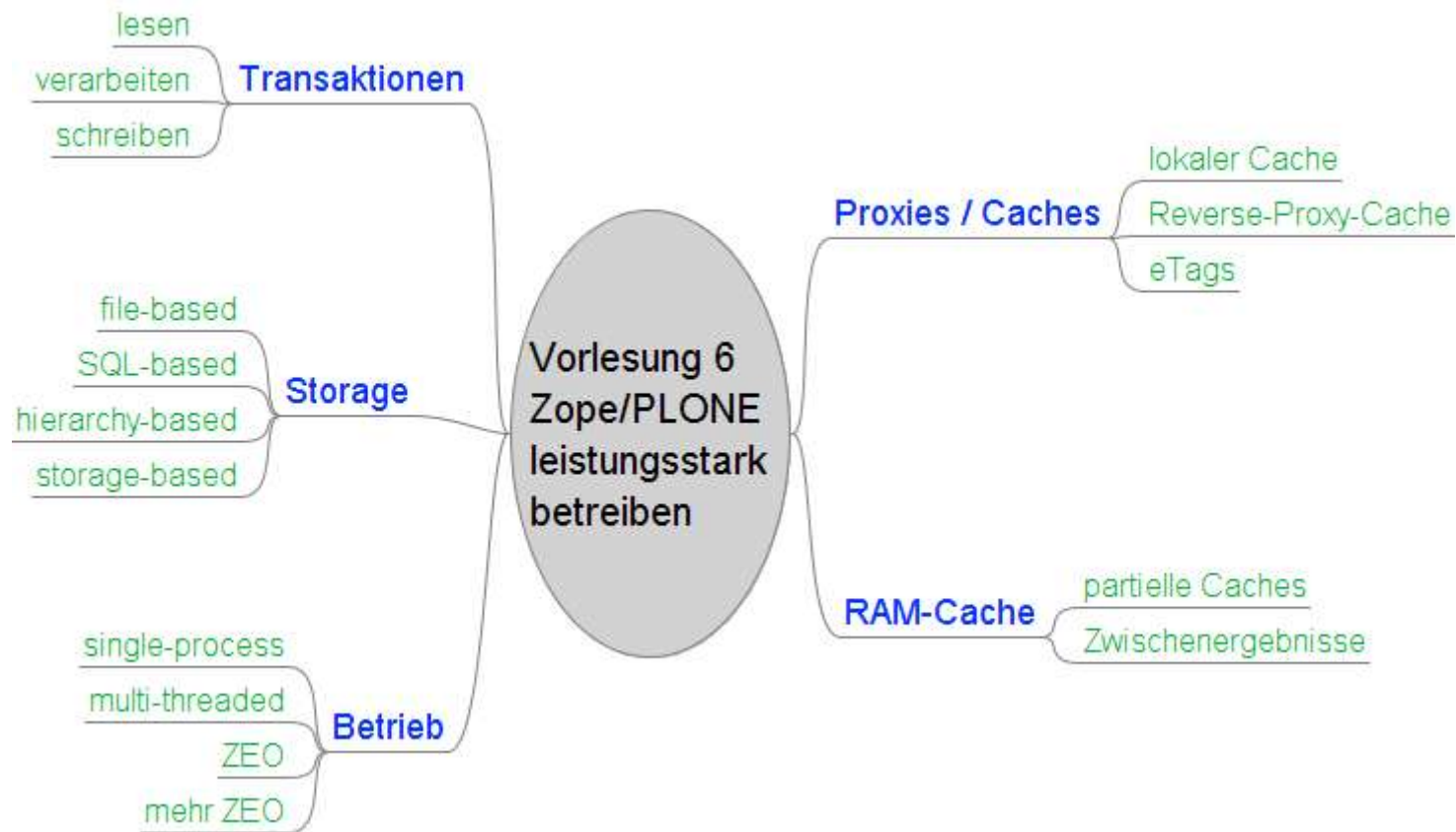


Content-Management-Systeme

Dipl.-Inform. Roman Jansen-Winkeln

Vorlesung 6: PLONE leistungsstark betreiben

Inhalt und Organisation



Transaktionen

Transaktionen kapseln jede Anfrage

Transaktionen - Prinzip

- Transaktion =
eine feste Folge von Operationen, welche als eine logische Einheit betrachtet werden
- ACID-Prinzip
 - Atomarität (Atomicity)
 - Konsistenz (Consistency)
 - Isolation (Isolation)
 - Dauerhaftigkeit (Durability)

Transaktionen - Zope

- Jede Anfrage ist in einer Transaktion gekapselt
 - HTTP-GET eröffnet Transaktion mit BEGIN
 - Persistente Objekte lesen aus Storage in RAM
 - Verarbeitung durch Methoden, Skript, Templates
 - Speichern veränderter/neuer Objekt im Storage
 - ROLLBACK bei Transaktionsfehlern
 - HTTP-PUT beendet die Transaktion mit COMMIT
- Datenintegrität sichern durch OPTIMISTIC LOCK

Optimistic Lock

- Optimistic Lock
Optimistic Concurrency Control (OCC)
- Verfahren
 - **Lesen:** Daten aus Storage lesen, private Kopie anlegen, Ausgangswerte zusätzlich speichern
 - <Verarbeiten der Daten in der Transaktion>
 - **Validieren:** prüfen, ob sich der Ausgangswerte eines geänderten Werts im Storage geändert hat --> Konflikt!!
 - Bei Konflikt: Konfliktauflösung versuchen.
Falls Fehlschlag: ROLLBACK
 - **Write:** Änderungen in Storage schreiben, eigene Kopien löschen, COMMIT

Optimistic Lock

- Beispiel:

- Class myTest(ZODDBObject):
 zahl = 2

- Def miniTrans(self):
 wert = self.zahl
 self.zahl = wert * 2
 return

- Gtest = myTest()

....

