Local Latency Reduction on the Web: Compressing, Caching and Prefetching

Ausarbeitung zum Seminar „Quality of Internet Services in E-Commerce“ (Wintersemester 2000/2001)

Axel Beckert, abe@cs.uni-sb.de

23. Februar 2001

Abstract

Vortrag und Ausarbeitung stellen verschiedene Techniken vor, mit denen ein Internet Service Provider (ISP) die Qualität seines Service gegenüber den Kunden verbessern kann. Dabei wird im ersten Abschnitt vor allem auf Möglichkeiten zur Reduktion der Wartezeit bei schmalbandigen Client-Anbindungen (z. B. per Modem) eingangen. Im zweiten Abschnitt werden dagegen client-unabhängige Techniken um die Trefferquoten in Proxy-Caches zu erhöhen behandelt.

Inhaltsverzeichnis

1 Motivation

Warten im Internet kostet nicht nur Geld sondern vor allem intellektuelle Zeit. Entsprechend herrscht ein großes Interesse an Wartezeitverringerung.

Hohe Wartezeiten und niedrige Byteraten im Web können verschiedene Gründe haben: Einerseits kann die Modem-Anbindung (oder eine andere schmalbandige Verbindung) des Clients zum Rest des Internet der Flaschenhals sein. Entsprechende Gegenmaßnahmen hierzu ist die Reduktion des Datenvolumens auf der Modem-Verbindung durch verschiedenste Techniken.

Andererseits können auch der angefragte Server oder Leitungen überlastet sein (z. B. durch temporär ausgefallene Router oder Leitungen und den daraus folgenden Überlastungen auf schmalbandigeren Ausweichstrecken). Hat sich der Kunde bei seinem ISP über die Telefonleitung eingewählt, so entsteht neben der Verschwendung von intellektueller Zeit auch noch ein weiteres Problem finanzieller Art: Verbindungen über Telefonleitungen werden meist nach Dauer und selten nach Volumen berechnet. Entsprechend kostet Tröpfeln aus der Leitung auch mehr Geld, selbst wenn der Kunde nichts für die Überlastung irgendwo anders kann. Aber gegen die Ursachen von Überlastungen auf dem Server oder irgendwo auf der Strecke zwischen ISP und dem angefragten Server der ISP im Normalfall nichts unternehmen. Trotzdem sieht es für die Benutzer so aus, als würde ihnen der ISP eine schlechte Qualität bieten. Deswegen ist ein wichtiges Ziel für ISPs, die Symptome von verstopften Leitungen im Internet zu verringern und damit die Qualität der eigenen Dienstleistung zu erhöhen. Die bisher effektivste Methode hierzu ist Chaching mittels einem oder mehreren Proxies.

2 Traffic-Reduktion zwischen Client und Proxy

Ansatzpunkte zur Traffic-Reduktion zwischen Client und Proxy gibt es viele:

- Großer Cache auf Client-Seite. Dies kann der Browser-Cache sein, auch ein lokalen Proxy-Cache[1].
- Filterung von nicht gewünschten Inhalten durch den Benutzer. Hier ist weniger Selbstzensur gemeint, sondern das Filtern von Werbebannern, Hintergrundmusik oder -bildern, sowie anderen für den Benutzer unnötigen Inhalten.

  Beispiel: Der Proxy „JunkBuster“[2] fängt die Requests von Werbebanner und anderen Dateien ab, in dem er die URL gegen entsprechende reguläre Ausdrücke matcht.
- Komprimierung von HTML auf Anwendungsebene, z. B. durch HTTP/1.1 Transfer-Encoding oder Content-Negotiation. Übliche Komprimierungsverfahren sind hier „deflate“ und „gzip“.
- Verwendung von durch Compilierung komprimierte Markup-Sprachen.
- Δ-Komprimierung, dem Übertragungen von „diffs“ geänderter Dateien zwischen ISP- und lokalem Cache.
- Client-Initiated Prefetching, z. B. „Webbeschleuniger“, die gerade angeschaute HTML-Seiten parsen und darin enthaltene Links und Bilder bereits beginnen, herunterzuladen, bevor der Benutzer sie anfragt.
- Proxy-Initiated Prefetching (Pushing): Der Proxy versucht, das oder die nächsten Dokumente vorherzusagen. Auf Client-Seite sitzt ebenfalls ein Proxy, der als Browser-Cache fungiert und auf die Push-Anfragen des Proxys auf der anderen Seite der Leitung reagiert. Alternativ kann der ISP-Proxy-Cache dem Client-Cache über HTTP-Header Prefetching-Empfehlungen mitteilen, sodaß kein Push-Protokoll implementiert werden muß.

