Middleware - Cloud Computing

Einführung

Was ist eigentlich Cloud Computing?

Cloud Anbieter stellen skalierbare Hardware-Ressourcen zur Verfügung, Software as a Service (SaaS) Anbieter bauen dann darauf Dienste für den Endnutzer auf.

Eigenschaften

- Verfügbarkeit von “unbegrenzten” Ressourcen
- Keine Kapazitätsplanung aus Sicht des Nutzers nötig
- **“Pay as you go”**-Modell

Vorteile

- geringe Anschaffungs- und Betriebskosten
- hohe Auslastung vorhandener Ressourcen (im Gegensatz zu traditionellen Rechenzentren)
- Nutzer sparen Kosten / reduzieren finanzielle Risiken
    - keine/wenig Hardware muss beschafft werden
    - Bedarfsabschätzungen nicht mehr nötig → keine Unter-/Überschätzung
    - Neue Dienste können schneller auf den Markt gebracht werden

Technische Basis und Herausforderungen

Cloud Computing baut auf einer Reihe von sogenannten Basistechnologien auf, dazu gehören Virtualisierung und Web Services. Auf beides wird später noch detailliert eingegangen.

Virtualisierung

- kann auf verschiedenen Ebenen erfolgen
- maximale Ausnutzung von Hardware Ressourcen (mehrere Benutzer auf einem physikalischen Rechner, Entkoppelung von Ausführungsort und Hardware)
- Isolation von Nutzern

Web Services

- sprachenunabhängige Basis für entfernte Kommunikation

Herausforderungen

- dynamische Skalierung von Diensten/Anwendungen
- Gewährleistung hoher Verfügbarkeit trotz Verwendung von Standardhardware
- Sicherheit
    - Angriffe von außerhalb oder innerhalb einer Cloud-Struktur
    - Informationssicherheit (Vertraulichkeit, Verfügbarkeit, Integrität)
    - besonders wichtig für Daten aus medizinischem Bereich, etc.

Ausprägungen von Cloud Computing

Infrastructure as a Service (IaaS)

Mittels Virtualisierung bereitgestellte Ressourcen: Server, Networking, Storage, ...
Beispiele: Amazon EC2, Eucalyptus

Platform as a Service (PaaS)

Konfektionierte Infrastruktur/Middleware zur Bereitstellung von Diensten: Datenbanken, Java Runtime, Development Tooling, Web 2.0 Application Runtime, ...
Beispiele: force.com, Microsoft Azure, Google AppEngine

Software as a Service (SaaS)

Vom Endnutzer oder anderen SaaS Einheiten genutzte Dienste (Kollaborationsplattformen, Geschäftsprozesse, Industrieanwendungen, ...
Beispiel: Google Apps, salesforce.com

Einsatzszenarien

Public Cloud

“typisches” Cloud Computing, man mietet Ressourcen

Private Cloud

Cloud für **einen** Nutzer (z.b. privat innerhalb von EC2) oder auch schon die durch Virtualisierung flexibilisierte Verwaltung von Ressourcen

Hybride Cloud

vermischtes public und private Cloud Computing, falls z.B. bereits eine Infrastruktur vorhanden ist, kritische Daten nicht ausgelagert werden dürfen/können oder Bedarfsspitzen gedeckt werden sollen.

Multi Cloud Computing

bezeichnet die parallele Verwendung verschiedener Cloud Anbieter

Web Services

Ein Web Service besteht daraus, dass eine Komponente einen Dienst anbietet und unter Ausführung einer Funktion aktiv eine Dienstleistung erbringt. Es werden standardisierte Protokolle und Konzepte benutzt, um Zugang zu Web Services (über das Web) zu erhalten, z.b. XML und HTTP.
Eine mögliche Definition für Web Services lautet: *"... ein Dienst, dessen Schnittstelle mit WSDL beschrieben, mit UDDI registriert und gefunden und mit SOAP angesprochen werden kann ..."*.
Die Motivation hinter Web Services ist die Vision vom "Markt der Web Services". Unabhängige Softwarefirmen verkaufen Web Services Software. Web Services sind nicht ortsgebunden, d.h. man kann aus Entfernung darauf zugreifen und man benötigt normalerweise keine lokale Software (-installation).

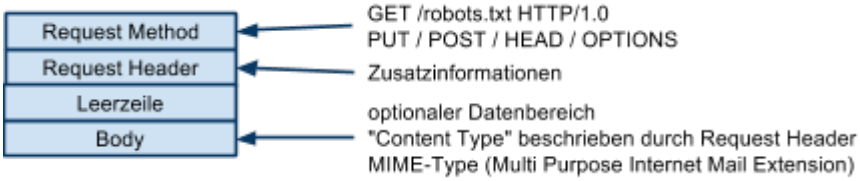
Grundlagen

Uniform Ressource Identifier (URI)

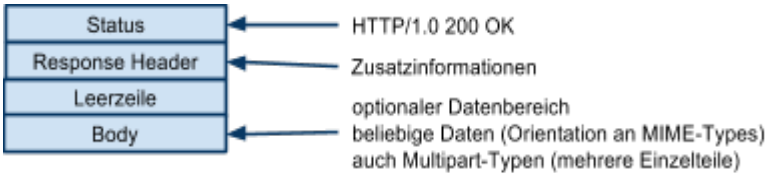
Ressourcen (Dokumente, Mailadressen, Websites, Objekte, …) sollen eindeutig referenzierbar sein und dazu soll ein einheitliches, erweiterbares Schema verwendet werden, das für Mensch und Maschine lesbar ist. Klassisch wurde in Ortsangabe der Ressource (Uniform Ressource Locator, URL) und Name der Ressource (URN) aufgeteilt. Heute ist alles eine URI. Aber es gibt verschiedene Schemata.
Beispielschema: `<schema> "://" [<user> [":" <pw>] @ ] host [":" <port>] ["/" <url-path>]`

Hypertext Transfer Protocol (HTTP)

- basiert auf TCP/IP (verbindungsorientiert)
    - früher: eine TCP/IP Verbindung pro Anfrage/Antwort-Paar
    - heute: Wiederverwendung der Verbindung für mehrere Anfragen; Pipelining möglich
- textbasiert (einfach analysierbar)
- Anfrage-Antwort (Request-Reply) Interaktion:
    - Client sendet Anfragenachricht (Request)



    - Server sendet Antwortnachricht (Response)



Extensible Markup Language (XML)

jedes XML Dokument besteht aus drei Teilen

- XML-Header (XML-Version und Zeichensatz)
- Verarbeitungsanweisungen
- XML-Elemente (Daten), können hierarchisch verschachtelt sein
    - können Attribute haben (bestehend aus Attributname und -wert)

    - eigentliche Daten stehen zwischen Start und Ende-Tag `<tag>daten</tag>`

Wohlgeformtheit

- sorgt für “Baumstruktur”
- jedes Start-tag hat zugehöriges Ende-tag
- Elemente dürfen geschachtelt sein, aber nicht überlappen
- genau ein Wurzelelement
- usw...

Gültigkeit

- Konkrete Strukturüberprüfung für ein korrektes XML-Dokument, basierend auf einer Dokumentenstrukturbeschreibung (DTD oder XML-Schema)
- Überprüfung der (gültigen) Anordnung von Elementen und Attributen

Namensräume

- jedes Element in einem XML Dokument kann einem Namensraum zugeordnet werden
- Identifikation des Namensraums durch URI

XML-Schema

- XML-Anwendung zur Beschreibung von XML-Anwendungen (ist selbst ein XML-Dokument)
- XML-Schema-Instanz ist ein XML-Dokument, das gültig ist bzgl. eines bestimmten XML-Schemas
- Strukturbeschreibung von XML-Dokument-Klassen (XML-Anwendung)
- Grundlage zur Validierung von XML-Dokumenten (Syntaxprüfung)
- … hat DTD mehr oder weniger abgelöst

XML Remote Procedure Call (XML-RPC)

Ein XML-RPC ist ein einfacher (primitiver) Fernaufruf, der auf XML-Nachrichten basiert. Es unterstützt zwar nur eine limitierte Menge von primitiven Datentypen (`int, double, string, boolean, dateTime, base64`), aber auch arrays und structs. XML-RPC benutzt HTTP als Trägerprotokoll und wird von nahezu allen aktuellen Programmiersprachen unterstützt.

Nachrichtenstruktur

Anfragenachricht (Request)

werden durch Methodennamen und Parameter beschrieben

HTTP Request

```
POST /account/4711 HTTP/1.0
Content-Type: text/xml
Content Length: 155
```

HTTP Body (die eigentliche Anfrage)

```
<?xml version=”1.0” ?>
<methodCall>
  <methodName>withDraw</methodName>
  <params> <param><value><double>200.25</double></value></param> </params>
</methodCall>
```

Antwortnachricht (Response)

HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 133
```

HTTP Body (die eigentliche Antwort)

```
<?xml version=”1.0” ?>
<methodResponse>
  <params> <param><value><double>23.42</double></value></param> </params>
  <fault> <value><string>Du Idiot!</string></value> </fault> <!-- falls Fehler auftritt -->
</methodResponse>
```

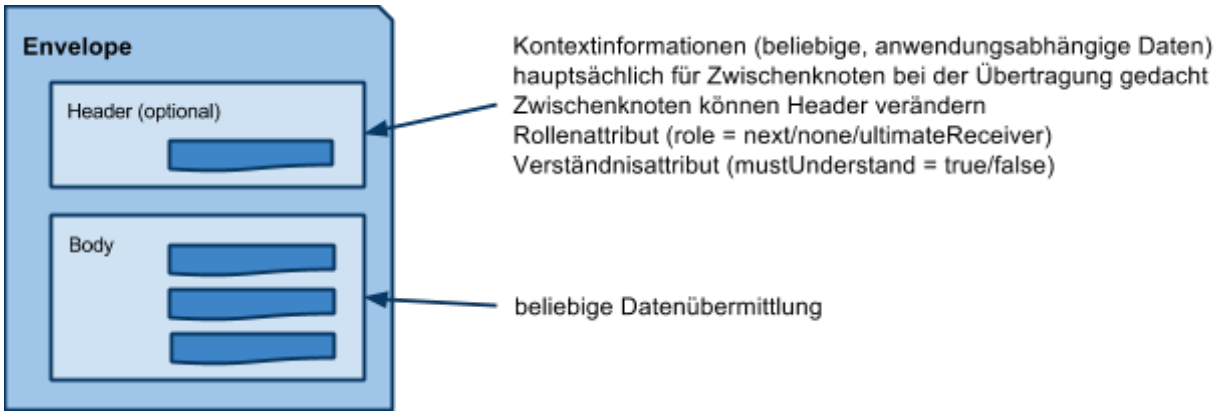
Simple Object Access Protocol (SOAP)

SOAP ist ein standardisiertes Kommunikationsprotokoll (basierend auf XML) für Web Services. Die Übertragung ist nachrichtenbasiert und verläuft vom Sender über Zwischenstationen zum Empfänger. Daher lassen sich aus mehreren Nachrichten komplexere Kommunikationsparadigmen zusammensetzen, z.B. Request-Reply-Protokolle (entfernter Aufruf) oder Multicast-Protokolle (mehrere Empfänger und/oder mehrere Antworten). SOAP Nachrichten sind XML Dokumente, die über ein Trägerprotokoll (HTTP, E-Mail, TCP/IP) transportiert werden.
SOAP kann auf vier Hauptbestandteile abgebildet werden.

Envelope

Definition des Nachrichtenumschlags, d.h. die Beschreibung des Nachrichteninhalts und dessen Verarbeitung. SOAP Tags werden über XML Namensräume von anwendungsspezifischen Tags getrennt. Der Envelope selbst ist auch eine XML Datei.

Grober Aufbau



Beispiel