- Gtest.miniTrans()

- Bei Aufruf von miniTrans() sei self.zahl == 2
Deshalb gilt wert == 2
Andere Transaction schreibt self.zahl = 3
Bei „return“ will miniTrans() schreiben self.zahl = 4
KONFLIKT

- Def _p_resolveConflict(self, old, saved, new):
 return new*2

Optimistic Lock

- Vorteile Optimistic Lock
 - Kompatibel mit ACID-Prinzip
 - Gut geeignet, wenn selten Konflikte auftreten
 - Sehr schnell, weil keine echten Datensperren erforderlich sind
- Nachteile Optimistic Lock
 - Bei vielen Konflikten ineffizient
 - Verhindert konflikthafte Transaktionen erst am Ende
 - Sagt nicht, wie auf Rollback reagiert werden soll
- Fazit
 - Web-Umgebung lesen viel, verändern wenig
 - Optimistic Lock für Zope geeignet

Transaktionen

- Bei Rollback: Transaktion n=3-mal wiederholen
- Transaktionen synchronisieren RAM-Speicher mit persistentem Storage
- Zope unterstützt Subtransaktionen
 - Erlauben bessere Speichernutzung
 - Machen Speicherveränderungen früher für alle sichtbar
 - Verhindern ineffektives „weiterrechnen“ bei Konflikten
- Literatur
 - <http://www.zope.org/Documentation/ZODB2>

Storage

Storage

- Zope speichert Objekt in der ZODB = Zope-Object-Database
- ZODB stellt Objekte und Operationen ähnlich einem Dictionary zur Verfügung
 - `MyDB['key1'] = myTest()`
 - Neue Instanz von `myTest()` wird in ZODB unter dem Namen 'key1' gespeichert
- ZODB garantiert dauerhafte Speicherung
 - Dazu verwendet die ZODB sogenannten STORAGE
- Was ist als Storage für Python-Instanzen geeignet?

Storage – File Based

- Datei-basierter Storage
- Alle Objekte stehen in einer Datei
per Konvention: var/Data.fs
- Python-Instanz wird als String-kodiert (serialisiert):
Methoden: pickle / unpickle
- Kodierung als XML-Datei oder Attribute-Wert-Paare.
- Objekt-Zugriff:
 - Neue leere Instanz erzeugen
 - Daten aus ZODB-suchen und String lesen
 - Instanz durch „unpickle“ füllen

Storage – Relational Storage

- Relationaler Storage – Speicherung in SQL-DB
- Objekt-Relationales Mapping erforderlich
- Intelligentes Mapping:
 - 1 Klasse == 1 Tabelle
 - 1 Objekt == 1 Tabellen-Record
 - 1 Field == 1 Tabellenattribut
- Generisches Mapping
 - Tabelle für alle Objekte
 - Tabelle für alle Attribute
 - Verknüpfung über Foreign Keys
- Storages für MySQL, PostGRES, (Oracle, MS SQL-Server)

Storage – Hierarchical Storage

- Directory Storage – Speicherung in Verzeichnis und Datei
- Hierarchische Objekt-Struktur direkt auf Datei-System abbilden
- Jedes Objekt ist eine Datei
- Jedes Folder-/Container-Objekt ist ein Verzeichnis

- Leicht lesbare Datenstruktur
- Einfache Implementierung
- 30% größer und langsamer als File-based

Storage – Aggregated Storage, Storage as Storage

- Aggregated Storage: Komplexe Datenverwaltungssysteme
Beispiel: LDAP
- Nutzt die komplexen Speichermöglichkeiten um Python-Instanzen persistent auszulagern.

- Storage as Storage:
Verwendet eine andere ZODB als Storage
- Beispiele: Demo-Storage, ZEO-Storage

Storage – Transaktionen

- ZODB-Storages unterstützen Transaktionen
 - ZODB implementiert Optimistic Lock
 - History mit Änderungen == Transaktionsprotokoll == Transaction Log
 - Frühere Objekt-Zustände werden als sog. *Revision* aufbewahrt
- Transaktionsprotokoll:
 - Storage implementiert optional Transaktionsprotokoll
 - Rollback über Transaktionsprotokoll möglich
== Undo-Operation
 - Speicherung von Revisionen erfordert viel Speicherplatz
 - Storage benötigt „Packing“-Operation
 - Einfacher Copy-Pack-Algorithmus für File-Storage

Zope-Storages

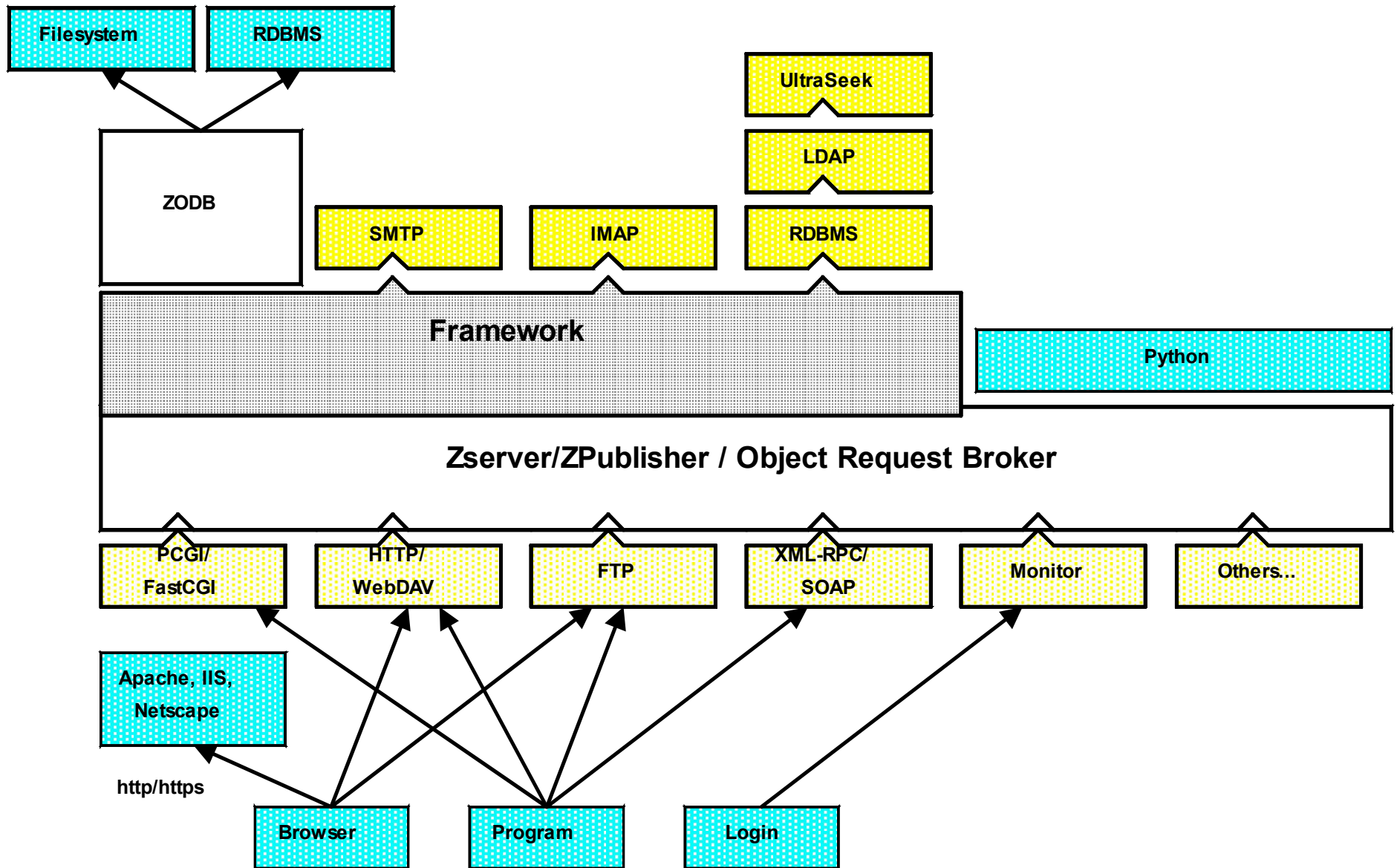
- Zope konfiguriert, welcher Storage verwendet wird
- Mehrere Mount-Points in einem Zope-Server
 - Unterstützt mehrere unabhängige Storages
 - Schwierig beim Kopieren und bei Transaktionen