2.1 Komprimierung

Da es sich bei HTML um Text mit vielen sich wiederholenden Zeichenketten (insbesondere den Tags) handelt, läßt er sich gut komprimieren. Dies gilt insbesondere

[1] Die meisten der im folgenden genannten Techniken gehen von einem lokalen Proxy-Cache aus, um browserunabhängig zu bleiben.

[2] verfügbar für Unix & Windows unter http://www.junkbuster.com/.

in der heutigen Zeit, wo das Layout von HTML-Seiten mit unsichbaren Tabellen gang und gäbe ist. Noch stärker trifft dies aber auf XML-Dokumente zu, da XML-Dokumente üblicherweise selbstbeschreibende (und damit längere) Tags haben und außerdem gegenüber HTML-Dokumenten meist wesentlich feiner strukturiert sind.

Der Vorteil der Komprimierung wird allerdings dadurch, daß Modem-Verbindungen selbst schon komprimieren, etwas abgeschwächt und der Effekt insgesamt etwas verringert.

Ebenfalls ein Nachteil von Komprimierung ist der erhöhte CPU-Bedarf, dies ist aber heutzutage wesentlich weniger relevant als noch vor ein paar Jahren.

2.1.1 HTTP Transfer- und Content-Encoding

HTTP/1.1 unterstützt neben der Übertragung komprimiert gespeicherter Dokumente (via *Content Negotiation*) auch die on-the-fly Komprimierung nur für die Übertragung der Daten (HTTP-Header `Transfer-Encoding`). Übliche Komprimierungsverfahren sind hier „deflate“ und „gzip“.

Eine weitere Einschränkung von Komprimierung findet sich in der Verteilung von Dateiformaten im Web: Ein Großteil des Traffics sind (leider) Grafiken (GIF, JPEG), die zum Teil selbst schon hochkomprimiert sind (JPEG, PNG). Auch dies schmälert wieder den Effekt der Komprimierung Effekt geringer.

Reduziert das Übertragungsvolumen laut [Fan, Cao, Lin & Jacobson, 1999][3] um durchschnittlich 25%.

2.1.2 Compilierte (binäre) Versionen von Markup-Sprachen

Eine andere Möglichkeit, Markup-Sprachen zu komprimieren, ist, sie in ein äquivalentes, binäres Format umzuwandeln (zu compilieren). Dabei wird das Markup und wenige (vordefinierte) häufig vorkommende Strings wie z. B. „`http://www.`“ durch bestimmte Bitfolgen repräsentiert.

Gegenüber der im vorherigen Abschnitt beschriebenen normalen Komprimierung besteht der Nachteil, daß der textuelle Inhalt der Seiten nicht mitkomprimiert wird und auch nur Markup-Sprachen komprimiert werden können. Der Vorteil dagegen ist, daß compiliertes Markup zum Parsen nicht dekomprimiert werden muß, also gut für rechenschwache Clients geeignet ist.

[3] Nach [Mogul, Douglis, Feldmann & Krishnamurthy, 1997].

Da bei diesem Verfahren vor allem Markup komprimiert wird, eignet es sich ebenfalls sehr gut für stark Tabellen-durchsetzte HTML-Seiten sowie XML-Dokumente. Allerdings muß das binäre Format für jeden DTD neu definiert werden und lohnt sich deshalb nur für DTDs, deren Anwendungen weit verbreitet sind.

Aktuelle Anwendung sind WAP-Seiten. Sie werden in compilierter Form (Compiled Wireless Markup Language, WMLC) übertragen[4] und interpretiert, da WAP-Clients üblicherweise eine sehr schmalbandige Anbindung, einen sehr kleinen Cache und eine geringe Rechenkapazität besitzen.

2.1.3 Δ-Komprimierung

Unter Δ-Komprimierung wird verstanden, daß bei einer Anfrage nach geänderten Dateien anstatt der gesamten Datei nur die Unterschiede zu der Version, die noch im lokalen Cache vorhanden ist, übertragen werden.

