlichem Datenbankserver kommuniziert. Der Orestes-Server wäre dabei in zwei Komponenten aufgeteilt. Ein Teil würde direkt auf dem Datenbankserver laufen und mit den jeweiligen Orestes-Teilen auf den verschiedenen HTTP-Server kommunizieren. Auf diese Weise wird beliebige Skalierbarkeit der Orestes-HTTP-Kommunikationsschicht möglich.

Implementierung einer Server- (OODBMS) und Client- (Persistenz-API) Einbettung

Einer der nächsten Schritte, bei der weiteren Entwicklung des Orestes-Ansatzes, ist die Einbettung in ein bestehendes Datenbanksystem und die Anbindung einer bestehenden Persistenz-API. Eine anschließende Untersuchung der Performanz macht einen daraus resultierenden Mehrwert quantifizierbar. Die Einbettung soll dabei zunächst in Kooperation mit der Versant AG und ihrem OODBMS VERSANT OBJECT DATABASE erfolgen, das dieses aufgrund von Eigenschaften wie optimistischer Mehrbenutzersynchronisation gut geeignet ist.

Untersuchung des Verhaltens von Web-Caches im Umgang mit Orestes-Objekten

Ein weiterer Teil der Untersuchung des Orestes-Ansatzes wird sich mit der Bewertung des Performanzgewinnes durch Web-Caches beschäftigen, insbesondere dem Verhalten in Bezug auf gecachete Orestes-Objekte. Dabei muss untersucht werden:

- Wie Web-Caches GET-Anfragen auf Objekte beschleunigen, wenn sie das Objekt aus ihrem Cache oder nach einer erfolgreichen Revalidierung ausliefern.
- Ob und in welchem Umfang Änderungen die per POST- und PUT-Request durchgeführt werden von Web-Caches an unbeteiligte Web-Caches propagiert werden.
- Wie standardkonform Web-Caches die spezifizierte Semantik der HTTP-Caching-Direktiven einhalten.
- Ob Web-Caches stets alle HTTP-Header korrekt und vollständig ausliefern und transaktional geänderte Objekte nicht cachen.
- Wie Caches Request-Pipelining behandeln.

Benchmarks und Vergleiche zwischen Orestes und anderen, proprietären Protokollen

Nachdem die Integration von Orestes in eine Persitenz-API und ein OODBMS vorgenommen ist, kann ein direkter Vergleich mit den proprietären Protokoll angestellt werden, um die Leistungsunterschiede der verschiedenen Datenoperationen für verschiedene Zugriffsszenarien und -profile aufzeigen.

Implementierung von Cache-Channels und Untersuchung ihrer Einsetzbarkeit

Mit der Cache-Channels Erweiterung von HTTP existiert eine Möglichkeit, um Einträge in Web-Caches bei geringer Belastung des Orestes-Servers weitgehend aktuell zu halten. Voraussetzung ist die Unterstützung der Cache-Channels durch Web-Caches. Deshalb ist es sinnvoll, die Verbreitung und Entwicklungsbemühungen der diesbezüglichen Unterstützung zu untersuchen. Anschließend kann die Nützlichkeit der Übermittlung einer Cache-Channel-URI in HTTP-Antworten des Orestes-Servers bewertet und ggf. implementiert werden.

Einführung eines Locking-Mechanismus für Hot-Spot-Objekte

In vielen praktischen Anwendungsszenarien existieren sogenannte *Hot-Spot-Objekte* (z.B. Zähler, Revisionsnummern, aggregierte/materialisierte Statistiken). Es wäre eine wertvolle Erweiterung der REST-Schnittstelle von Orestes, eine Operation bereitzustellen, mit der vorüberge-

hend einzelne Objekte gegen fremde Zugriffe gesperrt werden können. Da derartige Objekte aufgrund ihrer hohen Volatilität ohnehin für die Speicherung in Web-Caches ungeeignet sind, würde durch die Bereitstellung einer solchen Operation (z.B. als LOCK-Methode der HTTP-WebDav-Extensions) die Anzahl an vermeidbaren Transaktionsabbrüchen (Konflikte beim optimistischen Commit) verringert werden.