```
<?xml version=”1.0” ?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <b:account-ref xmlns:b="http://treasurebank.com/acc" \
    soap:role="http://www.w3.org/2003/05/soap-envelope/role/next” soap:mustUnderstand="true">
      <b:account-number>4711</b:account-number>
    </b:account-ref>
  </soap:Header>
  <soap:Body>
    <a:withdraw xmlns:a="http://treasurebank.com/account"> <a:amount>200.25</a:amount> </a:withdraw>
  </soap:Body>
</soap:Envelope>
```

Sonstiges

- Fehlerübermittlung erfolgt über festgelegte Tags für fehlerhafte Ergebnisse und Fehlerbeschreibungen
- Entfernter Aufruf erfolgt über Tag, der die Methode/Funktion benennt (`withdraw`); die Aufrufparameter sind darin eingebettete Tags (`amount`)
- Verarbeitung in Zwischenknoten
    - Zwischenknoten können Header verändern
    - Bank-Beispiel

Encoding Rules

Definition von Codierungsregeln für **anwendungsspezifische** Datentypen, also die Umsetzung der Datentypen in XML (z.B. über XML-Schema-Datentypen).

Abbildungsregeln für Body- und Header-Daten in ein XML-Dokument

- festgelegt durch freies Attribut `soap:encodingStyle` bei einem Body- oder Header-Element, dessen Wert ein URI ist (charakterisiert Abbildungsregel); gültig für alle eingebetteten (Unter-)Elemente
- mögliche URI-Werte
    - leer (keine Angabe zur Abbildungsregel)
    - anwendungsspezifisch (Kennzeichnung eines bestimmten XML-Formats)
    - SOAP-Encoding
        - Rückgriff auf XML-Schema
        - Definition der Abbildung komplexer Datenstrukturen
        - z.B. Abbildung einer JAVA-Klasse in SOAP-Struktur (wie in Übung)

Fehlernachrichten

- genau ein `soap:fault` Element im Body
- Header enthält Hinweis auf nicht verstandene Elemente
- Beispiele für Fehlercodes: `VersionMismatch, MustUnderstand, Sender, Receiver, ...`
- Fehlercodes können durch untergeordnete Fehlercodes verfeinert werden
- HTTP-Fehlermeldungen gelten nicht für die SOAP-Verarbeitung (HTTP Status kann OK sein während SOAP-Verarbeitung Fehlernachricht produziert)

Abbildung auf Trägerprotokoll

Regeln zur Abbildung auf HTTP und E-Mail

HTTP

Es werden die `GET` und `POST` Methoden von HTTP benutzt (`GET` nur bei **rein lesendem** Zugriff). Der Adressat wird durch URL bestimmt, die unabhängig vom Inhalt der SOAP-Nachricht den Adressaten vollständig bestimmt. Als `Content-Type` wird `application/soap+xml` verwendet. Die Antwort ist i.d.R. auch eine SOAP-Nachricht. So lässt sich eine Request-Reply-Interaktion einfach realisieren.

E-Mail

Einbettung von SOAP-Nachrichten (gleicher `Content-Type` wie in HTTP) in E-Mails.

Choreographie für komplexe Interaktionen

Konventionen für die Nachrichten entfernter Aufrufe. Hier: **Request-Reply-Nachrichten**.
In der Aufrufnachricht ist der Aufruf eine Struktur, in der der Name der aufgerufenen Methode der Name des Elements ist und alle `in` und `inout` Parameter Komponente der Struktur sind. Die Antwortnachricht ist ebenfalls eine Struktur, in der der Name des Tags der Methodenname plus "Response" lautet. Alle `out` und `inout` Parameter sind Komponenten der Struktur. Für das Ergebnis gibt es ein spezielles Tag `rpc:result`.
Unter Verwendung von HTTP geschieht die Zuordnung von Antwort zu Aufruf durch die "natürliche" Zuordnung. In E-Mails wird die Message-ID des E-Mail-Systems bei der Antwort angegeben und im Envelope jeweils eine eindeutige Identifikation codiert (meist im Header).
Es existieren spezifische Fehlermeldungen, die auf Ressourcenprobleme, unbekannte Kodierung, unbekannte Methodennamen und falsche Parameter hinweisen.

Web Service Description Language (WSDL)

WSDL ist eine Sprache zur Schnittstellenbeschreibung für Web Services.

Definitionen (Beschrieben werden...)

- Types (definieren XML-Elemente für Nachrichten)
    - Typisierung durch XML Schema Definitionen
- Interface (definieren Operationen und Fehlermeldungen)
    - Aufführung der einzelnen Operationen und der dazugehörigen (Fehler-)Nachrichten
    - insgesamt 8 verschiedene Kommunikationsmuster (Message Exchange Pattern), z.B.
        - `in-out`: eine einkommende Nachricht, eine ausgehende Nachricht
        - `in-only`: nur einkommende Nachricht
    - Stil der Nachrichtenübertragung (Style)
        - definiert Einschränkungen über den Nachrichtenaufbau (z.B. für SOAP RPC)
    - Sicherheit (`wsdlx:safe` definiert, ob Aufrufer eventuelle Verpflichtungen eingeht)
    - (Vererbung ist möglich: Attribut `extends`)
- Binding (definieren Abbildung auf ein Protokoll, z.B. SOAP über HTTP)
    - Attribut `type` (z.B. SOAP 1.2 Binding: `type="http://w3.org/ns/wsdl/soap"`)
    - Attribut `wsoap:protocol` definiert Trägerprotokoll (`wsoap:protocol="…/HTTP"`)
    - `wsoap:mep` definiert Message Exchange Pattern der zu bindenden Operation
- Service (definiert *konkrete* Web Service *Instanz*)

Beispiel

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://w3.org/ns/wsdl" targetNamespace="http://tbank.com/wsdl/accSvc" ...>
  <types> ... </types> <interface> ... </interface> <binding> ... </binding> <service> ... </service>
</description>
```

Representational State Transfer (REST)

REST ist entstanden aus der Betrachtung der *Eigenschaften des WWW* und ist hauptsächlich motiviert durch die Möglichkeit der *Skalierbarkeit in großen Dimensionen*. REST ist ein Architekturstil zum Aufbau von Systemen.

Randbedingungen

Damit ein System RESTful ist müssen folgende Randbedingungen / Einschränkungen erfüllt sein.

Identifikation der Ressourcen

Die Benennung aller Business-Komponenten (Ressourcen). Hier sind keine Software Komponenten gemeint, sondern alle "Dinge", mit denen die Anwendung umgeht (vgl. Entities). Dazu gehören Produkte, Personen, Warenkörbe, ... ebenso auch Anwendungszustände werden als Ressource aufgefasst und benannt (z.B. Zwischenschritte bei komplizierten Aktionen).

einheitliche Schnittstelle

Kleine und uniforme Schnittstelle für Aktionen (*Verben*), die anwendbar auf viele unterschiedliche Ressourcen (*Substantive*) ist. Sie hat eine universelle Bedeutung (nicht zu anwendungsspezifisch). Ein Beispiel für eine Schnittstelle: **Lesen** (ermittle Daten einer Ressource), **Schreiben** (ändere Ressource), **Erzeugen**, **Löschen** (vgl. Unix-Dateischnittstelle, bzw. HTTP Methoden). Möglichkeiten zur Optimierung ergeben sich aus der Eigenschaft, dass die Leseoperation unveränderlich (safe) ist und die Schreiboperation idempotent (bei geschickter Wahl der Parameter).

selbstbeschreibende Nachrichten

Nachrichten enthalten Ressourcenrepräsentationen

- eindeutige Identifikation für Identifizierung der Ressource
- Typisierung durch Repräsentation
    - versch. Repräsentationen möglich (z.B. XML-Repräsentation)
    - Aushandlung der verwendeten Repräsentation möglich
    - Die Repräsentation muss Links unterstützen, also Verweise auf andere Ressourcen über deren Identifikatoren.

Hyperlinks zu anderen Ressourcen

Navigation über verlinkte Ressourcen. RESTful Applications folgen Links zu anderen Ressourcen anstatt Operationen aufzurufen.

zustandslose Interaktion

- Möglichst wenig Zustand im Server (nicht *unbedingt* zustandslos), Zustand liegt in Ressourcen
- Zustand im Client (alles, was nicht in Ressourcen gepackt werden kann)

Problem: Clients können ihren Zustand beliebig verfälschen. Daher ist es eventuell nötig den Zustand doch auf Server zu halten (in Form einer Ressource).

SOA und REST

Service Oriented Architectures können RESTful sein: Services zum Zugriff auf Ressourcen. *Einfachste* Implementierung mit HTTP als Anwendungsprotokoll (einfacher Satz von Operationen steht bereits zur Verfügung und könnten im Bedarfsfall erweitert werden (falls auf eine Großzahl von Ressourcen anwendbar)). Typische Operationen sind angelehnt an CRUD (Create, Read, Update, Delete) und für die allermeisten Ressourcen ausreichend.

Repräsentation von Ressourcen mittels

- XML (anwendungsabhängige Schemata) oder
- JSON (JavaScript Object Notation) - ebenfalls weit verbreitet, leicht in JavaScript einlesbar, {Des,S}erializer für viele Sprachen erhältlich oder
- SOAP - Repräsentation im SOAP-Body, erlaubt Beschreibung des Services in WSDL

Beispiel: WWW

WWW ist Paradebeispiel für REST, es skaliert weltweit.

- Ressourcen: Web Seiten und Dateien beliebiger Art
- Identifikation
    - erfolgt über URI
    - Verarbeitung allerdings nicht trivial: komplizierte Escape- und Verarbeitungsregeln (z.B. bei Benutzung verschiedener Zeichensätze).
- einheitliche Schnittstelle (HTTP-Protokoll zum Zugriff auf Ressourcen)
    - **GET**, **HEAD** sicher (safe), da Ressourcen nicht verändert werden; ermitteln Repräsentation von Ressourcen; Content-Negotiation erlaubt Auswahl geeigneter Repräsentationen (Sprachenauswahl oder Formatauswahl (HTML, XML, PDF))
    - **PUT**, **DELETE** idempotent, verändern/löschen Ressourcen
    - **POST** erlaubt Implementierung vieler Operationen
- HTTP als Anwendungsprotokoll in RESTful Services (hier KEIN Transportprotokoll)
    - es existieren aussagekräftige Fehlercodes
    - Caching von Repräsentationen (ausgefeilte Cache-Kontrolle durch HTTP-Header, Caches im Client und in Proxies möglich)

Beispiel: Doodle

Eine Beispielanwendung ist der Doodle-Poll. Ressourcen sind Umfrage, Optionen und Wahl. Sie werden durch URIs identifiziert und alle Funktionen sind auf CRUD-Operationen abgebildet.

Virtualisierung

Unter Plattformvirtualisierung versteht man die Virtualisierung eines Basissystems zum Betrieb eines gegebenen Betriebs- oder Laufzeitsystems. Multiplexing des Basissystems zum (eventuell gleichzeitigen) Betrieb mehrerer Betriebs- oder Laufzeitsysteme ist möglich. Die virtuelle Plattform wird Virtual Machine (VM) genannt, die Infrastruktur zur Virtualisierung VM Monitor (VMM) oder Hypervisor.

Bedingungen

Eine virtuelle Maschine ist ein effizientes, isoliertes Duplikat einer realen Maschine, wenn die folgenden drei Bedingungen (Anforderungen) gelten.

Äquivalenz

- Programme verhalten sich äquivalent zur direkten Ausführung
- geringere Ressourcen-Verfügbarkeit und geändertes zeitliches Verhalten ausgenommen

Isolation

- effektive Isolation von VMs, die auf einer realen Plattform laufen
- VMM hat komplette Kontrolle über die Hardwareressourcen
- VMs können nicht auf Ressourcen zugreifen, die ihnen nicht zugewiesen wurden
- VMM kann Kontrolle über zugewiesene Ressourcen wieder entziehen

Effizienz

Mehrheit der Instruktionen an den virtuellen Prozessor sollten direkt (ohne Intervention des VMMs) auf dem realen Prozessor ausgeführt werden.
Effizienz-Anforderung ist nicht zwingend (je nach *Zweck* der Virtualisierung)

Kritische Punkte bei der Virtualisierung

Ausführung durch die CPU

VM *emuliert* CPU / stellt virtuelle CPU zur Verfügung, jedoch könnte der Prozessor in homogenen Umgebungen oft auch *direkt* verwendet werden.

Memory Management Unit (MMU)

Bei Zugriff auf Speicher mittels der MMU muss die VM den virtuellen Speicher emulieren.

Interrupts / Traps

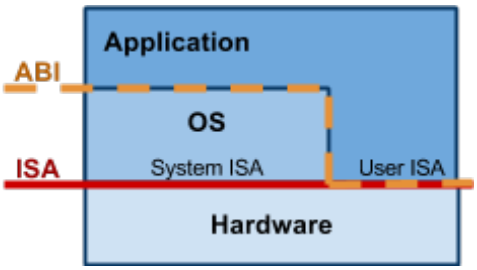
Beim Zugriff auf Geräte werden Interrupts verwendet. VM muss diese an Geräte weiterleiten. Auch die Reaktion von Endgeräten (I/O) erfolgt über Interrupts. VM muss sie an System weiterleiten. Der Zugriff auf Betriebssystem-Elemente erfolgt unter Verwendung von Traps (Software Interrupts). VM muss Traps an die entsprechende Stelle weiterleiten und Verwaltung von Interrupt-Tabellen virtuell gestalten.

Exceptions

Auch Exceptions müssen behandelt werden können, z.B. die Division durch Null.

Ebenen der Virtualisierung

Virtualisierung kann auf verschiedenen Ebenen stattfinden.



Instruction Set Interface (ISA)

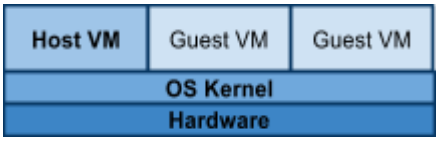
Schnittstelle der Hardware zum Betriebssystem und zu den Anwendungen. *System-Virtuelle* Maschinen virtualisieren das ISA.

- Vollständige Virtualisierung (VMM als Schicht zwischen Hardware und OS/Applications)
    - keine Anpassung des Gastsystems nötig
    - fremde Hardware wird unterstützt (emuliert ISA auf *fremder* Hardware-Plattform)
    - Beispiele: VMware Workstation, QEMU, Bochs
- Hybrid-Betrieb (z.B. Linux läuft nativ und Windows innerhalb einer VM)
- Hardware-basierte Virtualisierung
    - wenn gleiche ISA virtualisiert wird kann zur Optimierung ISA direkt benutzt werden (VMware, Linux/Windows auf x86)
    - Voraussetzung: Hardware unterstützt Virtualisierung, bzw. Schutz gewisser Einheiten durch VM (Intel VMM/VMX)
- Paravirtualisierung
    - Trennung von Host- und Gastbetriebssystem
    - Gastsystem muss definierte Schnittstelle implementieren, was die Kontrolle erleichtert. Es kann auch der direkte Zugriff auf Hardware verhindert werden.