Zope-Betrieb

Betrieb – Single Prozess

- Einfachster Fall: Zope als normales Python-Programm
 - Python -setup.py -.....
 - Gestartet als Shell-Skript bzw. BAT-File
 - Alternativ als Demon bzw. Service ausgeführt
- In einem Prozess laufen:
 - Integrierter Web-Server
 - Zope-Verarbeitung (Application-Server)
 - ZODB mit Zugriff auf File-based Storage Data.fs
- Schwachpunkt
 - Parallel Anfrage werden innerhalb des Prozesses geschedult
 - Internes Zope-Scheduling ist nicht gut

Betrieb – Single Prozess



Betrieb – Multiple Threads

- Ein Prozess mit mehreren Threads
- In einem Prozess laufen:
 - Gemeinsamer, integrierter Web-Server
 - Mehrere Thread mit Zope-Verarbeitung (Application-Server)
 - ZODB mit Zugriff auf File-based Storage Data.fs
- Ablauf:
 - Alle Anfragen gehen an den gleichen Web-Server
 - ORB verteilt Anfragen auf feste Anzahl Verarbeitungs-Threads (4-10)
 - Bei mehr parallelen Anfragen: Zope-internes Scheduling
 - Alle Objekte werden aus der gleichen Datenbank gelesen

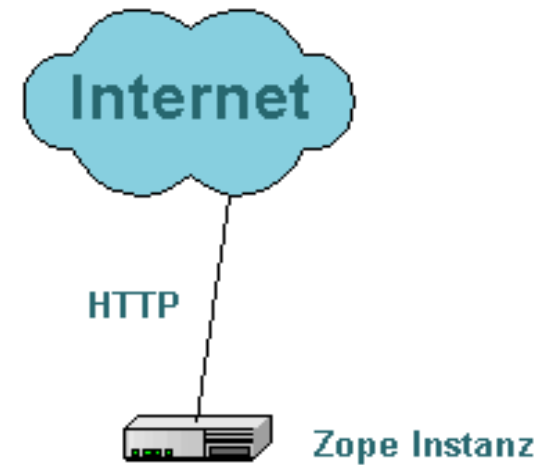
Betrieb – Multiple Threads

- Verbesserungen:
 - Threads schedulen über das Betriebssystem besser
- Schwachpunkte:
 - Alle Threads laufen im gleichen Python-Executable
 - Python kann keinen Multi-Prozessor-Support
 - Deshalb nutzen Multi-Thread-Zopes auch nur einen Prozessor
- Gute Betriebsform bei mittlerer Last und starkem Prozessor

Betrieb – Multiple Threads

Nicht skalierbar

Standard Zope



Betrieb – Zope Enterprise Objects – ZEO

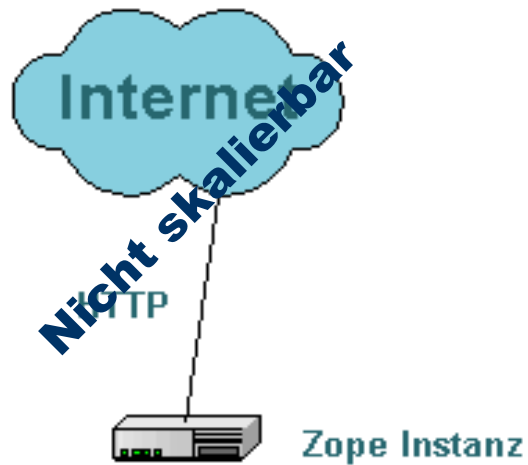
- Mehrere Prozesse mit je einem vollständigen Zope
 - Alle Zopes benutzen den gleichen Storage
 - Jedes Zope hat seinen eigenen Web-Server
 - Einzelne Zopes als Single-Process oder Multi-Threaded
- ZEO-Server und ZEO-Client-Storage
 - Spezieller ZEO-Client-Storage
 - Greift auf den Storage eines ausgewählte Zopes zu: ZEO-Server
 - ZEO-Server = Minimales Zope: ZODB+Storage + ZEO-Protokoll
 - ZEO-Client übernimmt Anfrage und Verarbeitung