Untersuchung von statistisch abgeleiteten Cache-Controlling-Parametern

Die präzise, auf gesammelten, statistischen Daten beruhende Vorhersage von optimalen Caching-Parametern ist eine Möglichkeit, um die Effektivität von Orestes weiter zu erhöhen. Die statistischen Daten können der heuristischen Angabe von Gültigkeitsdauern für Objekte dienen, sowie der Ermittlung von Abhängigkeitsgraphen für Änderungen (z.B. „Änderung von Objekt A bedingt mit 80-prozentiger Wahrscheinlichkeit eine baldige Änderung von Objekt B“). Derartige Abhängigkeitsgraphen können genutzt werden, um bei Änderungen bestimmter Objekte auch die Caching-Parameter assoziierter Objekte anzupassen. Notwendige statistische Untersuchungen würden neben den zur Laufzeit gesammelten Daten, auch die Untersuchung der Abfragehäufigkeitsverteilung für Objekte eines OODBMS umfassen. Für die Abfragehäufigkeitsverteilung bei WWW-Objekten (z.B. HTML) kann häufig eine Zipf-Verteilung beobachtet werden [Nie00]. Eine mathematische und statistische Überprüfung der Zipf-Verteilung für Objektzugriffe würde es ermöglichen, durch das Protokollieren von Zugriffshäufigkeiten ein Optimum zwischen langer Gültigkeitsdauer (→drohende Transaktionsabbrüche) und kurzer Gültigkeitsdauer (→potentielle Belastung des Servers und Clients) zu ermitteln.

Implementierung einer Web-Oberfläche für den Orestes-Server

Durch die von Orestes umgesetzte Entkopplung von Ressourcen und Repräsentationen ist es (bereits jetzt) möglich ein Orestes-Server über eine HTML-basierte Web-Oberfläche zu nutzen. Die Ausgestaltung einer solchen Oberfläche kann administrative Funktionen wie Datenverwaltung (z.B. Schemata einsehen, Objekte ändern), Monitoring und Management in einer leicht zu erlernenden benutzerfreundlichen Weise bereitstellen.

Einführung einer Rechte- und Benutzerverwaltung für Orestes

Eine Rechteverwaltung wäre für die verschiedenen Daten des Orestes-Systems denkbar, um den Zugriff auf bestimmte Daten einzuschränken. Die Rechteverwaltung könnte mit Benutzer- und Gruppenrichtlinien umgesetzt werden, sodass Benutzer oder Gruppen nur gemäß ihrer Autorisierung Daten lesen, ändern und löschen können. Diese Richtlinien können beispielsweise für Objekte eines bestimmten Datentyps oder Namespaces gelten, ähnlich einer Dateisystemrechteverwaltung. Dieser Aspekt ist besonders dann sinnvoll, wenn ein Client aus einer nicht sicheren Umgebung auf die Datenbank zugreift (z.B. ein JavaScript-Client), sich also nicht authentisieren kann. So könnte der Orestes-Server für anonyme Zugriffe Leserechte für öffentliche Daten erteilen und die Erstellung neuer Daten beschränken oder ausschließen. Damit wäre die Nutzung von Orestes im Web auch browserclientseitig möglich.

Festlegung einer Struktur für Status/Info/Version/Settings-Orestes-Ressourcen

Die Status- und Informationsressourcen sind derzeit Ressourcentypen, deren Inhalt komplett der serverspezifischen Implementierung überlassen ist. Es wäre hier ebenfalls denkbar, eine einheitliche Struktur der Ressourcen zu vereinbaren, sodass es für einen generischen Client leichter möglich ist, mit verschiedenen Datenbanken zu kommunizieren. Auch sie Vereinheit-

lichung der Versionsnummern kann eine generische Implementierung einer clientseitigen Persistenz-API-Einbettung für verschiedene Datenbanken erleichtern. So wäre ein Bestcase-Szenario eine spezifische Orestes-Client Implementierung für jede Programmiersprache und eine spezifische Orestes-Server Implementierung für jedes OODBMS. Außerdem sollten feingranulare Einstellungen für Orestes selbst ergänzt werden, damit weitere spezifische Einstellungen verwaltet werden können.