Application Binary Interface (ABI)

Schnittstelle zum Betriebssystem, die Applikationen bzw. Prozesse nutzen. *Prozess-Virtuelle* Maschinen beschränken sich auf die Virtualisierung der ABI. Es wird eine unifizierte Laufzeitumgebung unter verschiedenen Betriebssystemen zur Verfügung gestellt. Beispiele dafür sind Wine, High Level Language Virtual Machines und Betriebssystem-Virtualisierung.

- Betriebssystem-Virtualisierung
    - Verwendung von mehreren Betriebssystem-Instanzen mit gleichem Kernel zur Bereitstellung verschiedener Dienste



    - gemeinsame Nutzung eines Kernels: Ressourcen sparen, hohe Effizienz, dennoch Isolation
    - Einsatz von Mechanismen zur Umsetzung von Ressource- und Sicherheitscontainern für Isolation
    - Beispiele: Linux VServer, OpenVZ, Solaris Zones, FreeBSD Jail
    - *effiziente, leichtgewichtige Methode zur Virtualisierung in homogenen Umgebungen*
- High Level Language (HLL) VMs
    - virtualisieren keine reale Maschine (normalerweise)
    - Entwurf erfolgt gemeinsam mit der Applikationsumgebung für die HLL
    - Vorteile: Portabilität, zugeschnitten auf HLL
    - Beispiele: Java VM und .NET CLI

Beispiel: Linux VServer (Betriebssystem-Virtualisierung)

Linux VServer nutzt und erweitert existierende Ansätze zur Isolation auf Betriebssystem-Ebene und wird verwendet im Kontext von kommerziellen Anbietern. Es gibt eine dedizierte VM für Management des Gesamtsystems. Alle Betriebssystem-Instanzen nutzen den gleichen Kernel. Bereits bestehende Infrastruktur die benutzt und erweitert wird sind Linux Capabilities (Rechteverteilung), Hard-/Soft-Limits für Prozesse (`ulimit`), Dateiattribute und die Möglichkeit zur Änderung des Root-Dateisystems mit Hilfe von `chroot`.

Isolation und Beschränkung von Ressourcennutzung

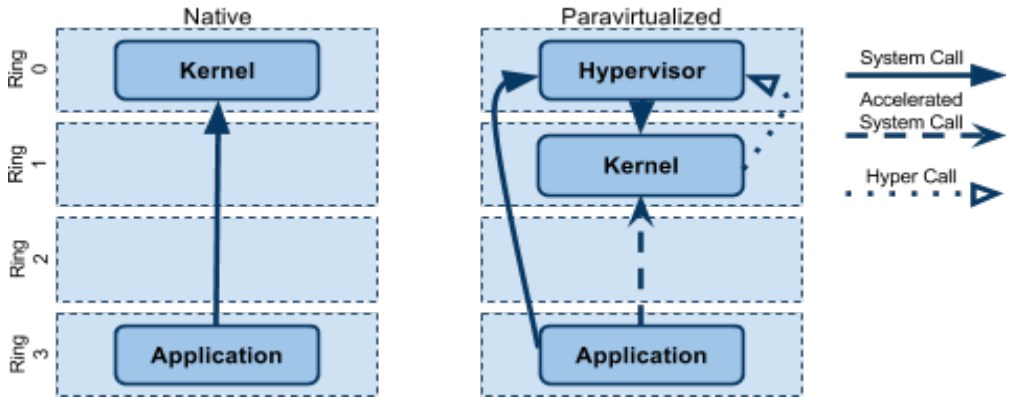
- **CPU-Verbrauch**: Verwendung von *Token Bucket Filters* (TBF), wobei Buckets individuell gefüllt werden. Pro Time-Tick wird der Bucket der aktiven VM reduziert. Ist er leer, wird sie angehalten bist wieder Mindestfüllstand erreicht ist.
- **I/O-Nutzung**: Verwendung eines *Hierarchical Token Bucket* (HTB) Ansatzes. Ermöglicht Einstellung einer reservierten Mindestdatenrate (reserved rate) und einer optionalen zusätzlichen Datenrate (share).
- **Speicher**: Limitierung der Festplattennutzung (Limit für Anzahl an Blocks/Inodes) und Limitierung des Arbeitsspeichers (strikte Begrenzung nicht unbedingt sinnvoll...)

Isolation und Beschränkung für Sicherheitszwecke

- **Isolation von Prozessen**: globaler Namensraum für alle Prozesse, daher Filter nötig, um Prozesse verschiedener VMs zu trennen. Kernel-Strukturen werden hierfür erweitert, um Abbildung auf Namensraum einzelner VMs zu ermöglichen (z.B. nötig für `init`-Prozess).
- **Isolation auf Netzwerkebene**: VMs teilen sich die Netzwerkinfrastruktur. Prozesse innerhalb einer VM werden auf bestimmte Netzwerk-Adressen beschränkt und Pakete werden markiert.
- **Isolation auf Dateisystemebene**: `chroot` ist nicht sicher, daher werden Verzeichnisse mit einem Flag markiert und es wird die Überschreitung dieser Markierung verhindert.
- **Verwendung von Linux Capabilities**

Beispiel: Xen (System-Virtualisierung)

Xen ist ein VMM für die x86-Architektur. Es sollte eine sehr kleine und effiziente VMM mittels Paravirtualisierung entwickelt werden. Leistungseinbußen nur SEHR gering. Durch Hypercalls und Split-Device Driver ist eine einfache und effiziente Virtualisierung verschiedener Betriebssysteme möglich. Xen wird im Kontext von IaaS-Clouds verwendet und konkurriert mit KVM.
Es wird getrennt zwischen Hypervisor, Betriebssystem und Anwendung.



Konsequenzen der Verlagerung des Kernels in Ring 1

Kernel kann keine privilegierten Instruktionen mehr ausführen. Diese werden durch *Hypercalls* an den VMM abgebildet. System Calls indiziert durch `80h`-Interrupt (System Call Interrupt) landen nicht im Kernel, sondern in der Interrupt-Behandlungsroutine des Hypervisors. Der Hypervisor leitet System Calls an Kernel weiter. Nachteil: zusätzlicher Kontextwechsel, daher als Alternative direkte System Calls, z.B. durch Modifikation der `libc`.

Hypercalls

- Verarbeitung ähnlich zur Behandlung von System Calls
- Hypercall = Auslösen eines Interrupts, bzw. Aufruf einer speziellen Call-Adresse
- Sprung in (privilegierten) Hypervisor zur Interrupt-Behandlungsroutine → Verarbeitung des Calls → Rückkehr in die Anwendung

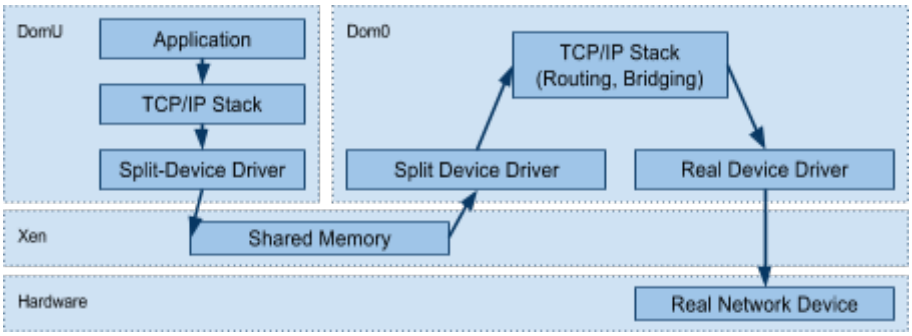
privilegierte Domäne: Dom0

- wird direkt nach Booten des Hypervisors gestartet
- übernimmt Verwaltung von Geräten (daher besitzt Xen keine Netzwerktreiber o.Ä.)
- multiplext die Hardware für alle unprivilegierten Domänen (DomUs)
    - nötig, da Geräte normalerweise nicht für direkten Zugriff verschiedener VMs ausgerichtet sind
    - Einsatz von *Split-Device Drivers* (siehe unten)
- vermittelt privilegierte administrative Operationen an Hypervisor (z.B. Erzeugung neuer VM)
- zuständig für Betrieb wichtiger Dienste (Verwaltungsschnittstelle zum Hypervisor und Verwaltung der systemweiten Registry)

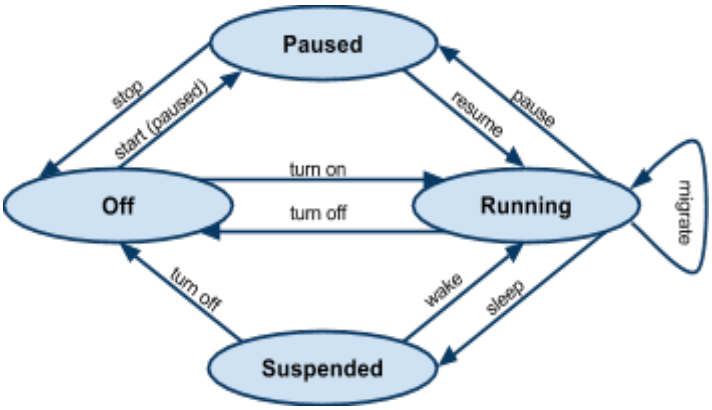
Split-Device Driver

- vermittelt zwischen VM und realer Hardware in Dom0
- Kommunikation mittels Shared Memory (Dom0 veröffentlicht gemeinsamen Speicher - vergleichbar mit Linux `procfs` - den jede VM einsehen kann und in dem jede VM einen eigenen Namensraum besitzt)

- Unabhängig von realer Netzwerkkarte immer der gleiche Split-Device Driver



Lebenszyklus einer VM



Infrastructure as a Service (IaaS)

Bereitstellung von Hardware-nahen Ressourcen in Form von virtuellen Maschinen (VMs) und Datenspeicher, entfernt zugreifbar über eine einfache Schnittstelle

- **“Pay as you go”** - abgerechnet werden nur verbrauchte Ressourcen
- Elastische Nutzung und nahezu unbegrenzte Skalierbarkeit

Status Quo

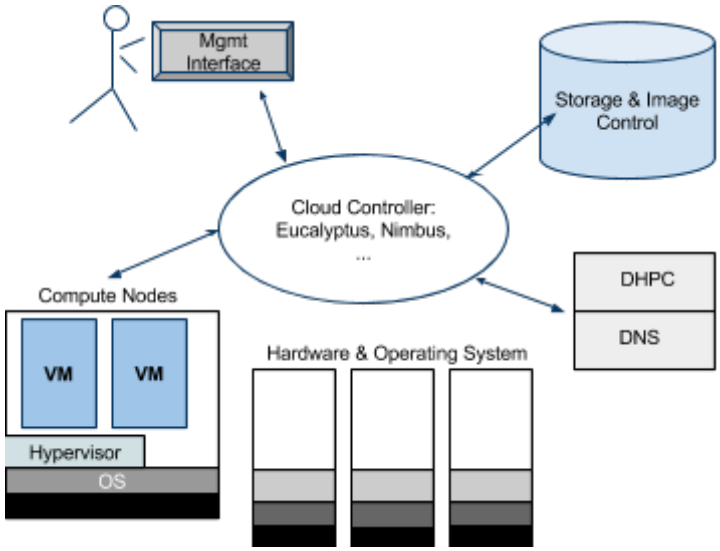
IaaS kann aktuell als wichtigste bzw. grundlegendste Form von Cloud Computing betrachtet werden
Entstehung von zahlreichen freien IaaS-Plattformen mit Amazon EC2 als erfolgreichen Vorgänger:

- Eucalyptus
- OpenNebula
- OpenStack
- Nimbus

Erschwerter Plattformwechsel aufgrund von unterschiedlichen Image-Formaten und Virtualisierung sowie verschieden erweiterten APIs (z.B. für dynamische Skalierung).
Erweiterung einer IaaS-Plattform noch problematischer, weil kaum Standardschnittstellen etabliert sind (Ausnahme: `libvirt`).

Generische Architektur einer IaaS-Cloud

1. Hardware und Betriebssystem
2. Netzwerk und Netzwerkdienste (z.B. DHCP, DNS)
3. Virtualisierung
4. Datenspeicher und Image-Verwaltung
5. Managementschnittstelle für Administratoren und Benutzer
6. Cloud Controller - Management der Ressourcen



Netzwerk und Netzwerkdienste

- Verwaltung und Einbindung der realen sowie virtuellen Infrastruktur
- Zuordnung einer eindeutigen virtuellen MAC pro VM
- Vermittlung des Datenverkehrs zwischen realer Maschine und VMs (bridge)
- Automatische Einbindung mittels DHCP und DNS (Zuordnung von IP-Adresse und Hostnamen)
- Evtl. Etablierung privater Netzwerke usw.

Umsetzung
Cloud Controller muss bei Erzeugung einer VM die Virtualisierungsinfrastruktur konfigurieren und entsprechende Informationen an DHCP- und DNS-Server weitergeben

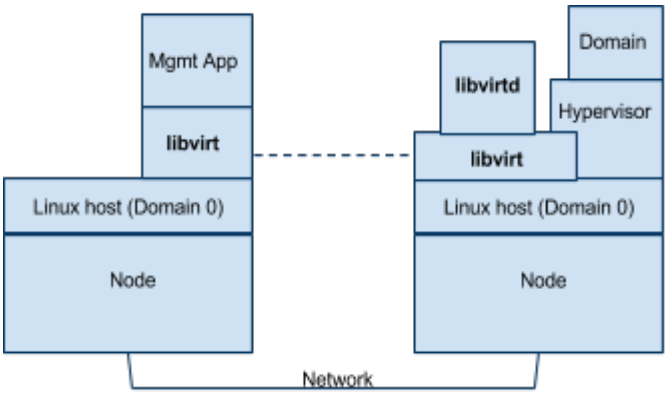
Virtualisierung

- Details im vorherigen Kapitel: Xen, Linux VServer, KVM, ...