Betrieb – Zope Enterprise Objects – ZEO

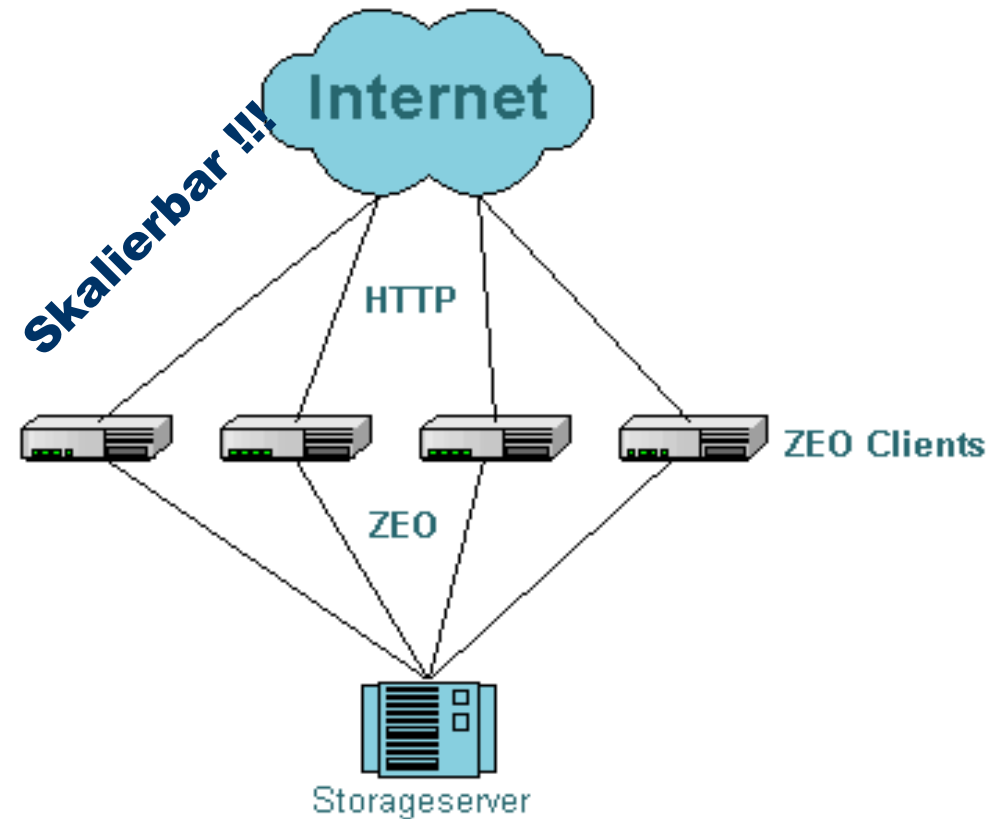
- Verbesserungen:
 - Mehrere ZEO-Clients auf einem Rechner nutzen Multi-Prozessoren
 - ZEO-Clients auf mehreren Rechnern möglich
 - Kaskaden mit ZEO-Server möglich
 - Dedizierte ZEO-Clients möglich: Read-Only, Long-Running-Jobs, ...
 - Synchronisation über gemeinsamen ZEO-Server-Storage
- Schwachpunkte:
 - Verteilung der Anfragen auf ZEO-Clients:
Dedizierter Load-Balancer, Web-Server mit Load-Balancer,
Manuelle Verteilung,
 - Ausfall des ZEO-Servers
- Robuste Betriebsform bei hoher Last und vielen Prozessoren

Betrieb – Multiple Threads

Standard Zope



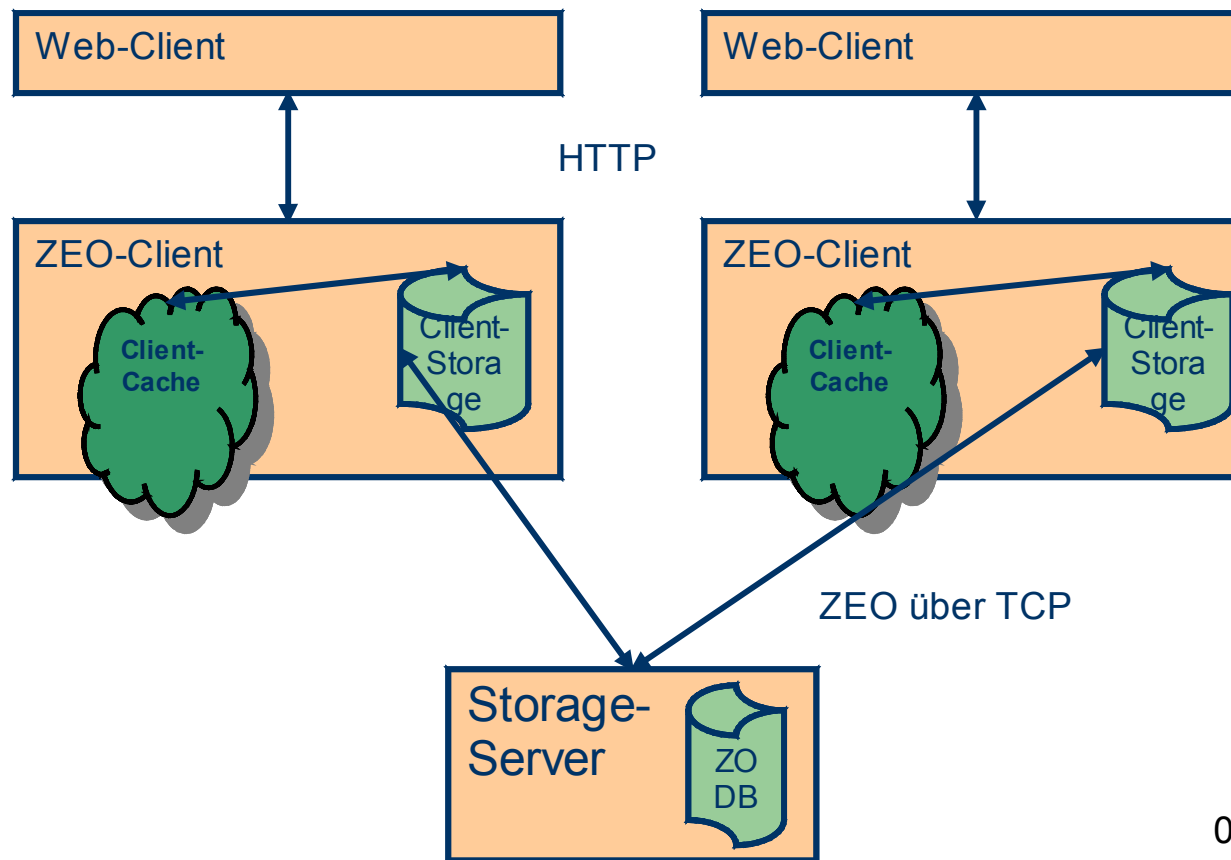
ZEO (Zope Enterprise Objects)



Betrieb – ZEO-Storage

- ZEO-Client-Storage: Storage as Storage
- Beinhaltet alle Funktionen eines echten Storage
- Eigenes IP-Protokoll zur Kommunikation mit ZEO-Server:
Z3-Handshake-Protokoll
- ZEO-Server: abgespeckter Zope-Server
- ZEO-Client hat lokalen Cache um alle abgerufenen Objekte zu speichern