Registrierung eines fehlenden Link-Relationstyps für gruppierte HTTP-Anfragen

Der Link-Header für HTTP-Nachrichten wird in Orestes zur Identifikation von Transaktionen genutzt. Der Relationstyp eines solchen Links ist jedoch eine *experimental URN* und somit von impliziter, nicht registrierter Semantik. Ein derartiger Relationstyp ist jedoch für viele Situation sinnvoll, um die Zusammengehörigkeit bestimmter Requests zu kennzeichnen. Ein Beispiel wäre die Übermittlung von Formulardaten, die asynchron, bei Ausfüllen des Web-Formulars erfolgt (z.B. erst persönliche, anschließend sachbezogene Informationen, die mit AJAX an einen Web-Server übertragen werden). Die Kennzeichnung derartiger HTTP-Requests würde es erlauben, die Zusammengehörigkeit explizit und dadurch nutzbar (z.B. für Load-Balancer) zu machen.

Standardisierung eines Multipart-Medientyps mit dedizierten Caching-Direktiven

Unter den MIME-Types nehmen Multipart-Medientypen eine gesonderte Stellung ein: sie erlauben es, unterschiedliche Meditentypen zu bündeln. Jeder darin gekapselte Medientyp kann je nach Typ (z.B. `multipart/mixed`) festgelegte Header besitzen (z.B. `Content-Type`). Es wäre denkbar einen Medientyp einzuführen (z.B. `multipart/cachable`), der dedizierte Caching-Direktiven und URIs der gekapselten Ressourcen enthält. Eine Anfrage an eine Sammelressource, die mit einer Repräsentation eines solchen Medientyps beantwortet würde, könnte ohne das Head-Of-Line-Blocking Problem übertragen werden. Gleichzeitig umgeht dies die mangelnde ebarkeitCachebarkeit, wie sie beispielweise durch neue Ansätze wie *Resource Packages* [Lim09] entsteht. In Orestes wären auf diese Weise sehr performant durchführbare, gebündelte Anfragen (z.B. über ein Query-Ergebnisse) möglich, die anschließend überdies als Einträge in Web-Caches verfügbar sind. Ein Übermittlung eines solchen Ansatzes an die HTTPbis-Working-Group der IETF als Internet Draft wäre deshalb erstrebenswert und auch für andere Kontexte des Webs von Vorteil (z.B. Bündelung von Bildern/Stylesheets/Scripts einer HTML-Seite).

Wie an der Vielschichtigkeit der verbleibenden Entwicklungsmöglichkeiten deutlich wird, ist sowohl für die Weiterentwicklung von Orestes, als auch seine statistische Evaluation in der Benutzung und Kombination mit Infrastruktur und Anwendungen weitere Forschungsarbeit unabdingbar.

Wir konnten mit dem Orestes-Ansatz zeigen, dass objektbezogene Operationen vollständig über HTTP abgewickelt werden können und daraus entscheidende Vorteile erwachsen.

A Anhang

A.1 Tabellenverzeichnis

A.2 Abbildungsverzeichnis

Alle in der Arbeit verwendeten Abbildungen sind eigene Darstellungen, erstellt mit Microsoft Visio 2010.

A.3 Quellen und Referenzen

[ABD+92] Atkinson, M; Bancilhon, F; DeWitt, D; Dittrich, K; Maier, D; Zdonik, S: The Object-*Oriented Database System Manifesto.* Building an Object-Oriented Database System, Morgan Kaufmann 1992

[ADJ+99] Dilley, J; Arlitt, M; Perret, S; Jin, T: *The distributed object consistency protocol.* Technical Report HPL-1999-109, Hewlett-Packard Laboratories, September 1999. http://www.hpl.hp.com/techreports/1999/HPL–1999–109. html.

[AKM07] Slee, M; Agarwal, A; Kwiatkowski, M.: *Thrift: Scalable Cross-Language Services Implementation*. April 2007. http://incubator.apache.org/thrift/static/thrift-20070401.pdf

[And10] Anderson, J.: *CouchDB: The Definitive Guide.* Februar 2010.

[BBC+09] Bizer, C; Becker, C; Cyganiak, R; Garbers, J; Maresch, O: The D2RQ Platform v0.7 - Treating Non-RDF Relational Databases as Virtual RDF Graphs. September 2009 http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/

[BDK92] Bancilhon, F.; Delobel, C.; Kanellakis, P.: *Building an Object-Oriented Database System*. Morgan Kaufmann Publishers, San Mateo (Cal.). 1992.