Δ-Komprimierung funktioniert allerdings nur, wenn sowohl auf lokaler als auch auf ISP-Seite ein sehr großer Cache vorhanden ist: Der lokale Cache ist notwendig, damit gegebenenfalls ältere Versionen von Dateien zum Vergleichen da sind und der Cache auf ISP-Seite muß nicht nur die aktuellste Version eines Dokumentes gespeichert haben, sondern auch alle älteren Versionen, die der Proxy einmal zurückgegeben hat, damit der zu sendende „diff“ errechnet werden kann.

Δ-Komprimierung reduziert das Übertragungsvolumen geänderter Objekte laut [Fan et al., 1999][5] um durchschnittlich 88% und schafft bis zu 14,6% Verzögerungsverringerung.

2.2 Proxy-Initiated Prefetching [Fan et al., 1999]

2.2.1 Vorteile gegenüber Client-Initiated Prefetching

- Der Proxy kann das Verhalten von sehr vielen Benutzern verwenden, um das Verhalten eines Benutzers vorauszusagen.
  - Einerseits kann dies zu einer Verringerung der Genauigkeit führen, da jeder Benutzer sich unterschiedlich verhält.
  - Andererseits kann der Algorithmus so auf eine wesentlich größere Datenbasis zurückgreifen, was die Wahrscheinlichkeit, daß ihm überhaupt Daten zur Verfügung stehen, wesentlich erhöht wird.

[4] Meist geschieht die Compilierung erst im Gateway des Providers.
[5] Nach [Mogul et al., 1997].

  - Außerdem sollte sowieso der Browser-Cache einen Großteil der wiederholten Zugriffe eines einzelnen Benutzers abfangen. Dies heißt auch, daß die Vorhersagen für Prefetching sich vor allem um Dokumente drehen, die der Benutzer vorher noch nicht betrachtet hat, was bei einer Datenbasis aus den Zugriffen vieler Benutzer wesentlich einfacher ist, als bei der Datenbasis des Benutzers, für den eine Vorhersage getroffen werden soll.
  - Ein weiterer Vorteil einer Datenbasis vieler Benutzer ist, daß dadurch einfach und schnell auch wesentlich weiter (als Tiefe 1) in die Zukunft geschaut werden kann.
  - Ebenso können die nicht unwichtigen Co-Referenzen[6] wesentlich leichter anhand statistischer Daten beachtet werden als durch Analysieren der Linkstruktur.

- Der Proxy muß deshalb auch nicht das gerade angefragte Dokument parsen, sondern kann auf eine interne, gegebenenfalls effiziente Datenstruktur zum Berechnen der Vorhersagen zurückgreifen.
- Das Prefetching wird nur auf Dokumente angewendet, die der Proxy im Cache hat, die also auf jeden Fall sofort verfügbar sind und es aufgrund der kurzen (in Hops gerechnet) Leitung zum Proxy sehr unwahrscheinlich ist, daß eine Leitung zwischen Client und Proxy überlastet oder defekt ist..
- Außerdem wird so durch Prefetching ausschließlich im (meistens nicht ausgelasteten) LAN zusätzlicher Traffic erzeugt, nicht aber im Rest des Internets.

2.2.2 PPM Prefetching Algorithmus

Der von Fan et al. (1999) vorgeschlagene PPM-Algorithmus (Prediction by Partial Matching) kennt folgende Parameter:

- *PrefixDepth*: Anzahl der vergangenen Zugriffe, die zur Analyse verwendet werden
- *SearchDepth*: Anzahl der Zugriffe, die der Algorithmus versucht, in die Zukunft zu schauen

[6] Beispiel Suchergebnis einer Suchmaschine: Zuerst schaut man sich das erste gefundene Dokument an, wenn das nicht die gewünschte Information enthalten hatte geht man in der History des Browsers einen Schritt zurück (RAM-Cache) und schaut sich das nächste gefundene Dokument an, etc.

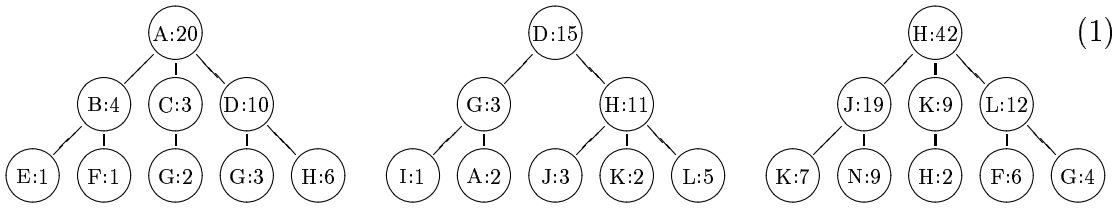
- *Threshold*: Wahrscheinlichkeit, ab der Prefetching veranlaßt wird