- `libvirt` ist häufiges Werkzeug im Open-Source-Bereich um von einer konkreten Virtualisierungslösung zu abstrahieren

Umsetzung - Libvirt

- bietet einheitliche Schnittstelle zum Verwalten von unterschiedlichen Virtualisierungslösungen
- einfache C-Schnittstelle und Kommandozeilensupport für Starten, Stoppen, Migrieren, ... von VMs
- `libvirtd` nimmt Kommandos entgegen → lokales sowie entferntes VM Management



Datenspeicher und Image-Verwaltung

für eine schnelle Erzeugung und De-/Installation von VM-Images (Images müssen konfiguriert und evtl. um Software ergänzt werden)

Umsetzung: Repository für Images

- Vorlagen von Images, die noch in ein lauffähriges System umgewandelt werden müssen
- Hinzufügen einer SWAP-Partition oder Anpassung der Partitionsgröße

Managementschnittstelle für Benutzer

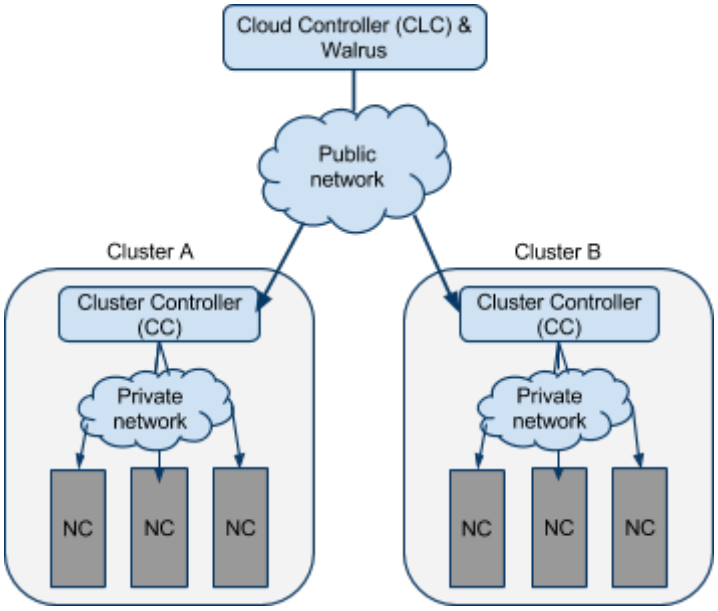
- meistens grafische Benutzeroberfläche für Anwender
- Anfordern neuer VMs
- Benutzerspezifische Konfiguration: z.B. EC2-Größenklassen für VMs oder Ausführungsort
- Verwaltung von Credentials für sicheren VM Zugriff

Cloud Controller

- Zentrale Komponente einer IaaS-Infrastruktur
- Umsetzung der Benutzeranforderungen für die Erzeugung von VM-Images und deren Installation
- Um-/Verteilen von VMs auf realen Maschinen

Beispiel: Eucalyptus

- Bereitstellung einer offenen IaaS-Infrastruktur
    - Basis für Forschung, die mit kommerziellen Produkten (EC2) so nicht möglich ist
    - frei verfügbare Basis für Private-Clouds (sogar Bestandteil von Ubuntu Linux)
- traditionelles hierarchisches Design vergleichbar mit Infrastrukturen aus Grid Computing
- Schnittstellen ähnlich bzw. identisch zu Amazon EC2
    - erleichterte Portierung von Anwendungen
    - existierende EC2 Werkzeuge können verwendet werden
    - Web Service Schnittstelle für alle tragenden Komponenten
- basierend auf `libvirt` wird aktuell Xen und KVM unterstützt



Node Controller (NC)

- verwaltet *Ressourcen eines Rechners*
- Verbindungsglied zum Cluster Controller
- Monitoring
    - installierte Software
    - verfügbare Ressourcen (z.B. CPU Anzahl/Last, Arbeitsspeicher)
    - Anzahl und Eigenschaften der laufenden VMs
    - Informationen werden auf Anfrage des CLC ermittelt (z.B. `describeInstances`)
- Verwaltung von VMs
    - authentifizierter Besitzer einer VM oder der Cloud Admin kann eine VM beenden
- Überprüfung ob genügend Ressourcen auf der Maschine verfügbar sind bevor VM gestartet werden kann (zuvor muss das Image installiert worden sein)

Cluster Controller

- kombiniert die Informationen eines Clusters, die z.B. mit `describeInstances` abrufbar sind
- vermittelt Aufrufe an einzelne NCs
- Verteilung von VMs
    - Anfrage bei den NCs nach freien Ressourcen (`describeResources`)
    - First-Fit-Strategie: Platzierung der VM auf dem ersten NC mit ausreichend Ressourcen

Virtual Network Overlay

- Anwender haben typischerweise mehrere VMs, die gemeinsam verwaltet werden müssen
- dazu muss mindestens eine VM von Extern zugreifbar sein
- VMs verschiedener Benutzer müssen voneinander isoliert sein
    - Anwender haben root-Privilegien auf ihrer VM und auf ihrer Netzwerkschnittstelle
- Verschiedene Konfigurationsoptionen von Netzwerkschnittstellen unterstützt
    - direkte Verbindung der virtuellen Netzwerkschnittstellen mit dem physikalischen Endgerät → VMs können exakt wie reale Maschinen behandelt werden
    - Zuweisung eines MAC-IP-Tupel durch den NC
      → feste Menge von IPs ist an laufende Instanzen gebunden
- Virtual Network Overlay ist ein durch Eucalyptus verwaltetes Netzwerk
    - pro Anwender wird ein eigenes VLAN erstellt
    - entsprechende Firewall-Regeln sorgen für Isolation

Walrus (Datenspeicher und Image-Verwaltung)

- implementiert die Amazon-S3-Schnittstelle
    - einfache Key-Value Schnittstelle für Dateien
    - bei konkurrierenden Schreiboperationen wird nur aktuellster Aufruf durchgeführt
    - Values werden nur komplett geschrieben
- Zugriff sowohl von innerhalb als auch von außerhalb der Cloud
- bildet auch Repository für VM-Images
    - Images werden als gesplittete, verschlüsselte Dateien abgelegt
    - bei Anforderung eines Images vom NC müssen einzelne Dateien erst entschlüsselt und auf ihre Integrität geprüft werden → abschließend Transfer zum NC

Cloud Controller Zuständigkeitein

- *Ressource Services*: Allokation von Ressourcen und ihre Konfiguration
- *Data Services*: Verwaltung von Anwender- und Systemdaten
  → ermöglicht flexible Konfiguration von Allokationsstrategien
- *Interface Services*: Schnittstellen für Anwender

Beispiel: Amazon S3

"storage for the internet ... designed to make web-scale computing easier"

- Bereitstellung von flexibel skalierbarem Datenspeicher
- zugreifbar über eine einfache und plattformunabhängige Schnittstelle
- Schnittstellen sind **SOAP** und **REST**
- bildet aktuell Standard bzw. Vorbild für andere Systeme
- leichte Umsetzung auf Serverseite aufgrund von eventual consistency
- erweiterte Konsistenzgarantien müssen auf Anwendungsseite realisiert werden

Basiskonzepte

- Buckets: konfigurierbarer Container für Datenobjekte
  - Wurzel eines Namensraums in S3
  - assoziiert mit einem Benutzer
  - Einheit für Kostenabrechnung und Statistik
  - flexible Zugriffsrechtevergabe für Teile des Namensraums möglich
  - Bsp.: `http://<bucketname>.s3.amazonaws.com`
- Objects: eindeutig identifizierbares Datenobjekt
  - besteht aus Nutzdaten und Datenbeschreibung (z.B. standard HTTP Attribute)
  - unterstützte Operationen: Erzeugen, Lesen, Schreiben, Löschen
- Keys: eindeutiger Name eines Objekts innerhalb eines Buckets
  - optionale Versionierungsinformation
  - Bsp.: `http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl`
- Region
  - Zuweisung des Lagerungsort der Daten trotz globaler Erreichbarkeit
  - erlaubt Optimierung auf geographisches Nutzungsprofil sowie teilweise Einhaltung von rechtlichen Auflagen (medizinische Daten)

Konsistenzmodell - Eventual Consistency

- starke Konsistenz
  - nach einer Änderung wird garantiert, dass alle Prozesse denselben Wert sehen
- schwache Konsistenz
  - nach einer Änderung wird nicht garantiert, dass immer derselbe Wert geliefert wird
    - ob und wann Änderung sichtbar werden hängt von Nebenbedingungen ab
  - **Eventual consistency** ist eine Form der schwachen Konsistenz
    - Änderungen werden erst nach einer gewissen Zeit sichtbar
    - Vorteil: Schneller Zugriff mit hoher Bandbreite (Replikation von Daten)
    - Nachteil: Lesen von aktuell schon *veralteten Daten* ist möglich

Authentifizierung

- Anfragen sind **anonym** oder von einem **authentifizierten Nutzer**
- registrierte Amazon Benutzer haben
  - einen mit Amazon gemeinsamen geheimen Schlüssel
  - einen mit dem Benutzer assoziierten öffentlichen Schlüssel
- Authentifizierung und Überprüfung von Datenintegrität über die REST Schnittstelle mittels eines Hash-based Message Authentication Code (HMAC)
  - URL der zugegriffenen Ressource zusammen mit dem geheimen Schlüssel wird durch eine Hashfunktion auf einen Hashschlüssel abgebildet

    - URL wird zusammen mit Hashschlüssel und öffentlichem Schlüssel übertragen
    - Empfängerseite ermittelt mit dem öffentlichen Schlüssel den (vermeintlich) gemeinsamen geheimen Schlüssel und kann damit so den Hashschlüssel überprüfen

Zugriffsrechte

- auf Ebene von Buckets oder Objects möglich
- mittels Access Control Lists (ACL)

(Rechte authentifizierter Nutzer: READ, WRITE, READ_ACP, FULL_CONTROL)

Datenmanagement in Clouds

Ein allgemeiner Trend ist zu verzeichnen: die extreme Zunahme der zu verarbeitenden Daten. Hieraus ergibt sich die Problematik, Daten trotzdem effizient und skalierbar zu verarbeiten. Lösungsansätze sind neue Konzepte / Algorithmen für *verteiltes* Datenmanagement. Beispiele hierfür sind Programmiermodelle wie MapReduce / Hadoop, verteilte, einfache Datenbank-ähnliche Systeme / Key-Value Stores wie Dynamo oder BigTable und verteilte Dateisysteme wie GoogleFS und HadoopFS.

MapReduce

Motivation zur Entwicklung von MapReduce war, dass die Programmierung von datenlastigen Anwendungen in verteilten Systemen aufwendig ist. Man muss sich um Kommunikation und Koordination kümmern, um die Wiederherstellung nach Ausfall eines Rechners, man benötigt Statusdaten zur Systemüberwachung, auch die Fehlersuche und Optimierungsmaßnahmen gestalten sich schwierig. Und vor allem: Diese Programmierarbeit muss für jede Anwendung wiederholt werden. Daher der Lösungsansatz: eine einfache und strukturierte Programmiermethode unterstützt durch eine Laufzeitumgebung für verteilte Systeme. MapReduce ist einfach zu verwenden und es kann eine Vielzahl an Problemstellungen gelöst werden (Sortieren, diverse Google Projekte wie z.B. das Indexing-System bei der Websuche).