[BEF+10] Brauer, B.; Endlich, S; Friedland, A; Hampe, J.: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Hanser Verlag. Oktober 2010.

[Ber08] Berg, R.: *How the 'Net works: an introduction to peering and transit.* September 2008. http://arstechnica.com/old/content/2008/09/peering-and-transit.ars/

[Ber09] Berners-Lee, T: Relational Databases on the Semantic Web. September 2009 http://www.w3.org/DesignIssues/RDB-RDF.html

[Blo70] B. Bloom. *Space/time Trade-offs in hash coding with allowable errors*. Communications of the ACM, Vol. 13, Nr. 7, S. 422–426, Juli 1970.

[BMF+05] Berners-Lee, T; W3C/MIT; Fielding, R; Day Software; Masinter, L; Adobe Systems: *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986. Januar 2005 http://tools.ietf.org/rfc/rfc3986.txt

[BPS+08] Bray, T; Paoli, J; Sperberg-McQueen, C. M.; Maler, E; Yergeau, F: Extensible Markup Language (XML) 1.0 (Fifth Edition). November 2008. http://www.w3.org/TR/REC-xml/

[Bre00] Brewer, E.: *Towards Robust Distributed Systems.* Juli 2000. http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[Bro09] Browne, J.: *Brewer's CAP Theorem, The kool aid Amazon and Ebay have been drinking*. Januar 2009. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[CDG+06] Chang, F; Dean, J; Ghemawat, S; Hsieh, W; Wallach, D; Burrows, M; Chandra, T; Fikes, A; Gruber, R.: *Bigtable: A Distributed Storage System for Structured Data.* November 2006. http://labs.google.com/papers/bigtable.html

[CDL01] Cao, P; Dahlin, M; Li, D.: *WCIP: Web Cache Invalidation Protocol.* März 2001. http://tools.ietf.org/html/draft-danli-wrec-wcip-01

[ChMa98] Chronaki, C.; Markatos, E.*: A TOP-10 approach to prefetching on Web*. Proceedings of INET'98.

[Cho10] Chodorow, K.: *Querying.* August 2010. http://www.mongodb.org/display/DOCS/Querying

[CKR98] Cohen, E; Krishnamurthy, B; Rexford, J.: *Improving end-to-end performance of the Web using server volumes and proxy filters.* ACM Computer Communication Review, 28(4):241–253, September 1998.

[Clo09] Cloudera, Inc.: *Hadoop Ecosystem Tour.* 2009. http://www.scribd.com/doc/13305091/Hadoop-Training-3-Hadoop-Ecosystem-Tour

[Coo99] Cooper, I.: *Inter Cache Co-operation, Protocol Extensions*. Internet Draft. Juli 1999. http://www.web-cache.com/Writings/Internet-Drafts/draft-cooper-intercache-cooper-00.txt

[Cri10] Cribs, S.: *RIAK Rest API.* 2010. https://wiki.basho.com/display/RIAK/REST+API

[Cro06] Crockford, D: *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627. Juli 2006 http://www.ietf.org/rfc/rfc4627.txt?number=4627

[DeGh04] Dean, J; Ghemawat, S: *MapReduce: Simplified Data Processing on Large Clusters.* Dezember 2004. http://labs.google.com/papers/mapreduce.html

[Deu96a] Deutsch, P.: *DEFLATE Compressed Data Format Specification.* RFC 1951. Mai 1996. http://www.http-compression.com/rfc1951.txt

[Deu96b] Deutsch, P.: *GZIP file format specification.* RFC 1952. Mai 1996. http://www.http-compression.com/rfc1952.txt

[DGK+01] Destounis, P.; Garofalakis, J.; Kappos, P.; Tzimias, J.: *Measuring the Mean Web page size and its compression to limit latency and improve download time.* Internet Research: Electronic Networking Applications and Policy 11, no. 1. 2001.

[DHS93] P. B. Danzig, R. Hall, M. F. Schwartz. *A Case for Caching File Objects Inside Internetworks*. Proc. ACM SIGCOMM '93, S. 239–248, San Francisco, September 1993.

