# World Wide Web
# HTTP - Mechanismen zur Optimierung

STUDIEREN AUF HÖCHSTEM NIVEAU

Prof. Dr. Jürgen Anders, Hochschule Furtwangen
Fakultät Digitale Medien

# Ausführung von HTTP muss optimiert werden

- Moderne Webseiten enthalten zahlreiche eingebettete Ressourcen wie z.B. Stylesheets, JavaScript, Bilder, Videos, …
  - **Beispiel**: Öffnen der DM-World Webseite erzeugt: 43 (!) HTTP-Request/-Response-Zyklen
- Bei Ausführung eines HTTP-Request/-Response-Zyklus muss zunächst TCP-Verbindung auf- und dann wieder abgebaut werden
  - Auf- und Abbau einer TCP-Verbindung verlangt Ausführung eines TCP-Handshakes und verursacht dadurch signifikanten Overhead
- Es werden dringend HTTP-Mechanismen gebraucht zur Request-Minimierung und zur Geschwindigkeitserhöhung, wie
  - Komprimierung
  - Persistente Verbindungen und HTTP-Pipelining
  - Caching

## Komprimierung

HTTP/1.1 erlaubt Server, Daten im Nachrichtenrumpf komprimiert zu übertragen; HTTP/2 erlaubt auch, den Nachrichtenkopf zu komprimieren

-> Höhere Übertragungsgeschwindigkeit und Bandbreitenausnutzung

- Clients teilen mögliche Komprimierungsalgorithmen im

  Request-Header mit: **`Accept-Encoding: gzip, deflate`**
- Server wählt passenden Komprimierungsalgorithmus und teilt diesen im Entity-Header mit: **`Content-Encoding: gzip`**
- Häufige HTTP-Komprimierungsalgorithmen:
  - **deflate**: Deflate-Algorithmus (RFC 1951)
  - **gzip**: GNU zip-Algorithmus (RFC 1952), am weitesten verbreitet
  - **exi**: W3C - Efficient XML Interchange
  - **identity**: Keine Komprimierung

# Persistente Verbindungen

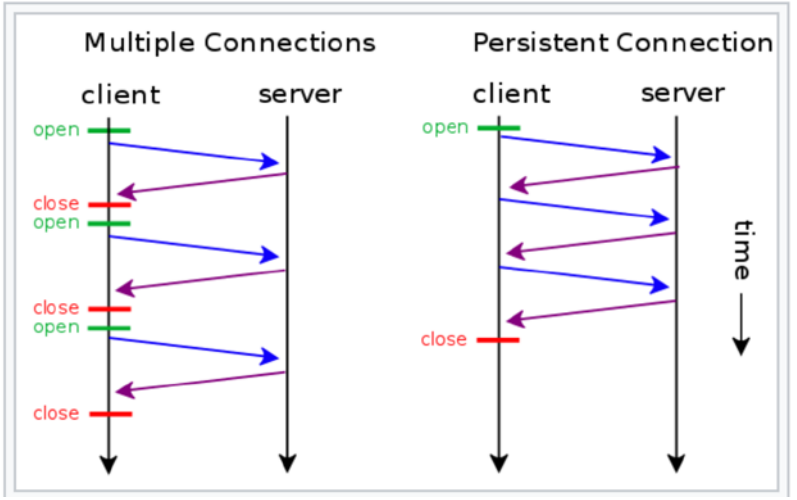
HTTP erlaubt **multiple Request/Response-Zyklen** innerhalb einer **„persistenten Verbindung“**:

- Um TCP-Überlastung zu vermeiden, werden zwischen Client und Server maximal zwei persistente Verbindungen aufgebaut
- Werden aufgebaut, indem Nachrichtenkopfs gesetzt wird:
  - Connection: **keep-alive**
- Server, die persistente Verbindungen unterstützen, antworten im Nachrichtenkopf des Response`s ebenfalls
  - Connection: **keep-alive**
- Client beendet HTTP-Session indem beim letzten HTTP-Request die Option „close“ im Nachrichtenkopf gesetzt wird
  - ansonsten wird Verbindung offen gehalten und erst durch Timeout geschlossen; dies kann zu Überlast führen!