ZEO: Architektur

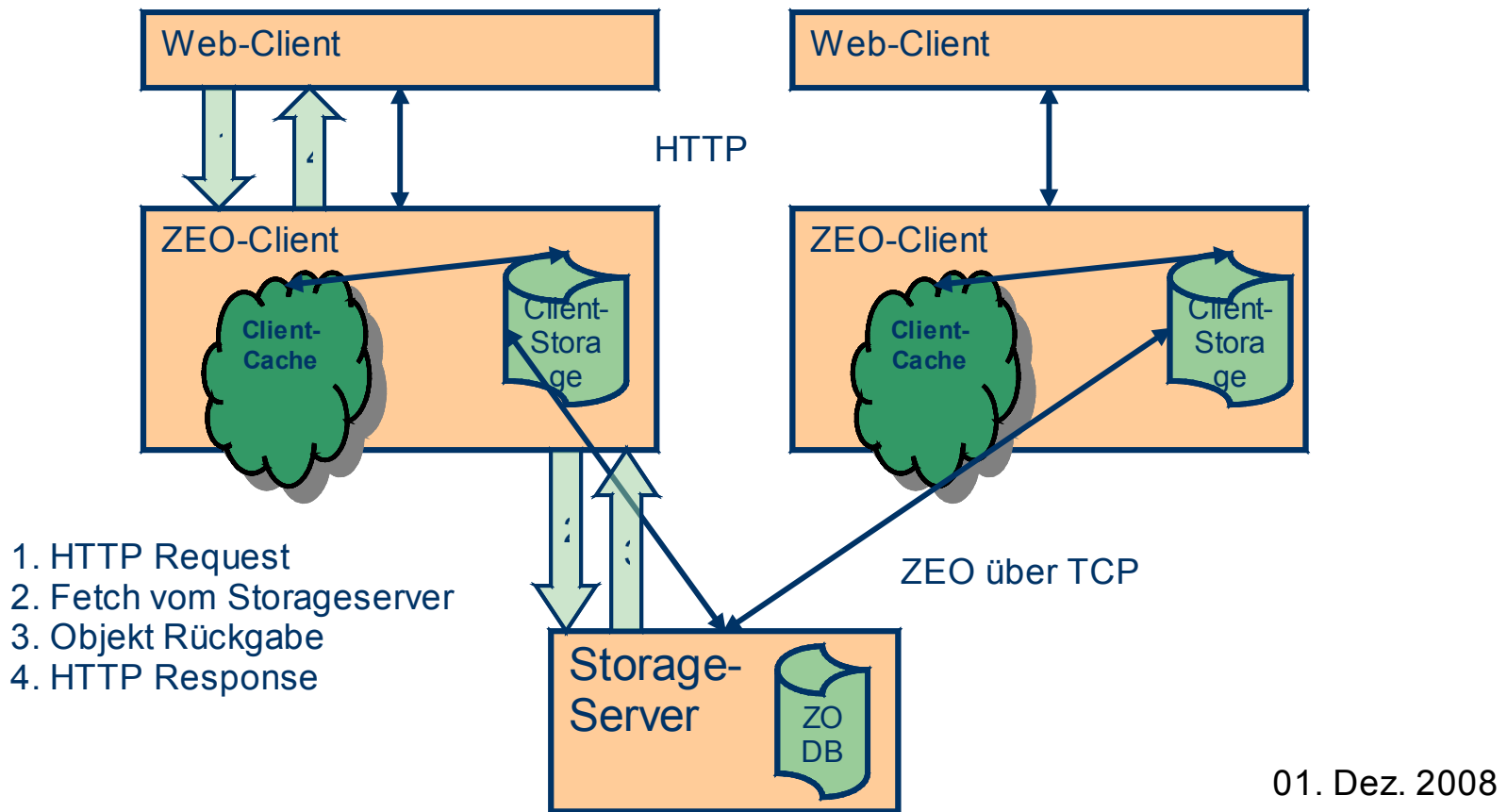


01. Dez. 2008

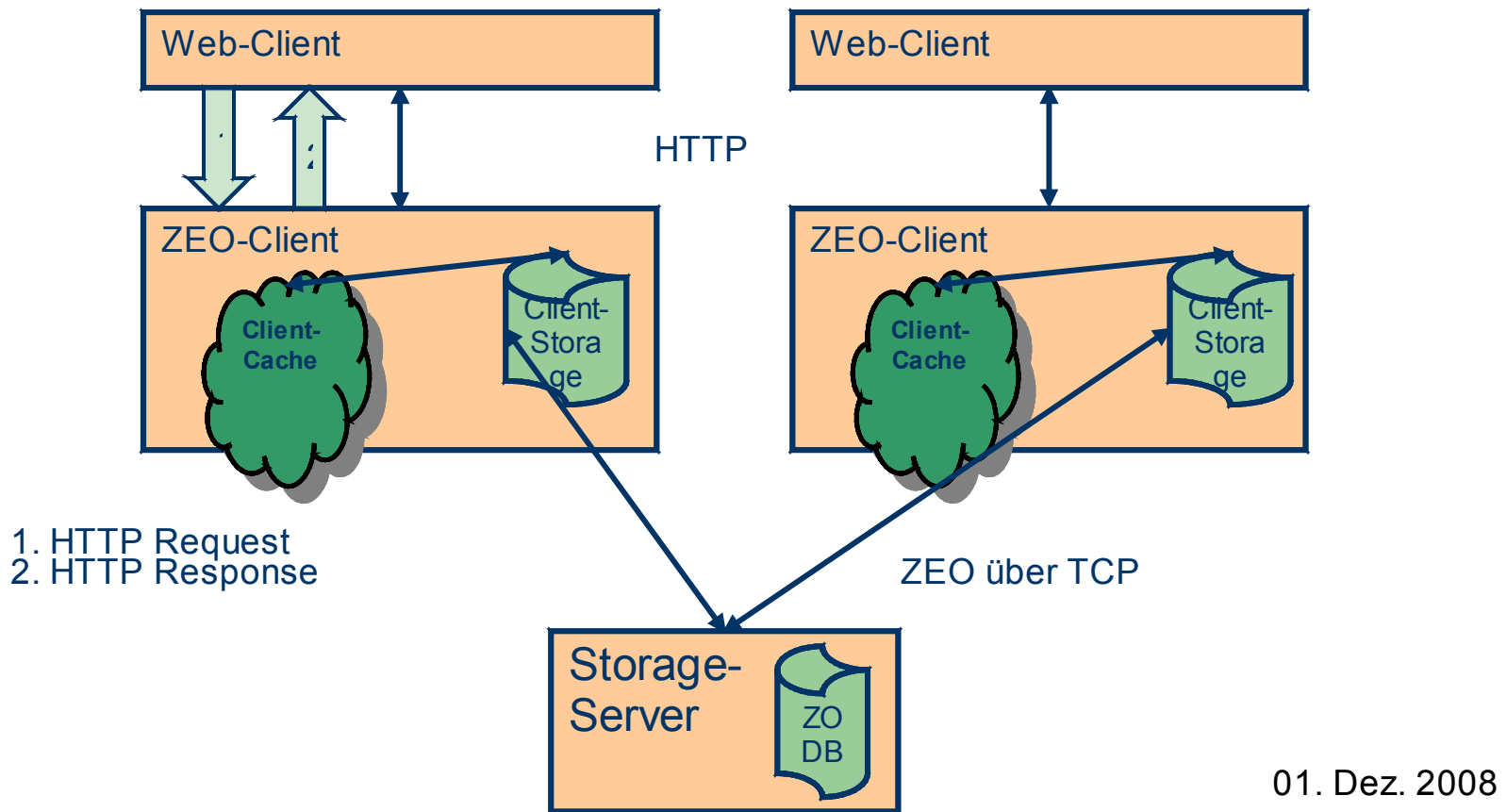
Betrieb – ZEO-Storage

- Lesende Anfragen in ZEO
 1. Lesende Anfrage aus einer Transaktion an den ZEO-Storage
 2. Der ZEO-Storage sucht im lokalen Cache nach angefragten Objekten
 3. Falls nicht lokal gecacht fragt der ZEO-Client-Storage den ZEO-Server nach dem Objekt an
 4. Zurückgeliefertes Objekt wird im lokalen Cache gespeichert.
 5. Objekt wird an die Transaktion geliefert.
- ZEO-Storage wird dank Cache immer schneller
- Vollständige Kopie des Server-Storage denkbar

ZEO: Laden (nicht gecacht)