[DiAr99] J. Dilley, M. Arlitt. *Improving Proxy Cache Performance: Analysis of Three Replacement Policies*. IEEE Internet Computing, Vol. 3, Nr. 6, S. 44–50, November 1999.

[Dim10] Dimitrov, M.: *NoSQL databases.* März 2010 http://www.slideshare.net/marin_dimitrov/nosql-databases-3584443

[DiPa95] A. Dingle, T. Partl. *Web Cache Coherence*. Proc. Fifth International World Wide Web Conference, Paris, Mai 1996.

[Ell09] Ellis, J.: *NoSQL Ecosystem.* November 2009. http://www.rackspacecloud.com/blog/2009/11/09/nosql-ecosystem/

[Eva09] Evans, E: *NoSQL: What's in a name?*. Oktober 2009. http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html

[Fie95] R. Fielding, *Relative Uniform Resource Locators.* RFC 1808, IETF, Juni 1995.

[FIG+99] Fielding, R; UC Irvine; Gettys, J; Compaq/W3C; Mogul, J; Compaq; Frystyk, H; W3C/MIT; Masinter, L; Xerox; Leach, P; Microsoft; Berners-Lee, T; W3C/MIT: *Hypertext Transfer Protocol -- HTTP/1.1.* RFC 2616. Juni 1999. http://www.ietf.org/rfc/rfc2616.txt

[FNH+99] Franks, J; Northwestern University; Hallam-Baker, P; Verisign, Inc.; Hostetler, J; AbiSource, Inc.; Lawrence, S; Agranat Systems, Inc.; Leach, P; Microsoft Corporation; Luotonen, A; Netscape Communications Corporation; Stewart, L; Open Market, Inc.; *HTTP Authentication: Basic and Digest Access Authentication.* RFC2617. Juni 1999 http://www.ietf.org/rfc/rfc2617.txt

[GoTo02] Gourley, D.; Totty, B.: *HTTP – The Definitive Guide.* USA, September 2002.

[Gre10] Gregorio, J. *URI Template.* März 2010. http://tools.ietf.org/html/draft-gregorio-uritemplate-04

[Gri02] Grimm, C.: *Simulation komplexer Cache-Verbünde im World Wide Web.* 2002

[GrNo09] Grossniklaus, M; Norrie, M.:*Object-Oriented Databases*. Oktober 2009. http://www.odbms.org/download/eth-oodb-2009.zip

[Här84] Härder, T.: *Observations on Optimistic Concurrency Control*. Information Systems 9 (2), 111-120,. 1984.

[Här87] Härder, T., Petry, E.: *Evaluation of Multiple Version Scheme for Concurrency Control. Information Systems* 12 (1), 83-98.1987.

[HaSa09] Hadley, M.; Sandoz, P.: *JAX-RS: Java™ API for RESTful Web Services.* Version 1.1 September 2009. https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec.html

[HiGo10] Hickson, I; Google, Inc.: HTML5. Juni 2010
http://www.w3.org/TR/html5/

[Ho08] Ho, R.: *CouchDB Implementation.* Oktober 2008.
http://horicky.blogspot.com/2008/10/couchdb-implementation.html

[Hur10] Hurst, N.: *Visual Guide to NoSQL Systems*. März 2010.
http://blog.nahurst.com/visual-guide-to-nosql-systems

[Inf09a] InfoGrid: *The Viewlet Framework*. Juni 2009.
http://infogrid.org/wiki/Docs/ViewletFramework

[Inf09b] InfoGrid: *Http Shell: Keywords*. Juli 2009.
http://infogrid.org/wiki/Docs/HttpShellKeywords

[Jef10] Jeffries, A. *Squid FAQ.*
http://wiki.squid-cache.org/SquidFaq/AboutSquid

[JLL01] Jacobs, L; Ling, G; Liu, X: ESI Invalidation Protocol. August 2001.
http://www.w3.org/TR/esi-invp

[Kam06] Kamp, P.: *Varnish - Notes from the architect.* 2006.
http://varnish.projects.linpro.no/wiki/ArchitectNotes

[Kap05] Kapferer, H.: *Consistency.* Januar 2005.
http://www.sti-innsbruck.at/fileadmin/documents/teaching_archive/acsp0405/
08_Kapferer_Ausarbeitung.pdf

[KlCa04] Klyne, G; Carroll J. J: Resource Description Framework (RDF) Februar 2004.
http://www.w3.org/TR/rdf-concepts/

[KrWi99] Krishnamurthy, B; Wills, B.: *Proxy cache coherency and replacement – towards a more complete picture*. In Proceedings of the ICDCS conference, June 1999.
http://www.research.att.com/~bala/papers/ccrcp.ps.gz.