Persistente Verbindungen sind Standard ab HTTP/1.1

Vorteile

- Effizientere Nutzung der Betriebsressourcen (CPU, Speicher) durch begrenzte Anzahl simultaner Verbindungen
- Effiziente Bandbreitenauslastung (weniger unnötige TCP-Pakete)
- Verzögerungen werden für eingebettete Ressourcen reduziert (keine weiteren TCP-Handshakes notwendig)
- Persistente Verbindungen erlauben **HTTP-Pipelining**

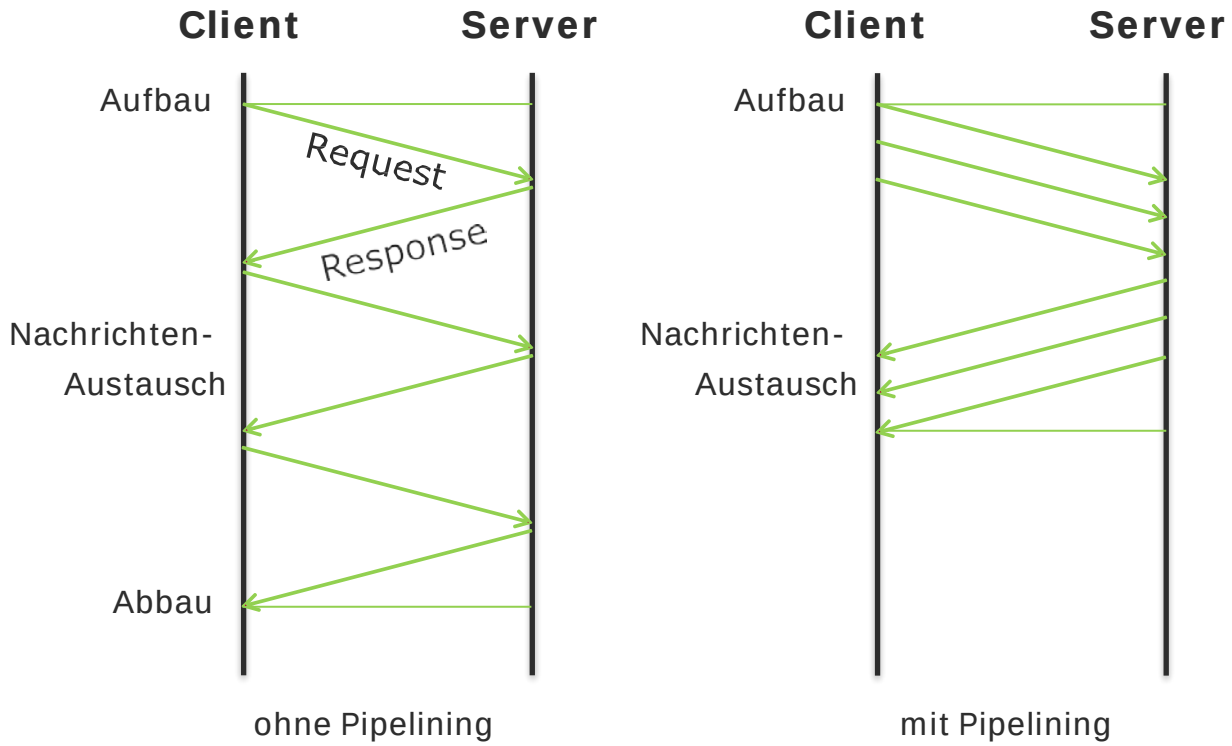


# HTTP-Pipelining

HTTP/1.1 erlaubt Client, über eine offene TCP-Verbindung seine Anfragen an einen Server zu parallelisieren, ohne dass vorherige Antworten abgewartet werden müssten:

HTTP-Pipelining

- Pipelining ermöglicht Übertragungsbeschleunigung, vor allem bei Verbindungen mit hoher Latenz
- Es dürfen nur Anfrage-Sequenzen gepipelined werden, die ohne frühere Anfragen nicht notwendig sind, z.B.
  - Sequenzen von GET- oder HEAD-Requests können immer mit Pipelining verwendet werden
- Anfang jedes HTTP-Request muss explizit klargestellt werden durch Angabe seiner Länge im Headerfeld: **`content-length`**
- Obwohl die meisten Browser HTTP-Pipelining unterstützen, ist es dennoch bei den meisten Browsern defaultmäßig deaktiviert