Funktionsweise:

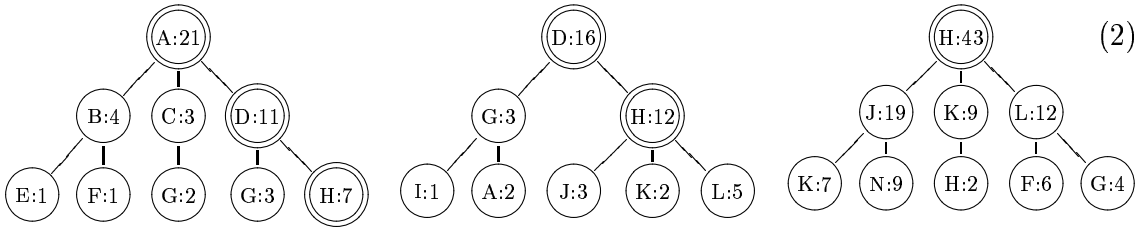
1. Bei jedem Zugriff wird die History-Struktur aktualisiert. Dies wird in den Bäumen der Formeln 1 und 2 (nächster Abschnitt) näher erläutert.
2. Der Algorithmus vergleicht das letzte, die beiden vorletzten, ... und die *PrefixDepth* letzten zugegriffenen Dokumente des Benutzers mit den statistischen Daten in einer Menge von Bäumen und sammelt die Werte aller Knoten, die innerhalb einer Tiefe von *SearchDepth* Schritten in den entsprechenden Teilbäumen vorkommen.
3. Anschließend werden alle Werte durch die Wert des Knotens, in dessen Teilbaumes sie gefunden wurden, dividiert. Damit bekommt man die Wahrscheinlichkeit jedes Knotens.
4. Beschreiben mehrere Knoten dasselbe Dokument, so werden ihre Werte aufsummiert.
5. Unter diesen Dokumenten werden diejenigen herausgesucht, deren Wahrscheinlichkeit über *Threshold* liegt.
6. Die so gefundenen Dokumente werden absteigend nach der Länge des gematchten Präfixes sortiert, innerhalb einer Präfixlänge nach der Wahrscheinlichkeit. Davon werden die ersten acht Dokumente zum Prefetching vorgeschlagen.

Datenstruktur der für PPM notwendigen History Für jedes Dokument im Cache existiert ein Baum der Tiefe K=*PrefixDepth*+*SearchDepth*, in dem alle K Zugriffe nach einem Zugriff auf das entsprechende Dokument gespeichert wird.

Beispiel: Bäume vor (1) und nach (2) dem Zugriff auf die Dokumentsequenz (A, D, H) bei K=3.



(1)



(2)

Wir nehmen an, daß der Benutzer den Proxy noch nicht benutzt hat und nacheinander auf die Dokumente A, D und H zugreift.

Beim Zugriff auf Dokument A gab es keine vorherigen Dokumente und so wird nur der Knoten A im Baum für das Dokument A um 1 erhöht.

Beim darauffolgenden Zugriff auf das Dokument D wird sowohl der Knoten D am Ende des Pfades A-D im Baum A um 1 erhöht als auch der Knoten D im Baum des Dokumentes D.

Nach dem anschließenden Zugriff auf das Dokument H wird im Baum des Dokumentes A der Knoten am Ende des Pfades A-D-H, im Baum des Dokumentes D der Knoten H am Endes des Pfades D-H sowie der Wurzelknoten H des Baumes für das Dokument H um 1 erhöht.