[Kud07] Kudraß, T: *Taschenbuch Datenbanken.* Hanser Verlag. September 2007.

[KuRo10] Kurose, J., Ross, K.: *Networking – A Top-Down Approach*. 5. Auflage, 2010.

[KuRo81] Kung, H.T., Robinson, J.T.: *On Optimistic Methods for Concurrency Control.* ACM Trans. on Database Systems 6 (2), 213-226. 1981.

[Lac07] Lacey, P.: *Code Talks: Demonstrating the "ilities" of REST.* November 2007.
http://wanderingbarque.com/presentations/qcon2007.ppt

[Lim09] Limi, A.: *Making browsers faster: Resource Packages. A proposal to make downloading web page resources faster in all browsers.* November 2009.
http://limi.net/articles/resource-packages/

[LoBo09] Louvel, J; Boileau, T.: *Restlet in Action.* MEAP Began, November 2009

[Maz10] Mazumder, S: *NoSQL in the Enterprise.* April 2010. http://www.infoq.com/articles/nosql-in-the-enterprise

[Meg07] Megginson, D: *„With REST, every piece of information has its own URL."* Februar 2007 http://www.megginson.com/blogs/quoderat/2007/02/15/rest-the-quick-pitch/

[Mer10] Merriman, D.: *On Distributed Consistency.* April 2010. http://blog.mongodb.org/post/523516007/on-distributed-consistency-part-6-consistency-chart

[Mer10b] Merriman, D.: *NoSQL and MongoDB.* Juni 2010. http://www.se-radio.net/2010/07/episode-165-nosql-and-mongodb-with-dwight-merriman/

[MeSa04] Meinl, C.; Sack, H.: *WWW Kommunikation, Internetworking, Web-Technologien*. Springer Verlag, Xpert.press, 2004.

[MTV10] Völter, M; Tilkov, S.; Meyer, M.: *NoSQL - Alternative zu relationalen Datenbanken.* Juni 2010. http://www.heise.de/developer/artikel/Episode-22-NoSQL-Alternative-zu-relationalen-Datenbanken-1027769.html

[Nie00] Nielsen, J., "Zipf Curves and Website Popularity,". 2000.

[Not06] Nottingsham, M: *The state of Browser Caching*. 2006. http://www.mnot.net/blog/2006/05/11/browser_caching

[Not07a] Nottingsham, M: *The state of Proxy Caching*. 2007. http://www.mnot.net/blog/2007/06/20/proxy_caching

[Not07b] Nottingsham, M: *Cache Channels.* 2007. http://potaroo.net/ietf/idref/draft-nottingham-http-cache-channels/

[Not10a] Nottingsham, M: *Caching Tutorial for Web Authors and Webmasters. 2010.* http://www.mnot.net/cache_docs/

[Not10b] Nottingham, M.: *Making HTTP Pipelining Usable on the Open Web*. Web Draft. August 2010. http://tools.ietf.org/id/draft-nottingham-http-pipeline-00.html

[Not10c] Nottingham, M.: *Web Linking*. Mai 2010 http://ietfreport.isoc.org/all-ids/draft-nottingham-http-link-header-10.txt

[NRT02] Niranjan, T.; Ramamurthy, S; Tewari, R: *WCDP: A protocol for web cache consistency.* Februar 2002. http://2002.iwcw.org/papers/18500180.pdf http://www.ietf.org/proceedings/53/I-D/draft-tewari-webi-wcdp-00.txt

[Phil10] Philips, M: An Introduction to Riak. August 2010. https://wiki.basho.com/display/RIAK/An+Introduction+to+Riak

[Pri08] Pritchett, D.: *BASE: An Acid Alternative.* Juli 2008
http://queue.acm.org/detail.cfm?id=1394128

[Pur09] Purtell, A.: *HBase Stargate.* Juli 2009.
http://wiki.apache.org/hadoop/Hbase/Stargate

[Pus10] Puschke, K.: *Not only SQL - CouchDB und andere NoSQL-Datenbanken*. Juni 2010.
http://www.slideshare.net/titanoboa/couchdb-nosql-4403384

[Rah88] Rahm, E.: *Optimistische Synchronisationskonzepte in zentralisierten und verteilten Datenbanksystemen*. Informationstechnik 30 (1), 28-47. 1988.