Benutzerdefinierte Funktionen

- `map`: Transformation einer Menge von Datensätzen in eine Zwischendarstellung
- `reduce`: Reduktion der Zwischendarstellung auf das Endergebnis
- Parametrisierung (z.B. Festlegung des Grads der Nebenläufigkeit)
- MapReduce-Aufruf: Bereitstellung des äußeren Programmablaufs (enthält z.B. Management der Ein- und Ausgaben)

durch das Framework definierte Funktionen

- Parallelisierung
- Fehlerbehandlung
- Aufteilung der Daten
- Lastverteilung

einfaches Beispiel

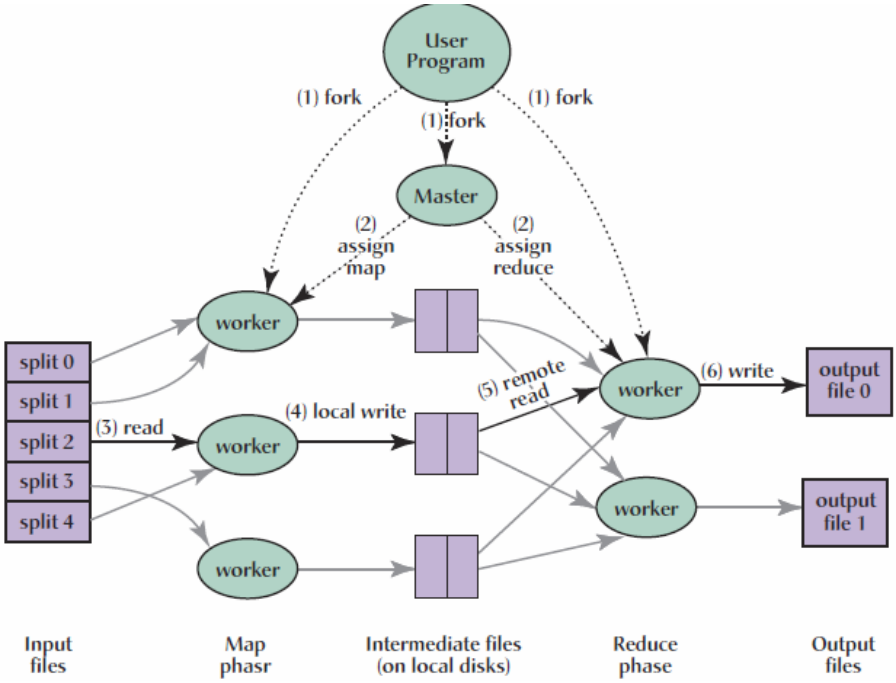
Aufgabe: Buchstaben zählen; Input: “`acaabb`”

```
map(): a,1; c,1; a,1; a,1; b,1; b,1 --> group(): a,1,1,1; b,1,1; c,1 --> reduce(): a,3; b,2; c,1
```

Ablaufmodell

1. Vorbereitungsphase: Aufteilung der Eingabedaten, Erzeugung von Master und Workern
2. Vergabe von Map-/Reduce-Tasks an freie Worker
3. Verarbeitung der Eingaben
    a. Worker liest Inhalt aus ihm zugewiesenen Teilbereichen der Eingabedaten
    b. Aufteilung des Inputs in Schlüssel-Wert Paare und Weitergabe an `map()`
    c. Erzeugung und Pufferung (im Speicher) der Zwischendarstellung
4. Sicherung und Vermittlung der Zwischenergebnisse
    a. gepufferte Wertepaare werden periodisch auf lokale Festplatte geschrieben
    b. Übertragung der Adressen der Daten an Master

5. Gruppierung der Daten
    a. Master übermittelt Speicherbereiche an Worker
    b. Worker liest Zwischenergebnisse aus lokalem Speicher und sortiert sie nach Schlüssel
6. Ermittlung der Endergebnisse
    a. Worker iteriert über Zwischenergebnisse, gibt Key und zugehörige Values an `reduce()`
    b. Ausgabe von `reduce()` wird zu Ausgabedatei hinzugefügt



Fehlertoleranz

- Watchdog: falls keine Antwort auf Ping wird Worker als ausgefallen betrachtet und alle laufenden Tasks dieses Workers zurückgesetzt. Fertiggestellte Map-Tasks müssen erneut ausgeführt werden (lokaler Speicher benötigt), fertiggestellte Reduce-Tasks nicht.
- Periodisches Sichern der Datenstrukturen des Masters: Task-Zustände, Identität der Worker, Adresse der Zwischenergebnisse. Bei Ausfall wird Kopie vom Master mit zuletzt gesichertem Zustand gestartet

Optimierungen

- Datenlokalität: Daten werden durch GoogleFS mehrfach repliziert und Master kennt Speicherorte der Eingabedaten und positioniert Worker dementsprechend in der Nähe
- Backup-Tasks: Reduktion der Gesamtlaufzeit durch redundante Ausführung der letzten Tasks
- frühere Zusammenfassung von Teilergebnissen mittels Combiner
- Differenzierende Datenpartitionierung: Aufteilung der Daten auf Reduce-Tasks üblicherweise durch modulo-Operation zwar häufig ausbalanciert, aber nicht unbedingt optimal

Dynamo

Dynamo ist ein verteilter, skalierbarer Key-Value Store für kleine Datensätze (z.B. Warenkörbe). Hochverfügbarkeit ist wichtigstes Ziel. *Eventual Consistency* ist eine Eigenschaft von Dynamo, die Schreibzugriffe immer ermöglichen soll und in der Konsistenzzusicherungen zugunsten von Verfügbarkeit reduziert wurden. Ausgleich schafft Versionierung. Eine weitere Eigenschaft ist die P2P-artige Datenverteilung, d.h. es gibt keine Master-Knoten (alle Knoten haben exakt die selbe Funktionalität), unter Zuhilfenahme von *consistent hashing*.

Verarbeitung einer Client-Anfrage bei Amazon

1. Seitenerstellungs-Komponenten (Page Rendering Components) nehmen Anfrage entgegen
2. diese stellen Anfragen an Sammel-Dienste (Aggregator-Services)
3. diese kombinieren Informationen angeschlossener Datenbanken und
4. geben die Daten an Page Rendering Components zurück
5. Page Rendering Components erstellen dann die Website und
6. liefern sie an den Client

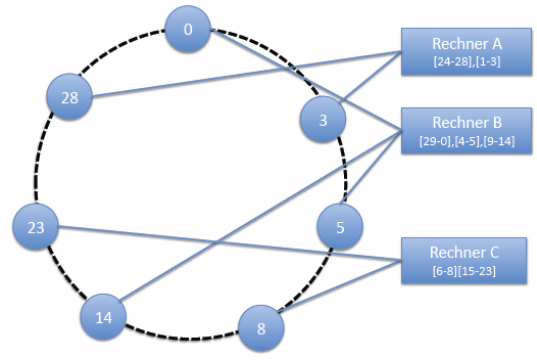
Das ganze läuft nicht in einer Public Cloud, d.h. die komplette Infrastruktur ist vertrauenswürdig und somit sind keine Mechanismen zur Authentifizierung und Autorisierung nötig!

Dienstschnittstelle / Anfragen

Eine Anfrage kann an beliebigen Dynamo-Knoten gesendet werden, welcher sie dann an den verantwortlichen Knoten weiterleitet. Es gibt zwei verschiedene Anfragen:

- **Lesen**: `get(byte[] key)`
  - liefert Liste von `(Object, Context)`-Paaren (wegen Versionierung)
  - entspricht Primary-Key Zugriff (DBMS), also keine komplexe Anfrage
- **Schreiben**: `put(byte[] key, Context c, byte[] object)`
  - `Context` hält Versionsnummer (vorheriges `get`)
  - Erhöhen der Versionsnummer bei lokalem Schreiben des Objekts
  - Asynchrone Aktualisierung der Replikate: Konsistenzprobleme möglich

Basiskonzept



Der Schlüsselraum wird ringförmig auf die vorhandenen Knoten abgebildet; mehrere IDs pro Knoten, möglichst gleichmäßig auf alle Knoten verteilt.
Abbildung der Daten auf Knoten erfolgt mittels Schlüssel. Die Daten werden dem Knoten mit der unmittelbar im Ring folgenden ID übergeben, für Fehlertoleranz / Ausfallsicherheit den *N* Knoten aus einer Liste von präferierten Nachfolgeknoten. Die *N* Knoten werden so gewählt, dass die virtuellen Knoten möglichst auf verschiedenen physikalischen Rechnern und eventuell in verschiedenen Datenzentren liegen.

Behandlung von neuen Knoten

Über *wohlbekannte* Knoten (Seed Nodes) wird der Kontakt zum Netzwerk aufgenommen. Es werden virtuelle IDs erzeugt und benachbarte Knoten von der Teilnahme benachrichtigt. Der neue Knoten fordert letztendlich Daten entsprechend seiner Verantwortung an.

Behandlung von kurzfristigen Knotenausfällen (Hinted Handoff)

Falls ein Knoten der ersten *N* der Vorzugsliste ausfällt, wird ein Ausweichknoten gewählt, welcher die entsprechenden Daten in einer separaten Datenbank speichert, zusammen mit der Information, zu welchem Knoten die Daten tatsächlich gehören. Wenn der ausgefallene Knoten wieder anläuft, werden die Daten an diesen zurückgegeben (periodisches Abfragen der Datenbank).

Behandlung von inkonsistenten Zuständen

Da *Eventual Consistency* verwendet wird können inkonsistente Zustände auftreten. Theoretisch kann die Auflösung beim Lesen oder beim Schreiben erfolgen, doch da Dynamo als *always writable* ausgelegt ist, erfolgt sie beim Lesen. Inkonsistenzen werden durch *Vektoruhren* erkannt und entweder automatisch durch das System oder durch spezifische Behandlung auf Anwendungsebene aufgelöst.

Verwendung von Vektoruhren

Jeder Rechner besitzt eine lokale Uhr, die aus einem Vektor der Größe *N* = #Prozesse besteht. Initialisierung jedes Zeitvektors mit dem Nullvektor. Bei jeder Schreiboperation eines Prozesses P_i wird dessen Komponente in seinem Zeitvektor inkrementiert: `c_i[i]++;`
Ein Prozess P_i, der eine Nachricht empfängt, inkrementiert seine Komponente im Zeitvektor und kombiniert diesen dann komponentenweise mit dem empfangenen Zeitvektor t:

```
C_i[i]++; for each k = 1...N: C_i[k] := max(C_i[k], t[k]);
```

Dies erzeugt eine kausale Ordnung, welche eine exakte Aussage zum kausalen Zusammenhang von Ereignissen mit Hilfe des Zeitstempels ermöglicht.
In Dynamo besitzt nicht jeder Prozess, sondern jedes Objekt eine lokale Uhr, die aus einem Vektor mindestens der Größe *N* = #zu_replizierende_Knoten besteht. Die Anwendung sendet beim Schreiben ein Kontext-Objekt, welches die Vektorzeit des aktuellen Objekts dem System übermittelt. Lesende Anfragen liefern bei Inkonsistenzen mehrere Objekt- bzw. Vektorzustände, was die Erkennung / Auflösung von Konflikten ermöglicht.
Konflikte können aufgelöst werden z.B. durch Propagierung der aktuellsten Version oder bei Warenkörben: hinzugefügte Artikel dominieren entnommene.

Behandlung von Anfragen

Es werden Read/Write-Quoren verwendet, eine Menge von Knoten, die im Kollektiv eine Entscheidung treffen. Ein Quorum-Tripel: (N,R,W), W+R > N, N = #Knoten. R/W: minimale Anzahl der N Replikat-Knoten, die für erfolgreiches Read/Write übereinstimmen müssen. Werte für N, R und W sind frei konfigurierbar, daher kann man hier eine gute Lösung für Performance, Verfügbarkeit, Dauerhaftigkeit und Konsistenz einstellen. (3,2,2) ist üblich für Dynamo.
Dynamo verwendet Sloppy Quorum. Strenge Konsistenz erfordert R+W > N, W > R/2, was aber hohe Zugriffszeiten zur Folge hat. Falls W = 1 ist Schreiben immer möglich, solange mindestens ein Knoten arbeitet.

BigTable

BigTable wird zur feingranularen, verteilten Datenspeicherung eingesetzt, z.B. zur Indexierung von Websites. Es ist ein spaltenorienterter Key-Value Store, der multi-dimensional, versioniert, hochverfügbar und performant ist. Ziel ist die Verwaltung von Milliarden von Zeilen, Millionen von Spalten und Tausenden von Versionen, also große Datenmengen. BigTable soll linear zur #Knoten skalierbar sein. Open-Source Pendant zu BigTable: HBase.

Verwendungszweck

Web-Tabelle für Suchmaschine

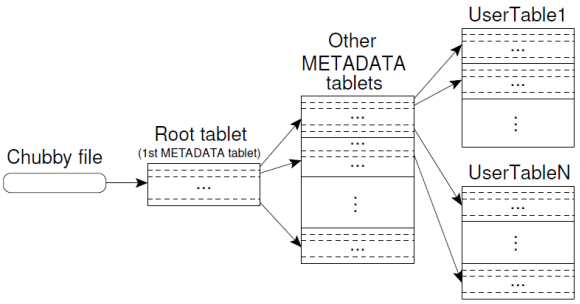
- Tabelle mit eingelesenen Websites und ihren Attributen und Inhalten
- Schlüssel: Website-URL
- wahlfreier Zugriff durch *Crawler* zum Einfügen neuer/geänderter Websites
- wahlfreier Zugriff durch Suchmaschinennutzer (temporäre Kopie von Website)
- Batch-Auswertung zum Aufbau eines Suchmaschinenindex

Software Stack

BigTable setzt auf einer Reihe von anderen Komponenten auf.

- GoogleFS zur Ablage der Daten
- Scheduler zur Verwaltung der einzelnen Teilaufgaben von BigTable
- Lock-Service bzw. Chubby zur Bestimmung eines Masters und als Ortsdienst
- MapReduce zur Manipulation von in BigTable verwalteten Daten

Datenmodell



Eine verteilte, mehrdimensionale und sortierte Abbildung mit Spalten- und Zeilenschlüssel und Zeitstempel auf Daten bestehend aus beliebigen Zeichenketten / Bytestrings.