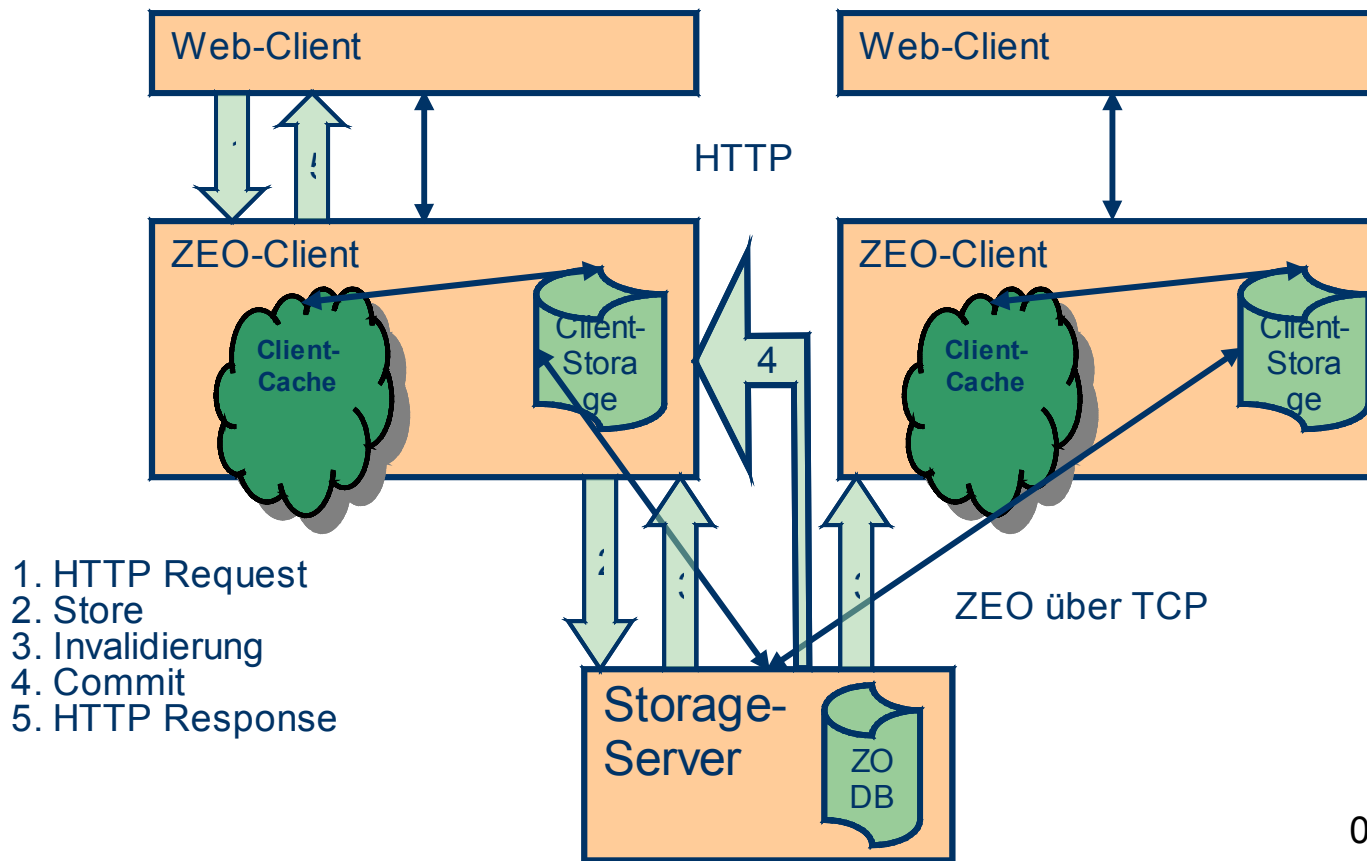


ZEO: Laden (aus Client-Cache)



- Schreibende Anfragen in ZEO
 1. Schreibauftrag für ein Objekt aus der Transaktion.
 2. ZEO-Storage überprüft lokal Optimistic-Lock und sendet den Schreibauftrag an den Server.
 3. Der ZEO-Server speichert das Objekt in seinem Storage
 4. Der ZEO-Server sendet **allen** ZEO-Clients ein „invalidate“-Signal für dieses Objekt.
 5. Alle ZEO-Clients markieren daraufhin dieses Objekt in ihrem lokalen Cache als nicht mehr aktuell bzw. löschen es.
 6. ZEO-Server bestätigt Commit an auslösenden ZEO-Client
 7. Storage bestätigt Commit an die ausführende Transaktion