[RiRu07] Richardson, L.; Ruby, S.: *RESTful Web Services.* Mai 2007.

[RLM+09] Rogers, R.; Lombardo, J.; Mednieks, Z.; Meike, B.: *Android Application Development Programming with the Google SDK* O'Reilly Media, Mai 2009

[RoWe98] A. Rousskov, D. Wessels. *Cache digests*. Computer Networks and ISDN Systems, Vol. 30, S. 2155–2168, November 1998.

[San09] Sanderson, D.: *Programming Google App Engine* O'Reilly Media, Dezember 2009

[Sun09] Sun Microsystems, Inc.: *RESTfulWeb Services Developer's Guide* April 2009
http://dlc.sun.com/pdf/820-4867/820-4867.pdf

[ThRa90] Thomasian, A., Rahm, E.: *A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking*, Proc. 10th IEEE Int. Conf. on Distributed Computing Systems, 294-301. 1990.

[Til09] Tilkov, S.: *REST und HTTP – Einsatz der Architektur des Web für Integrationsszenarien*. 2009.

[Vog08] Vogels, W.: *Eventually Consistent – Revisited*. Dezember 2008.
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[WBD+08] Wilde, E; UC Berkeley; Duerst, M; Aoyama Gakuin University: *URI Fragment Identifiers for the text/plain Media Type.* RFC 5147. April 2008
http://www.ietf.org/rfc/rfc5147.txt

[WeCl97] D. Wessels, K. Claffy. *Internet Cache Protocol (ICP)*, version 2. RFC 2186, IETF, September 1997.

[Wig09] Wiggins, A: *SQL Databases Don't Scale.* Juli 2009.
http://adam.heroku.com/past/2009/7/6/sql_databases_dont_scale/

[Zic09] Zicari, R.: *A New Renaissance for ODBMSs?* August 2009.
http://www.odbms.org/download/Panel.Renaissance.ODBMS.pdf

B Arbeitsaufteilung

Die Verfassung der vorliegenden, schriftlichen Ausarbeitung haben sich die Autoren wie folgt aufgeteilt:

	Florian Bücklers	Felix Gessert
Kapitel 1 Einleitung	zusammen: Abstract, 1.1, 1.4;	zusammen: Abstract, 1.1, 1.4; 1.2, 1.3
Kapitel 2 Zusammenhänge mit genutzten und verwandten Technologien		2.1, 2.2, 2.3, 2.4, 2.5
Kapitel 3 Ausarbeitung und Architektur des Orestes-Systems	3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8	3.1, 3.9
Kapitel 4 Implementierung	4.1, 4.2, 4.4	4.0, 4.3, 4.5
Kapitel 5 Ausblick und Weiterentwicklung	zusammen	zusammen

C Erklärung

Hiermit erkläre ich, dass ich, Felix Gessert, abgesehen von der gemeinsamen Planung und Besprechung der Inhalte, meinen Teil der Arbeit selbstständig und ohne fremde Hilfe angefertigt, sowie mich anderer als der im Literatur- und Quellenverzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Bachelorarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Hamburg, den

[Felix Gessert]

Hiermit erkläre ich, dass ich, Florian Bücklers, abgesehen von der gemeinsamen Planung und Besprechung der Inhalte, meinen Teil der Arbeit selbstständig und ohne fremde Hilfe angefertigt, sowie mich anderer als der im Literatur- und Quellenverzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Außerdem erkläre ich, dass ich mit der Einstellung dieser Bachelorarbeit in den Bestand der Bibliotheken der Universität Hamburg einverstanden bin.

Hamburg, den

[Florian Bücklers]