# Caching

Viele angeforderte Ressourcen verändern sich nur selten

- Nutzungseffizienz des WWW lässt sich drastisch erhöhen, wenn wiederholter Datentransfer gleicher Informationsressourcen mit Hilfe intelligenter Zwischensysteme – Caches – vermieden wird
- Caches agieren als drittes Element in der Client-Server-Interaktion:
  - Alle Browser-Anfragen an einen WWW-Server werden über einen zwischengeschalteten Cache geleitet
  - Cache speichert für begrenzte Zeit die jeweiligen Antworten
  - Wird Informationsressource erneut angefordert, leitet zwischengeschalteter Cache die Anfrage nicht an Server weiter, sondern beantwortet sie direkt
- Caching reduziert drastisch Kommunikationsaufkommen und Serverlast

Cache kann verschieden platziert sein:

- Clientseitiger Cache – Cache ist beim Client und speichert Antworten auf Clientanfragen. Beim Laden von Unterseiten der aktuellen Webseiten werden viele (geteilte) Ressourcen (Bilder, CSS, JavaScript,…) wiederverwendet -> Diese geteilten Ressourcen können einfach gecacht werden
- Eigenständiger Cache – Cache ist logisch selbständig zwischen Browser und Server platziert, z.B. auf Gateway, das Intranet mit Internet verbindet. Client muss entsprechend konfiguriert werden, damit er den Cache nutzen kann -> mehrere Clients verwenden ein Gateway und können Cache gemeinsam nutzen
- Serverseitiger Cache – Cache ist beim Server. Cache speichert vom Server verschickte Antworten und liefert diese bei späteren gleichlautenden Anfragen selbständig aus

Cache-Konsistenz:
Cache darf keine veralteten Ressourcen (stale) ausliefern. Muss deshalb entscheiden können, ob angeforderte, bereits zwischengespeicherte Ressource noch gültig ist

- Cache-Hit – angeforderte Ressource ist zwischengespeichert und gültig – Cache kann Anfrage direkt bedienen
- Cache-Miss – angeforderte Ressource ist entweder nicht zwischengespeichert oder ist zwischengespeichert und stale – Cache muss Anfrage an Origin-Server weiterleiten
- Problem: Wie kann Cache-Konsistenz sichergestellt werden?
- Ist die Cache Ressource gültig? - Übereinstimmung zwischen Originaldokument und im Cache zwischengespeicherter Variante?

-> Lösung: Dokumente werden mit Zeitstempeln und Verfallsdaten versehen …

# Cache-Header

Das wichtigste Nachrichtenkopf-Feld für den Cache ist: **`cache-control`**

Beispiel: **`cache-control: max-age: 3600`**

- Wichtige **`cache-control`**-Einstellungen
  - max-age: Definiert Zeit in Sekunden, nach der gecachte Ressource ungültig wird und neu vom Server angefordert werden muss
  - no-cache: Mit Setzen von cache-control auf no-cache wird Cache mitgeteilt, die Ressource bei jeder Anfrage neu zu validieren
- no-store verhindert, dass Cache die Ressource speichert Eine ältere HTTP-Cache Funktionalität ist das expires-Feld
- Definiert Zeitpunkt, nachdem Ressource ungültig wird
- Setzen von **`max-age`** in den **`cache-control-`**Einstellungen überschreibt **`expires`**-Einstellungen

Content-Revalidierung

- Client muss gecachte Ressource erneut laden, wenn:
  - maximales Alter **`max-age`** erreicht ist
  - **`cache-control`**-Einstellungen auf **`no-cache`** oder **`must-revalidate`** gesetzt sind
- Client sendet Zeitpunkt, zu dem Ressource zuletzt angefragt wurde, im Feld **`if-modified-since`**
  - Wenn Ressource innerhalb dieses Zeitraums verändert wurde, antwortet Server mit Status 200 OK und sendet neue Ressource
  - ansonsten wird ein Status 304 Not Modified gesendet und keine Ressource im Nachrichtenrumpf – Daten werden aus dem Cache ausgeliefert)

# World Wide Web
# HTTP - Mechanismen zur Optimierung

STUDIEREN AUF HÖCHSTEM NIVEAU

Prof. Dr. Jürgen Anders, Hochschule Furtwangen
Fakultät Digitale Medien