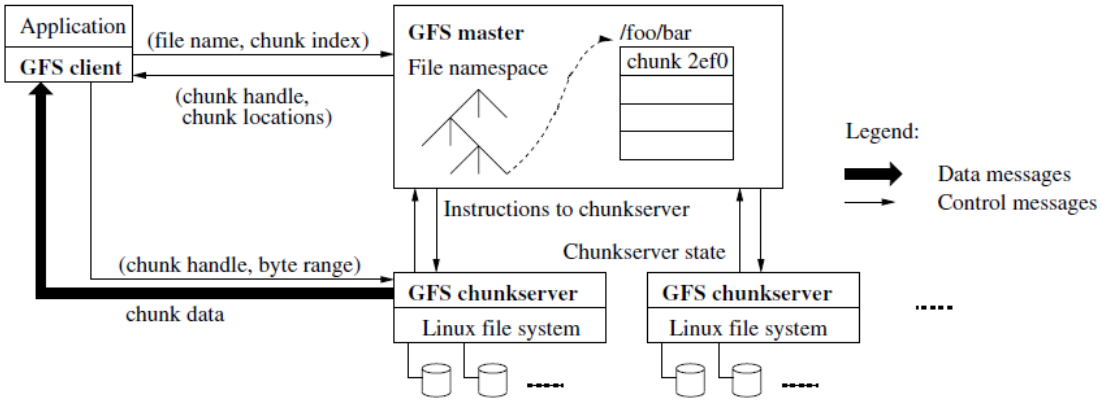
```
(row:string, column:string, time:int64) --> string
```

Nur Lese- und Schreiboperationen auf eine Zeile sind atomar. Speicherung der Daten erfolgt in lexikographischer Reihenfolge der Zeilenschlüssel.
Durch den Zeitstempel können mehrere Versionen pro Zelle existieren. Es gibt eine festgelegte maximale Versionszahl, was dazu führt, das “zu alte” Daten automatisch gelöscht werden.
Technisch umgesetzt ist das Datenmodell durch *Tablets*, eine horizontale Partitionierung von Tabellen. Es gibt mehrere Tablet-Server zur Aufnahme der Tablets (Lastbalancierung). Tablets werden in zwei gleich große Tablets gesplittet, wenn eine maximale Größe (z.B. 128 MB) erreicht wurde.
Es wird eine Bibliothek auf Anwendungsseite verwendet und Anwendungen kommunizieren direkt mit Tablet-Servern. Der Master ist zuständig für die Verwaltung der Tablet-Server, also er weist Tablets an bestimmte Server zu und integriert eventuell neue Server. Die Tablet-Server wickeln Lese- und Schreiboperationen auf den Tablets ab.

GoogleFS

GoogleFS ist ein skalierbares, fehlertolerantes verteiltes Dateisystem für sehr große Datenmengen und verwendet Standardhardware. Bei Google werden große Datenmengen sequentiell gelesen und geschrieben und Dateien von mehreren Anwendungsteilen parallel ergänzt. Traditionelle verteilte Dateisysteme dagegen sind dafür ausgerichtet, kleine Dateien wahlfrei mit geringer Latenz zu lesen und zu schreiben.

Architektur



GoogleFS nicht POSIX-konform, unterstützt aber Standardoperationen wie erzeugen, löschen, öffnen, schließen, lesen und schreiben von Dateien. Zusätzlich dazu noch anfügen (record *append*) und Sicherungspunkte. Anfügen ist im Kontext von MapReduce besonders wichtig. GoogleFS verfolgt einen Master-Ansatz, also ein einfaches Design, dass Platzierung und Verteilung von Daten basierend auf globaler Sicht ermöglicht. Fehlertoleranz durch *Shadow-Master*, der im Fehlerfall sofort einspringen kann.

- **Master-Server**
  - Verwaltung von Metainformationen: Namensinformationen, Zuordnung von Chunks (Dateneinheiten) zu Dateien, Ortsinformationen von Chunks
  - Metadaten nicht persistent, sondern im Hauptspeicher (kürzere Latenz)
  - Operationen bzgl. Ortsinformationen werden protokolliert und periodisch durch Master von den *Chunk-Servern* abgefragt
  - Funktion als Monitor: periodische Überprüfung der Chunk-Server
  - Übernahme zentraler Verwaltungsaufgaben: Kontrolle der Chunk-Replikation
  - Mechanismen zur Vermeidung eines Flaschenhalses am Master:
    - Master nicht involviert in eigentliche Schreib- und Leseoperationen
    - Clients speichern Metadaten zwischen (z.B. Ortsinformationen)
    - Clients so ausgelegt, dass Anfragen an Master kombiniert werden und Master daher mehr Metadaten gleichzeitig ausliefern kann (Batching?)
- **Chunk-Server**
  - lagern keine Dateien explizit zwischen (aber implizit durch Linux Systempuffer), da Datenbestand zu groß ist und Koordinierungsaufwand reduziert wird
  - Größe eines Chunks: 64MB (größer als typischer Dateisystemblock)
    - Vorteile: weniger Client/Server-Interaktion, weniger Metadaten
    - Nachteile: interne Fragmentierung, Chunks können sehr populär sein (Lösung: Replikation, zeitliche Verteilung von Anfragen, direkte Kommunikation zwischen Clients)
- **Client**

- lagert kurzzeitig Metadaten zwischen
- lagert **keine** Dateien zwischen (hohes Inkonsistenzrisiko)

Konsistenzmodell

Operationen auf Metadaten werden durch den Master atomar ausgeführt. Anwendungen sollen lieber Dateien ergänzen, statt zu schreiben, da dies stabiler ist. Es gibt Sicherheitspunkte. Wenn mehrere Schreiber eine Datei ergänzen, sollten IDs für Einträge und Prüfsummen verwendet werden.

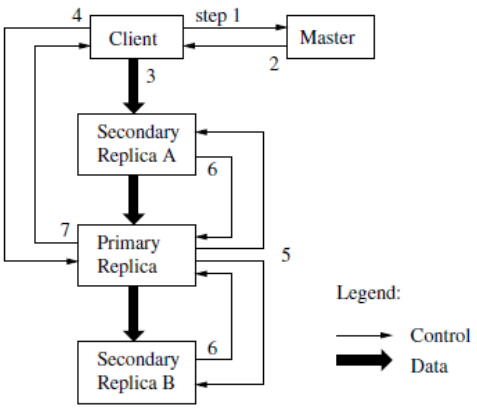
Ablauf einer Schreiboperation

1. Anwendung generiert Schreibanfrage
2. GoogleFS Client übersetzt und sendet an Master
3. Master antwortet mit *chunk handle* und (Primär- und Sekundär-) Chunk-Servern
4. Client sendet Daten zu allen diesen Chunk-Servern
5. Chunk-Server speichern Daten in Puffer zwischen
6. Client sendet Schreibbefehl zum primären Chunk-Server
7. primärer Chunk-Server ermittelt serielle Reihenfolge für Daten in Puffer und schreibt sie in den Chunk
8. anschließend sendet er die Reihenfolge an die sekundären Chunk-Server
9. diese bestätigen den Schreibbefehl an den primären Chunk-Server
10. primärer Chunk-Server berichtet Client über Erfolg (oder Misserfolg)

Ablauf eine Leseoperation

1. Anwendung generiert Leseanfrage
2. GoogleFS Client übersetzt Anfrage und sendet sie an Master
3. Master antwortet mit einem *chunk handle* und einer Liste aller Chunk-Server, die diesen Chunk speichern
4. Client wählt einen dieser Chunk-Server und sendet Anfrage (`chunk handle, byte range`)
5. Chunk-Server sendet die angeforderten Daten zum GoogleFS Client
6. dieser leitet Daten an Anwendung weiter

Datenreplikation



Jeder Chunk wird über mehrere Chunk-Server repliziert. Die primäre Zuständigkeit für einen Chunk wird über (zeitlich limitierte) *Leases* realisiert. Die Replikation wird in zwei Kontrollabläufe gegliedert, zuerst die Replikation der Nutzdaten und anschließend die Replikation der Operationen und Festlegung ihrer Reihenfolge.

- Nutzdaten werden direkt *durch* alle Replikate geleitet, von Replikat zu Replikat vermittelt
- sobald Daten versendet: Übermittlung der eigentlichen Anfrage an primäres Replikat, welches eine Id an die Anfrage vergibt und diese Id an alle weiteren Replikate vermittelt

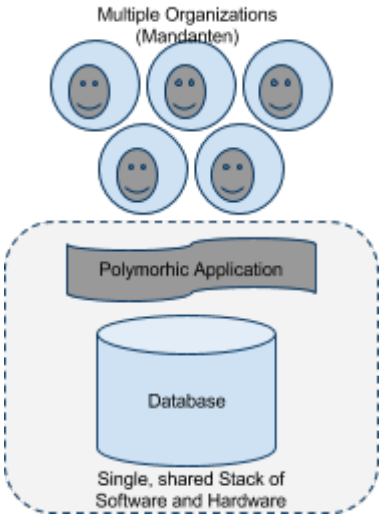
Sonstiges

- Atomares Anfügen (*record append*)
  - Initiale Schritte wie beim normalen Schreiben der Daten

    - a) Daten zu groß für letzten offenen Chunk? Primary Chunk-Server weist alle Replikate an, diesen Chunk zu schließen → Benachrichtigung des Clients, dieser versucht dann erneut zu schreiben (in anderen Chunk, von Master angefordert)
    - b) Daten passen in offenen Chunk? Operation durchführen! falls mindestens ein Replikat einen Fehler meldet, wird die Operation erneut durchgeführt. Dadurch können Daten mehrfach in eine Datei geschrieben worden und Replikate nicht identisch sein
- Sicherungspunkte
    - realisiert mittels *copy-on-write*
    - Master invalidiert existierende Leases oder wartet bis sie ablaufen
    - im Anschluss: alle Anfragen über betroffene Daten gehen über Master
    - Master weist die Erzeugung von Kopien an
- Namensraumverwaltung
    - Besonderheit gegenüber Standarddateisystemen
    - Verzicht auf traditionelle verzeichnisorientierte Verwaltung
    - Verzeichnisse/Dateien werden mittels ihrer vollständigen Pfade verwaltet
    - einfache Implementierung und gute Unterstützung für nebenläufige Operationen
    - Beispiel für traditionelle Dateisysteme: Nebenläufiges Anlegen zweier Dateien: `/x/1` und `/x/2`; beide Operationen fordern ein Leseschloss für Verzeichnis und ein Schreibschloss für Dateinamen an; jeweilige Datei kann ohne beide Schlösser nicht erzeugt werden.

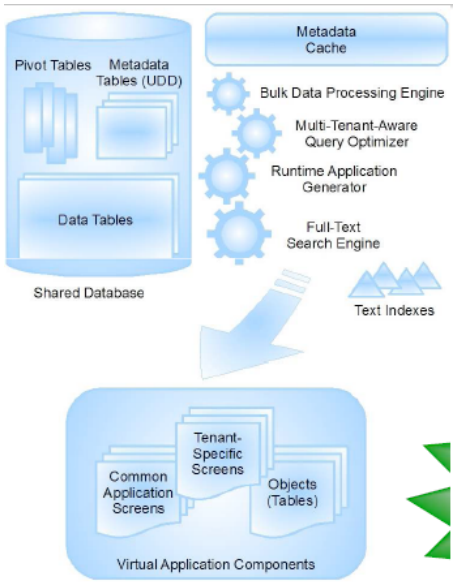
force.com

- **Platform-as-a-Service** Paradebeispiel
- Plattform für viele Kunden mit spezifischen Anforderungen
    - Mandantenfähige Betriebsplattform für datenlastige Anwendungen
    - Existierende Lösungen im DB-Bereich skalieren für wenig Benutzer mit viel Daten
- Ziel: Flexibel anpassbare Plattform, die parallel sehr viele Mandanten/Kunden unterstützt
    - Mandantenfähige Plattform, die auch zuverlässig, sicher, erweiterbar ist
- Umsetzung: Viele Kunden auf einer Virtualisierung bzw. einer "großen unterteilten DB"
    - Zuverlässige Kernplattform ergänzt durch Metadaten, welche die Anwendungen der Mandanten selbst beschreiben
    - Starke Isolationsmechanismen und Optimierung notwendig
- Ausblick: weitere ähnliche Angebote (MS Azure)



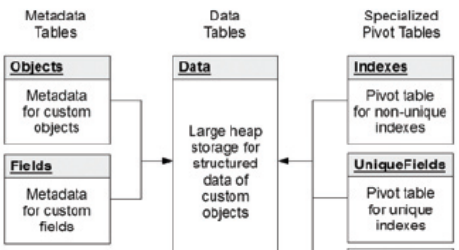
Architektur

- Universal Data Dictionary (UDD): Alle Metadaten, die Anwendungen oder Benutzer beschreiben
- Mechanismen für bessere Skalierbarkeit
    - Metadata Cache: UDD-Daten werden zwischengespeichert
    - Bulk Data Processing Engine: modifizierende Operation werden gesammelt und gemeinsam ausgeführt
    - Full-Text Search Index: Einbindung eines Dienstes, der nebenläufig die Daten indiziert
    - Multi-Tenant-Aware Query Optimizer: Optimiert Datenbankanfragen ausgehend von den Metadaten und Pivot-Tabellen



Datenorganisation

- Object Metadata Table: anwendungsspezifische Objekte (Tabellen/Einträge), identifiziert durch ObjID, OrgID (Organisation/Kunde) und ObjName
- Field Metadata Table: anwendungsspezifische Felder (Spalten/Attribute), identifiziert durch FieldID, zugeordnet durch OrgID und ObjID, beschrieben durch Name und Typ sowie optionale Feldindizierung
- Data Table
    - Ablage der Anwendungsdaten entsprechend der Definition der Metadata Tables
    - Jeder Eintrag in der Tabelle wird durch eine eindeutige GUID (OrgID + ObjID) beschrieben (Strukturfestlegung der Daten)



    - Nutzdaten werden als Zeichenketten variabler Länge (unformatiert) abgelegt (→ Verwendung typspezifische Konvertierungsroutinen)
- Character Large Object Table: Speicherung von langen Texten (wenn nötig kombiniert mit Data Table)
- Indexes Pivot Table
    - Aufgrund der flexiblen Strukturen im Data Table kaum Indizierung möglich
    - Lösung: Daten entsprechend ihres tatsächlichen Typs mittels Pivot Table indizieren
- Weitere Pivot Tables
    - Unique Indexes: Eindeutigkeit von Feldinhalten
    - History Tracking: Historie von Feldänderung wird gespeichert