ZEO: Speichern



01. Dez. 2008

Betrieb – ZEO-Storage

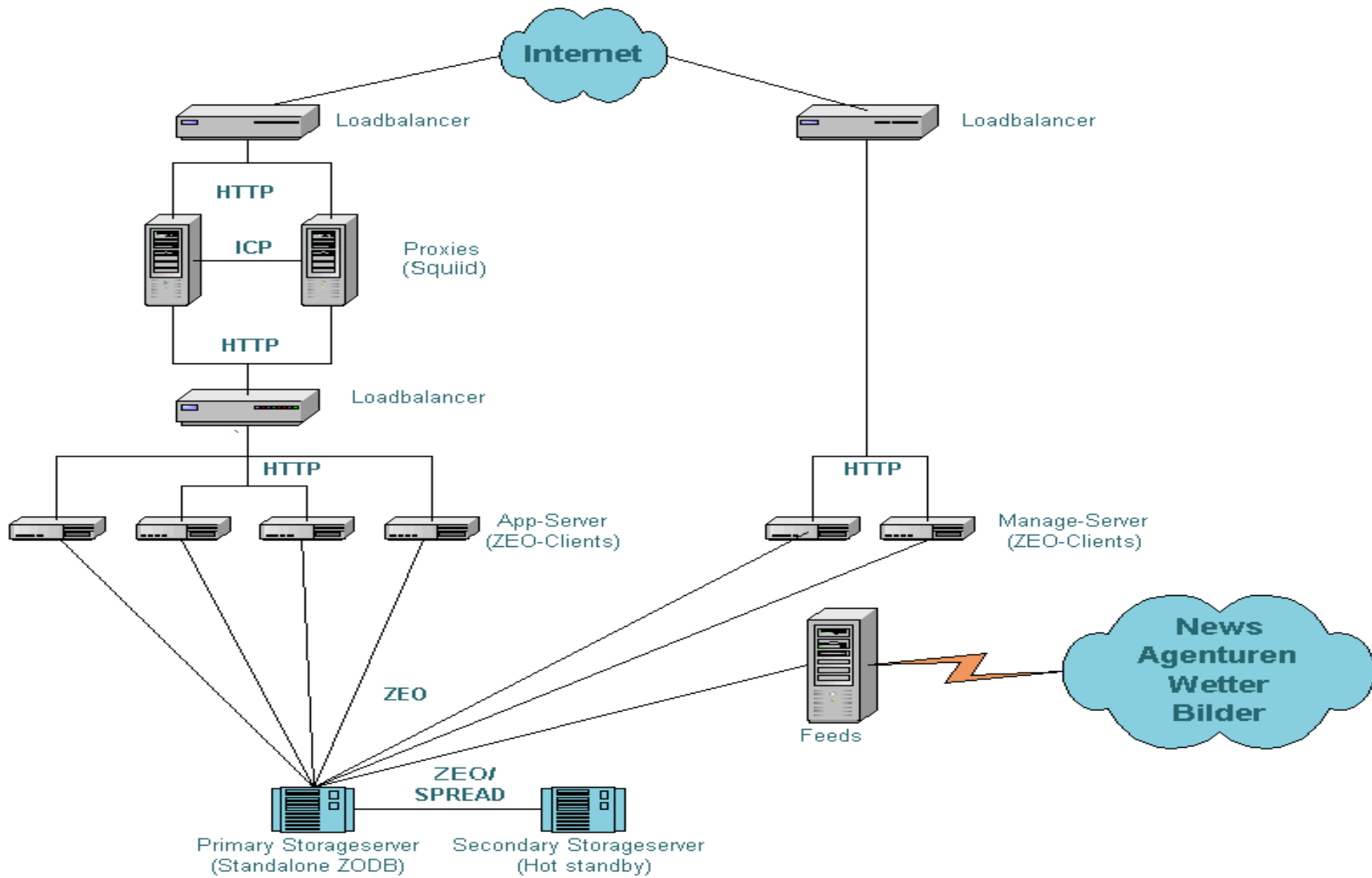
- Schreibende Anfragen in ZEO sind langsamer
 - In Web-Server-Umgebungen akzeptabel, da überwiegend Datenabruf
- Schreiben wird langsamer je mehr ZEO-Clients aktiv sind
 - Typischerweise 2-20 ZEO-Clients
- Achilles-Ferse ist der ZEO-Server
 - Redundant auslegen
 - Stand-by-Server mit kurzen Anlaufzeiten verwenden
 - Bei File-Based ZEO-Storage ggf. NAS für Data.fs verwenden
 - Variante: Relational-Storage mit ausfallsicherem RDBMS verwenden

ZEO-Betrieb in Multi-Server-Szenarien

- Die Verteilung der ZEO Prozesse auf Hardware ist ein heuristischer Prozess.
- Bei der Ermittlung der optimalen Verteilung helfen Tests und Erfahrung.
- Sinnvoll ist die Nutzung von Partitionierungsmöglichkeiten
 - Anwendungs-Cluster zu bilden
 - einfach und ohne Zope-Kenntnisse installieren und vervielfältigen
 - Beispiel: Blade-Server, SUN-Zones oder Linux-VMs.

ZEO-Betrieb in Multi-Server-Szenarien

- Typische Konstellation in Blade-Architekturen:
 - Master-Blade mit Apache, Load-Balancer und ZEO-Server
 - Redundantes Master-Blade als Stand-By
 - Arbeits-Blade, je nach Leistungsfähigkeit mit 2-4 ZEO-Clients
 - NAS als gemeinsames Speichersystem für alle Blades
- SUN-Umgebungen unter SOLARIS
 - an Stelle von Blades kann man Zones definieren
 - Die beiden Master-Zones auf unterschiedlicher Hardware verteilen
- Arbeits-Blades bzw. -Zones werden sooft vervielfältigt wie die Benutzerlast es erfordert und die Hardware es erlaubt.



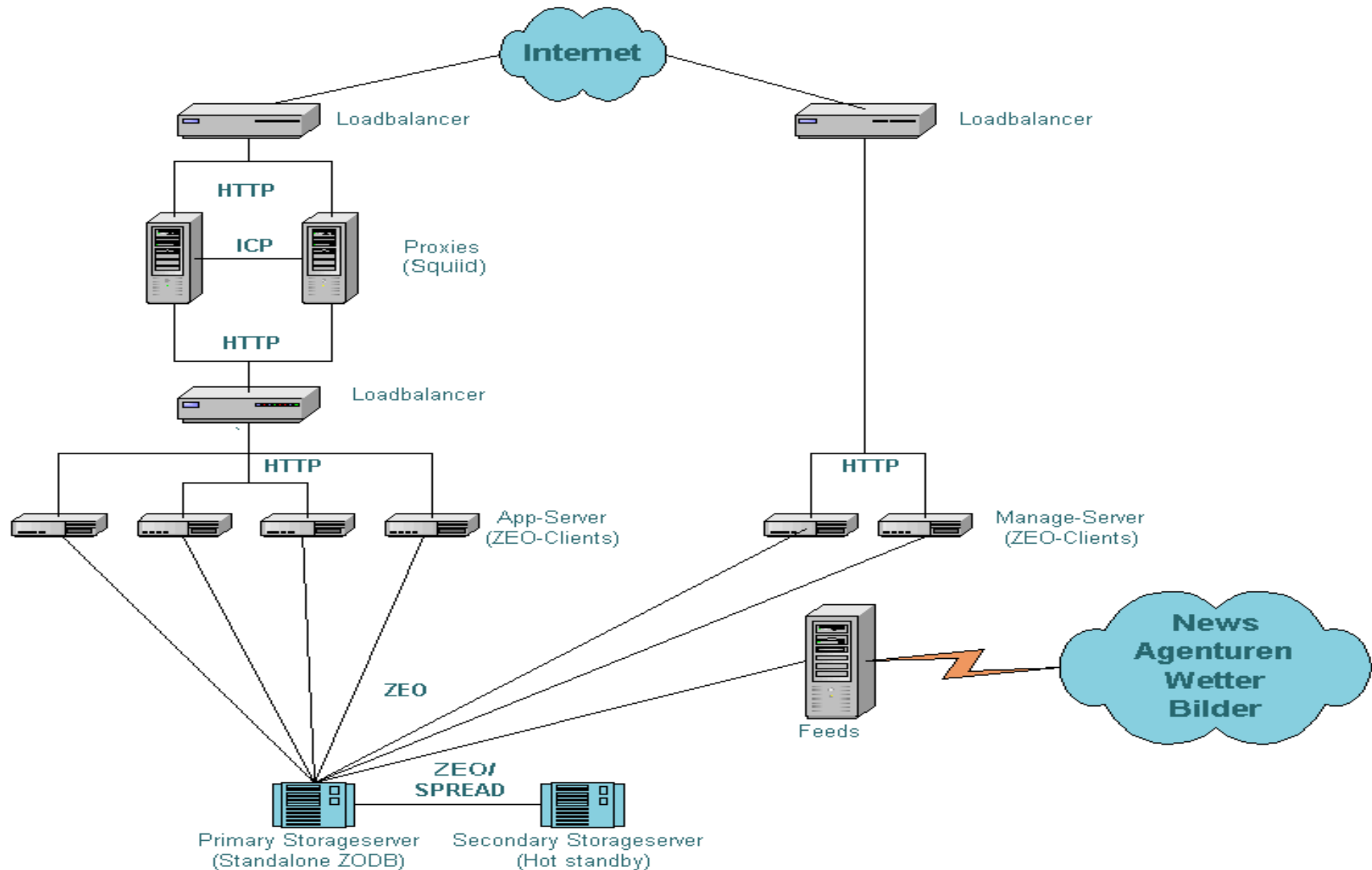
01. Dez. 2008

Reverse-Proxy + Cache

Web-Server als Reverse-Proxy/Cache-Frontend

- Normaler Web-Server leitet Anfragen an Zope weiter.
- Vorteile:
 - Robuster Vor-Filter für Anfragen (Abwehr von Angriffen)
 - Nutzt integrierten Cache zur Beschleunigung
 - Hostet rein statischen Content (Apache Faktor 1.000 schneller)
 - Unterstützt SSL / HTTPS – Protokolle
 - Integriert Load-Balancer für ZEO-Verteilung
- Beispiele:
 - Apache
 - LH-Proxy, ????
 - (Microsoft IIS), (Squid)

Revers-Proxy/Cache vor Zope



Fakten – Fakten - Fakten

- Performanz:
 - 10 Millionen Hits/Tag
 - ca. 900 Hits/Sekunde (Peak)
- Squid-Cache Hitrate: ~95 Prozent
- Situation am 11. September 2001:
 - Zope Cluster waren ausnahmslos erreichbar
 - Konkurrenzsites waren stundenlang nicht erreichbar

01. Dez. 2008

Zope-Support für Reverse-Proxy/Cache

- Virtual Host Monster
 - Zope-Product für URL-Rewriting
- Caching
 - Zope setzt Cache-Hinweise in HTTP-Header
Cache ja/nein/dauer
 - Cache-Hinweise je nach Objekt-Typ, z.B. für Bilder
 - Cache registriert sich bei Zope
Zope sendet Invalidate an Cache bei Datenänderungen

Zope-Support für Reverse-Proxy/Cache

- Etags: Cache nach Inhalt
 - Normales Cache-Kriterium: Name, Alter (TTL), Änderungsdatum
 - Portal-Rahmen: anderes Datum im Calendar-Portlet
 - Alternative: Alfa-numerischer Schlüssel
Beispiel: Version oder MD5
 - Wird als ETAG im Header übermittelt
 - Dokumenttyp kann eigene ETAG-Methode implementieren
--> inhaltliche Entscheidung, ob ETAG sich ändert
- Edge Side Includes ESI: partielle Web-Seiten Cachen
 - Spezielle XML-Tags (ESI-ML) in View-Methoden
 - Separates Caching z.B. der Portlets und des Contents
 - Support: Squid, Oracle AS Web Cache

Zope-interner RAM-Cache

- Partielles Caching in Zope
 - Zwischenspeicher vor der View-Methode
 - Entscheidet pro Objekt, ob die view neu berechnet wird.
- Beispiel: Calendar-Portlet
 - Wird nur beim ersten Aufruf am Tag berechnet
 - Wird neu berechnet, wenn jemand einen Kalender-Eintrag ändert
 - Wird sonst aus dem Cache abgerufen

Caching und Benutzerrechte

- View hängt von der URL und mehr ab
 - Hängt ab von angemeldeten Account
 - Hängt ab von vorangegangenen Transaktionen, gespeichert im internen SESSION-Objekt
- Caches haben Mühe diese MEHR zu erkennen
 - Abhängigkeiten im HTTP-Header und in Cookies sichtbar machen
 - Cache so konfigurieren, dass er alle Abhängigkeiten berücksichtigt
 - RAM-Cache erkennt alle Abhängigkeiten.
- Einfache Cache-Strategien für Web-Server
 - Alle anonymen Anfragen cachen
 - Angemeldete Anfragen über andere, nicht-gecachte URL laufen lassen

Internationalisierung

Aktive Suche