Entwicklungsunterstützung für Anwendungen

- Grafisch unterstützte Funktionen für Workflows, Konformitätstest, Berechnete Felder, …
- Zugriff und Programmierung über zwei verschiedene Web Service Schnittstellen
    - Enterprise WSDL: stark typisiert für Entwicklung von Anwendungen und Datenmanipulation
    - Partner WSDL: schwach typisiert für Zugriff auf Daten mehrerer Kunden
- Effiziente Eingabe und Verarbeitung von Datensätzen: Bulk Processing API - Zusammenfassing von Operationen mittels Arrays als Eingabe
- Management von Daten und Feldentfernung: Gelöschte Felder erstmal nur markiert → Revidieren noch möglich
- Ergänzung von Felder zur Laufzeit einer Anwendung (Ziel: durchgängig verfügbare Anwendungen (Bsp.: Auswahl → Freitextfeld))
- APEX Programmiersprache (Java-ähnliche Syntax)
    - stark typisierte prozedurale Programmiersprache
    - Eingebettete SQL-ähnliche Befehle werden automatisch validiert
    - Starke Laufzeitüberwachung von Programmen: Bsp.: Speicher, CPU, #Requests
    - Integration neuer Anwendungen bei 75% bestandenen Unit-Tests des Codes

Optimierung

- Anfrageoptimierung abhängig von
    - Rechte des anfragenden Nutzers
    - Spezifische Statistiken für Kunden, Benutzer und Gruppen
    - Statistiken für Indizes
- Volltextindizierung von textlastigen Feldern ist ausgelagert in externen Dienst

Fehlertolerante Koordination in Clouds

Verteilte Anwendungen benötigen zwingend Koordination, z.B. bei der Wahl eines Anführers aus einer Menge gleichberechtigter Prozesse, etc. Das ganze kann durch eine Bibliothek realisiert werden, die ein Einigungsprotokoll implementiert oder durch einen Koordinationsdienst wie Chubby und ZooKeeper. Hat man einmal einen Koordinationsdienst implementiert, lässt sich dieser vielseitig einsetzen und bei einer einfachen und verständlichen Schnittstelle reduziert man gleichzeitig die Anforderungen an Anwendungsentwickler. Außerdem können einen einzigen Koordinationsdienst eine große Menge an Anwendungen gleichzeitig nutzen (Chubby: bis zu 90000 Prozesse). Das Programmiermodell ist ähnlich der Koordination mittels gemeinsamen Speichers. Die wichtigsten Anforderungen an solch einen Dienst sind Hochverfügbarkeit, Konsistenz und Performanz.

Chubby

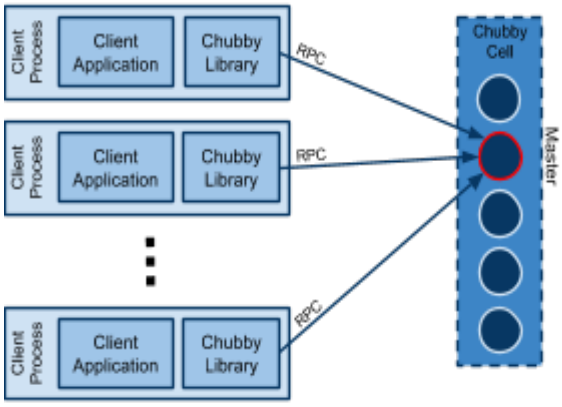
Ein Koordinationsdienst für verteilte Anwendungen in Datenzentren mit einer Dateisystem-ähnlichen Schnittstelle zur Ablage und Verwaltung von Datenobjekten (max. 256KB), verwendet für GoogleFS und BigTable.
Chubby ist implementiert unter Verwendung bestehender Konzepte wie Einigungsprotokollen, Zwischenlagerung von Daten, Benachrichtigungen (Events). Verwendet werden kann Chubby zur Bestimmung eines Anführers, als Namensdienst oder als hochverfügbarer Datenspeicher.
Chubby ist optimiert für lesenden Zugriff und unterstützt Benachrichtigungen über Ereignisse (*Events*). Dateien können mit starker Konsistenz zwischengespeichert werden und Zugriffskontrolle wird über ACL gewährt. Chubby bietet (nur?) Unterstützung für *langlebige Locks*.

Locks

- langlebige Locks
    - reduzieren Last des Koordinationsdiensts
    - geringere Verzögerungen bei Ausfällen des Diensts
    - können Ausfälle des Diensts überdauern (Ausfälle < 30s)
    - geringe Anforderungen an Verfügbarkeit des Dienstes → geringe #Replikate nötig
- kurzlebige Locks
    - höhere Last für Dienst
    - größere Auswirkung für die Anwendungen bei Ausfällen
    - sollten durch Anwendungen selbst implementiert werden

Architektur

In der Regel besteht eine Dienstinstanz (*Chubby Cell*) aus 5 Replikaten, unter denen der Anführer durch den Paxos-Einigungsalgorithmus bestimmt wird. Jedes Replikat unterhält eine Kopie des Dienstzustands im Hauptspeicher. Schreibende Zugriffe müssen durch eine Mehrheit der Replikate bestätigt werden, lesende Zugriffe führt allein der Anführer aus. Ist ein Replikat langfristig ausgefallen, so wird es durch ein neues ersetzt.



Die Anwendungen verwenden die *Chubby Library*, ermitteln den Anführer der Chubby Cell und kommunizieren fortan nur noch mit diesem. Wird ein langfristiger Ausfall erkannt, wird der neue (hoffentlich bereits bestimmte) Anführer gesucht.

Anwendungsschnittstelle

Chubby bietet eine einfache, reduzierte, spezialisierte Dateisystemschnittstelle, die sowohl Daten- als auch Metadaten-Operationen unterstützt.
Explizit **nicht** unterstützt (aber eigentlich üblich für Dateisysteme)

- Verschieben von Dateien
- Zugriffsrechte auf Pfade
- Modifikations- / Zugriffszeitstempel

Die Anwendungsschnittstelle bietet unter Anderem folgende Funktionen: `open, close, poison, delete, setContents, getContentsAndStat, getSequencer, setSequencer, checkSequencer, ...`
`open` erwartet Übergabe eines vollständigen Pfads, überprüft die Zugriffsrechte und bietet Registrierung für Ereignisse und weitere Konfigurationseinstellungen. `close()` trennt/beendet eine Verbindung, während `poison` alle aktuell laufenden Aufrufe beendet.
Es gibt zwei Formen von Objekten: kurzlebige (werden gelöscht sobald nicht mehr benötigt) und permanente. In den Metadaten werden Zugriffsrechte über ACL-Listen verwaltet, verschiedene Zähler für Modifikationen geführt und eine Prüfsumme des Objektinhalts (Daten) gespeichert.
Handles sind in Aufbau und Funktion ähnlich zu Unix-Datei-Handles. Eine wichtige Eigenschaft ist, dass sie auch bei Wechsel des Chubby-Masters erhalten bleiben, was durch Sequenznummer und Zugriffsmodusinformationen ermöglicht wird.

Mechanismen zur Koordination

- Locks
  - jedes Objekt stellt potenzielles Lock dar
  - Unterscheidung zwischen Lese/Schreib-Lock und einfachem Lock
  - sind nur Konvention um Zugriff zu koordinieren, d.h. eigentliche Zugriffskontrolle muss innerhalb der Anwendung erfolgen
  - erleichtern Fehleranalyse
  - zur Belegung eines Locks werden Schreibrechte für das Objekt benötigt
- Sequencers
  - Motivation: Nachrichten können verzögert ankommen oder verloren gehen → Verwendung von Sequencern im Kontext von Locks
  - Sicht des Anwenders: opake Zeichenkette
  - beschreibt Zustand eines Locks nach der Belegung
  - werden innerhalb von Anwendungen von Clients an Dienst übergeben, der mittels Chubby den Sequencer überprüft
- Events
  - Clients können sich für verschiedene Ereignisse / Events registrieren
  - Auftreten eines Ereignisses → Callback an registriete Anwendungen
  - unterstützte Ereignisse: Modifikation eines Objekts, Verwaltungsoperationen innerhalb eines Verzeichnisses, neuer Chubby-Master gewählt, Invalidierung eines Locks oder Handles, Belegung eines Locks

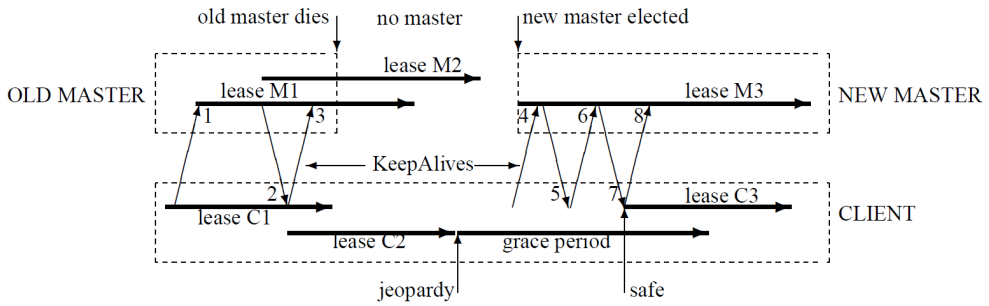
Beispiel: Wahl eines Anführers (Anwendungsebene)

Mehrere Knoten versuchen das gleiche Datenobjekt zu öffnen und gleichzeitig das Lock anzufordern, aber nur ein Knoten gewinnt und schreibt dann seine Identität in das Datenobjekt (`setContents`). Alle weiteren Knoten finden das Ergebnis der Wahl durch `getContentsAndStat` heraus (eventuell durch Ereignis). Anführer fordert dann Sequencer an (`getSequencer`).

Verbindungsmanagement

- Sessions
  - werden durch KeepAlive-Nachrichten auf Funktion überprüft
  - KeepAlive-Nachrichten werden indirekt genutzt um Gültigkeit von Handles, Locks und zwischengelagerten Daten zu überprüfen
  - werden entweder explizit getrennt oder durch Timeout eines Leases
  - Session-Lease muss regelmäßig verlängert werden (durch Quittierung von KeepAlive-Nachrichten) → Problem: neuer Chubby-Master
- Behandlung der Session Leases
  - Clients senden Aufforderung zur Verlängerung eines Lease unmittelbar nach letzter Verlängerung
  - Chubby-Master verlängert erst kurz vor Ablauf des Leases
- Behandlung von Invalidierungen
  - Handles, Locks und zwischengelagerte Daten bleiben erhalten/aktuell, bis Chubby-Master Invalidierung sendet
  - Invalidierung muss durch Clients quittiert werden (→ KeepAlive-Nachrichten)
- Behandlung der Leases
  - Clients verwalten lokal das Session-Lease und überwachen dessen Timeout
  - konservative Abschätzungen bzgl. Timeout aufgrund von Uhrzeitunterschieden
  - Timeout? Sperrung des lokalen Zwischenspeicher, zunächst Erhalt der Session
  - Master meldet sich wieder? Aktualisierung des Zwischenspeichers
- lokaler Zwischenspeicher der Anwendungen
  - enthält Objekt- und Metadaten, sowie Handles und Locks
  - Invalidierung: Master kennt Zwischenspeicher der Clients; bei Schreibzugriffen: Invalidierungsnachricht, Clients entfernen betroffene Daten und quittieren Nachricht, lesende Zugriffe weiterhin möglich (über Dienst direkt)
  - Invalidierungen werden zwar an Clients propagiert, aber Aktualisierung erfolgt erst implizit bei nächster lesender Anfrage

Wechsel des Chubby-Masters



Chubby-Master verwirft seinen lokalen Zustand, Lease-Timer wird angehalten. Zwei Varianten:

- Schneller Wechsel: Wechsel ist vollzogen, bevor Lease abläuft.
- Langsamer Wechsel: Client blockiert Zwischenspeicher und wartet gewisse Zeit. Erlaubt Erhalt von Sessions

… aus Sicht des neuen Chubby-Masters

- Beginn neuer Epoche (nötig um alte Anfragen zu identifizieren)
- initial wird nur auf Anfragen nach dem Chubby-Master (Anführer) geantwortet
- Aufbau des Betriebszustands auf Basis der persistenten Datenbank
- Antworten auf KeepAlive-Nachrichten
- Aussenden von FailOver-Nachrichten an Clients (erwirkt Löschung der Zwischenspeicher)
- warten bis entsprechende Nachrichten beantwortet sind oder Sessions abgelaufen
- Übergang in Normalbetrieb

Backup

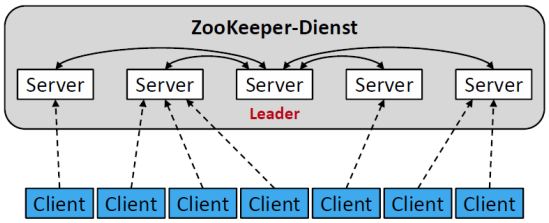
Daten müssen in einer Datenbank konsistent und verfügbar abgesichert werden. Ursprünglich durch BerkeleyDB, auf Grund von technischen Problemen wurde einfachere Lösung implementiert auf Basis eines Einigungsprotokolls.
Es wird alle paar Stunden ein Backup erstellt und in GoogleFS an verschiedenen Orten abgelegt, was die Behandlung von *Katastrophen* und eine schnelle Initialisierung von neuen Replikaten ermöglicht.

Verbesserung der Skalierbarkeit

Clients sind keine Maschinen sondern Prozesse, können also die Zahl der Maschinen um ein Vielfaches übersteigen, daher kann eine hohe Last erzeugt werden und mit steigender Kapazität der Clients steigt auch die Last bei Chubby. Daher:

- Reduktion der Kommunikation essenziell
    - Bereitstellung von mehr Chubby-Instanzen
    - Verlängerung der Lease-Zeit
    - Rigorose Zwischenlagerung von Daten
- Einführung von vermittelnden Proxy-Knoten
    - leiten Client-Anfragen an Chubby weiter
    - Verarbeitung von lesenden Anfragen und KeepAlive-Nachrichten
    - schreibende Anfragen jedoch weiterhin durch Chubby direkt (#writes << #reads)
    - Nachteile: größere Latenz (schreibende Anfragen); zusätzliche Komplexität
- Zustandspartitionierung: Aufteilung des Namensdiensts auf verschiedene Chubby-Instanzen

ZooKeeper



- ebenfalls fehlertoleranter Koordinationsdienst für verteilte Systeme (Teil von Hadoop)
- Fehlertoleranz durch aktive Replikation des Diensts auf mehreren Rechnern
- Replikationskonsistenz mittels Leader-Follower-Ansatz: Leader gibt Reihenfolge vor, Follower bearbeiten Anfragen in vorgegebener Reihenfolge
- 2*f + 1 Replikate zur Tolerierung von f Fehlern / Ausfällen
- unterstüzt synchronen und asynchronen Aufruf

Verwaltung von Daten

- hierarchischer Namensraums (Knoten in Baumstruktur)
- jeder Knoten über eindeutigen Pfad identifizierbar
- Knoten (auch “Verzeichnisknoten”) können (im Gegensatz zu Dateisystemen) Nutzdaten aufnehmen
- Daten werden atomar gesetzt, ersetzt und gelesen (komplett, nie partiell)
- Versionierung der Nutzdaten (Versionsnummer inkrementieren bei Schreibvorgang)
- bedingtes Schreiben möglich (falls Versionsnummer passt: schreiben, sonst nicht)
- verwaltete Metadaten (pro Knoten): Zeitstempel von Erstellung und Modifikation, Versionsnummer und Größe der Nutzdaten, #Kindknoten, Ephemeral-Owner-ID
- Nutz- und Metadaten werden komplett im Hauptspeicher gehalten (was wenn voll?)

Arten von Knoten (Nodes)

- persistente Knoten (**regular**): Erzeugung und *explizites* Löschen durch Client
- flüchtige Knoten (**ephemeral**)
    - Erzeugung durch Client (EPHEMERAL-Flag)
    - explizites Löschen durch Client möglich
    - *implizites* Löschen (ZooKeeper), falls Verbindung zu *Ephemeral-Owner* abbricht
- sequentielle Knoten (**sequential**)
    - Erzeugung durch Client (SEQUENTIAL-Flag)
    - automatische Erweiterung des Namens um Sequenznummer (vom System vergeben)
    - Anwendungsbeispiel: Herstellung einer Ordnung auf Clients

Benachrichtigung über Ereignisse (Events)

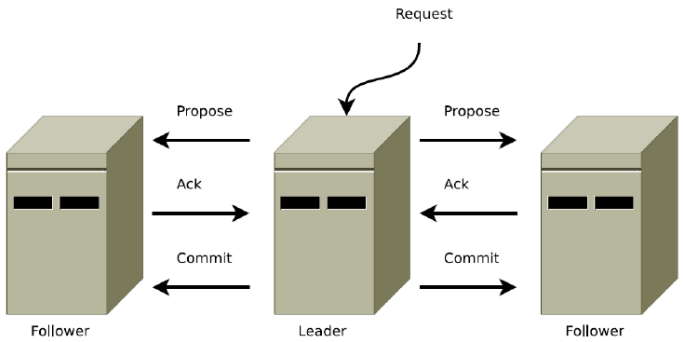
Aktives Nachfragen durch Client ineffizient, daher *Watches*, Objekte auf Client-Seite, die einen Rückruf (*Callback*) des ZooKeeper-Diensts entgegennehmen.
Anwendungsbeispiele sind Benachrichtigungen über Erstellung eines Knotens, Löschen eines Knotens oder Änderung der Nutzdaten eines Knotens.

Beispiel: Wahl eines Anführers (Anwendungsebene)

Aus einer Gruppe von Clients soll ein Anführer gewählt werden und bei Ausfall des Anführers muss ein neuer bestimmt werden. Dieses Problem kann mittels einer Kombination aus sequential-ephemeral Nodes und Watches gelöst werden.

Zab

Das von ZooKeeper verwendete Protokoll zur Reihenfolgeherstellung (*Totally Ordered Broadcast Protocol*). Zwei Modi: *Broadcast* (Normalbetrieb) und *Recovery* (Wahl eines neuen Leaders). Ziel ist die Herstellung einer einheitlichen Reihenfolge alle Client-Anfragen.
Dies geschieht mittels PROPOSE - ACK - COMMIT:



Recovery-Modus:

- Abbruch des Broadcast-Modus (Ausfall des Leader oder Leader hat keine Mehrheit mehr)
- Wahl eines neuen Leader: rundenbasierte Abstimmung mit irgendwelchen zxid's, bei Fehlern während der Wahl: Neustart
- nach erfolgreicher Wahl: Leader stellt verloren gegangene Vorschläge und Bestätigungen bereit und der Broadcast-Modus wird wieder aufgenommen

Konsistente Zeitstempel

Es gibt in den Metadaten einen Zeitstempel der letztem Modifikation der Nutzdaten. Lokale Uhren der Replikate können aber voneinander abweichen, also kann es inkonsistente Zeitstempel geben (bei trivialer Implementierung). Gelöst wird das dadurch, dass der Leader jeder Transaktion einen Zeitstempel zuordnet, der seiner lokalen Uhrzeit entspricht. Jeder Follower übernimmt dann diese Zeit, egal wie die Systemzeit des Followers ist.

Optimierungen

Lesende Anfragen haben auf Replikationskonsistenz keinen Einfluss, daher werden sie nur auf einem Replikat ausgeführt. Die Bearbeitung der lesenden Anfrage erfolgt auch **sofort** nach ihrem Empfang. Das spart Ressourcen und sorgt für kürzere Antwortzeiten. Allerdings können Antworten auf eine lesende Anfrage unterschiedlich ausfallen, je nachdem an welches Replikat die Anfrage geschickt wird. Dafür gibt es eine Möglichkeit zur Erzwingung eines Synchronisationspunkt mit `sync`.

Unterschiede: Chubby / ZooKeeper

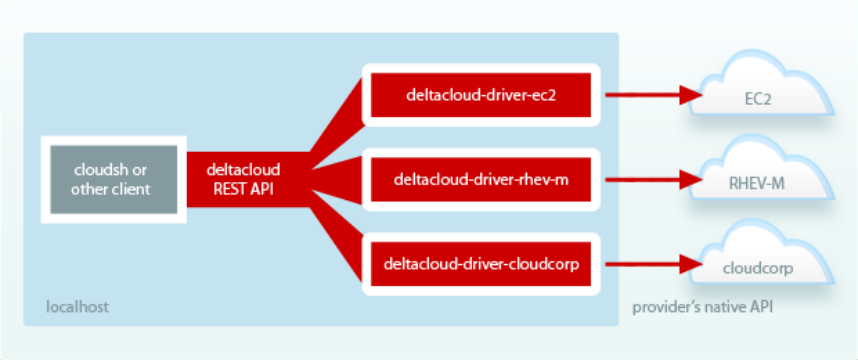
	Chubby	ZooKeeper
Locks	nur langlebig	auch kurzlebig
Kommunikation	Clients nur mit Master verbunden	mit irgendeinem der Server verbunden
Metadaten	kein Modifikations/Zugriffszeitstempel	Erstellungs- und Modifikationszeitstempel + Versionsnummer der Nutzdaten
Wahl eines Anführers (Anwendungsebene)	möglich über Locks auf Objekte	möglich über Benutzung von sequential-ephemeral Node und Watches
Bestimmung des Masters / Leaders	Paxos-Einigungsalgorithmus	Zab
		(to be continued)

Multi Cloud Computing

MCC bezeichnet die parallele Verwendung verschiedener Clouds für die Realisierung eines Diensts. Einzelne Anbieter können für den Anwender transparent ausgetauscht werden und oft spielt Replikation eine zentrale Rolle. Motiviert ist MCC durch die Vermeidung des *Vendor-Lock-In* Problems auf Ebene der Schnittstellen, sowie auch bezüglich des Preis/Leistungs-Verhältnisses. Desweiteren spielt Zuverlässigkeit eine wichtige Rolle. Zwar sind Cloud-Anbieter gegen Komponentenausfälle gerüstet, aber es könnten auch gesamte Datenzentren ausfallen. Außerdem macht die homogene Infrastruktur eines Anbieters Systeme anfälliger.
Viele Cloud-Dienste auf Infrastrukturebene (IaaS) verfügen über eine einfache, oft generische Schnittstelle und neue Anbieter orientieren sich häufig an Marktführern (Amazon). Doch MCC wird mit der Nutzung komplexerer Dienste zunehmend schwieriger (vgl. force.com oder MS Azure).

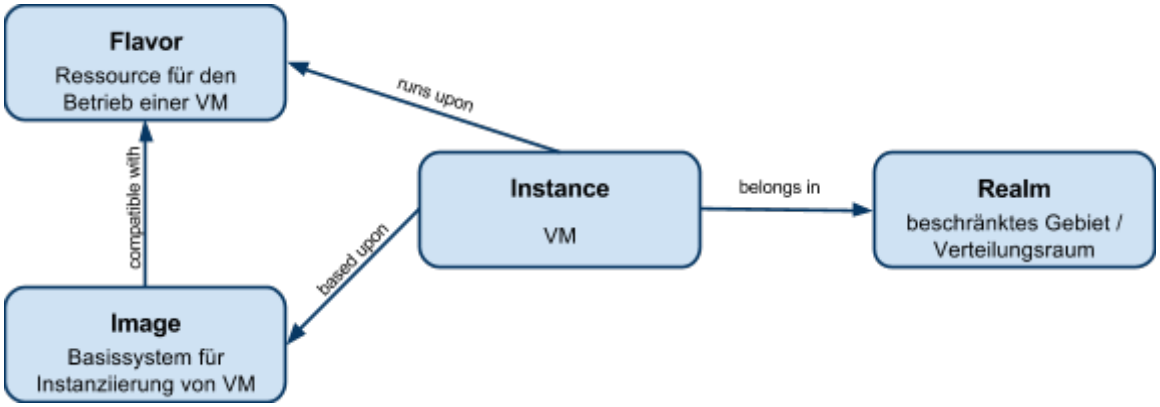
Deltacloud

Deltacloud hat die Entwicklung einer einheitlichen Schnittstelle zum Betrieb von Infrastruktur-Clouds zum Ziel. Der Zugriff erfolgt über eine Web Service-Schnittstelle, die von einem Proxy zur eigentlichen Zielplattform vermittelt wird. Es existiert als Alternative *Simplecloud*, die einheitlichen Zugriff auf Dateien, Dokumente und Ereignisse unterstützt.

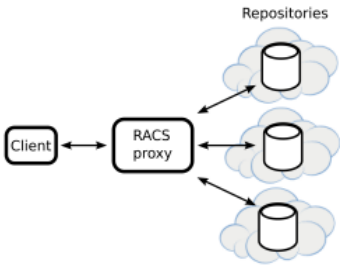


Unterstützte Basiskonzepte

Unterstützte Basiskonzepte sind Flavors, Realms, Images und Instances.



Redundant Array of Cloud Storage (RACS)



RACS ist motiviert durch die Prävention der *Vendor-Lock-In* Problematik, genauer dadurch, dass die Umlagerung von Daten kostenintensiv sein kann und die Kosten mit der Menge der gelagerten Daten steigen. Desweiteren bietet RACS Fehlertoleranz, dadurch können z.B. auch Ausfälle durch Insolvenz eines Cloud-Anbieters abgefangen werden. RACS verfolgt den Ansatz der gleichzeitigen Nutzung mehrerer Anbieter und benutzt *Erasue Codes* um Replikationskosten zu reduzieren und gleichzeitig Fehlertoleranz zu bieten.

Erasure Codes

RACS verwendet Reed-Solomon-Code. Eigenschaften von Erasure Codes:

- erlauben Vorwärtsfehlerkorrektur bei der Datenübertragung
- Daten werden in m gleich große Pakete aufgespalten
- spezielles Codierungsverfahren bildet die m Pakete auf n Pakete ab
- Anzahl zur Rekonstruktion benötigte Datenpakete r = m/n < 1
- Originaldaten können rekonstruiert werden wenn mindestens m Pakete vorliegen

Basisarchitektur

- Schnittstelle ähnlich zu S3
  - `put()`: n Dateneinheiten auf n Anbieter abbilden
  - `get()`: m Dateneinheiten holen
  - `list()`: kontaktiert genau einen Anbieter (Metadaten werden nämlich nicht encodiert)
- temporäre und permanente Ausfälle von maximal n-m Anbietern tragbar
- Möglichkeit zur Preis/Leistungs-Optimierung bei Benutzung der m günstigsten Anbieter
- Flexibilität bei der Integration neuer Anbieter

Verteilte RACS-Variante

Zur Vermeidung des Proxy-Flaschenhalses wird ZooKeeper benutzt um parallelen Zugriff auf die gleichen Datenobjekte durch mehrere Proxies zu erreichen. Schreibvorgänge müssen hierfür aber exklusiv sein, Leseoperationen können problemlos nebenläufig erfolgen.