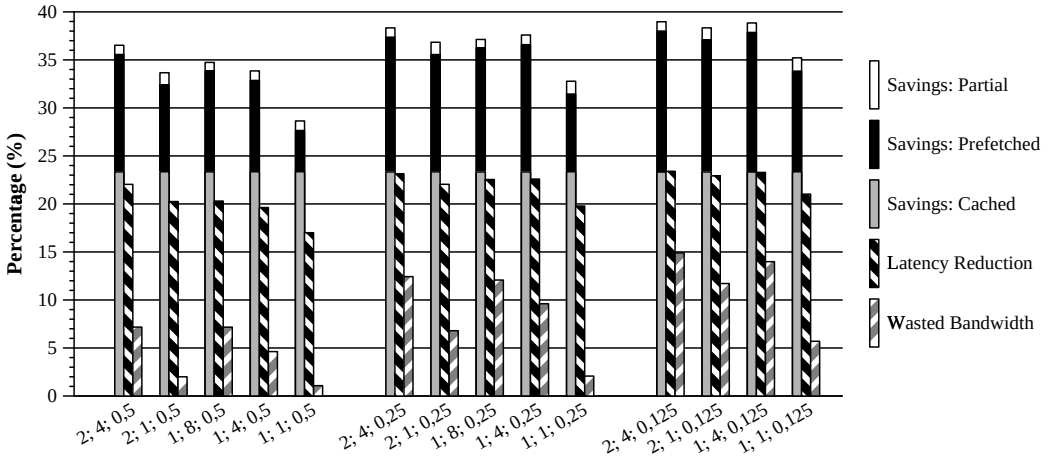


Abbildung 1: Anfrage-Einsparungen, Verzögerungsverringerung und verschwendete Bandbreite des PPM-Algorithmus. Konfigurationen des PPM-Algorithmus dargestellt als Tupel (*PrefixDepth*, *SearchDepth*, *Threshold*).

Potential verschiedener Konfigurationen des PPM-Algorithmus Basierend auf Traces über fünf Tage an der *University of California, Berkeley* wurde die Performance des PPM-Vorhersage-Algorithmus studiert. Es wurden u. a. die Annahmen gemacht, daß jeder Browser einen Cache von 16MB hat und den Least-Recently-Used-Algorithmus verwendet. Weiter wird angenommen, daß der Proxy den Inhalt jedes Browser-Caches kennt und daß bei modifizierten Objekten Prefetching zusammen mit Δ-Codierung verwendet wird.

Weiter wurden folgende (pragmatische) Einschränkungen bezüglich des Prefetching vorgenommen:

- Prefetching wird nur für Dokumente bis zu einer bestimmten Größe (hier: 50 kB) veranlaßt.

  Begründung: Ein Großteil (ca. 95%, Stand 1996[7]) der Dokumente im Web ist kleiner als 50 kB und mit der Größe des per Prefetching heruntergeladenen Dokumentes steigt auch die Wahrscheinlichkeit, daß der Benutzer ein Dokument anfragt, bevor das Herunterladen beendet ist. (Die Wahrscheinlichkeit, daß es sich bei dem per Prefetching geholten Dokument um das richtige handelt, bleibt dabei konstant.)

- Verwendung einer oberen Schranke für die Anzahl der pro Vorhersage zum Prefetching zugelassenen Dokumente (hier: 8).

  Damit möchte man das exzessive Prefetching während langer Idle-Zeiten und damit das Überfluten des Browser-Caches sowie die Überbelastung des Proxys verhindern.

Es wurden Versuche mit *PrefixDepth* gleich 2 und 1, *SearchDepth* gleich 1, 4 und 8 und *Threshold* gleich 0,5, 0,25 und 0,125 gemacht. Die Ergebnisse sind Abbildung 1 graphisch dargestellt. Da die Ergebnisse für *SearchDepth*=8 denen für *SearchDepth*=4 sehr ähnlich waren, werden sie in den meisten Fällen nicht weiter betrachtet.

Allgemeine Beobachtungen:

- Prefetching und Δ-Komprimierung reduzieren die Verzögerung um 17% bis 23,4%.

[7] Obwohl [Fan et al., 1999] von 1999 ist, sind die verwendeten statistischen Daten von 1996, da es nicht sehr viele Traces gibt.

- Ca. 60% der Anfrage-Ersparnis beruht auf Hits im Browser-Cache, ca. 40% auf Prefetching.
- Insgesamt reduzieren 16MB Browser-Cache und Δ-Komprimierung die Verzögerung um 14,4%, Prefetching sorgt für weitere 9%
- Der zusätzliche erzeugte Traffic reicht von 1% bis 15%

Abhängigkeiten von den Parametern:

Threshold: Allgemein verschwendet ein niedriger *Threshold* mehr Bandbreite als ein hoher. Ist *SearchDepth*=1, verringert ein niedriger *Threshold* aber auch die Verzögerung. Trotzdem scheint für *SearchDepth*>1 *Threshold*=0,5 der beste Wert zu sein.

Begründung:

- Bei ansonsten fixen Parametern ergibt ein niedriger *Threshold* mehr mögliche Dokumente.
- Bei *SearchDepth*=1 gibt es nicht sehr viele potentielle Prefetching-Kandidaten und somit ergibt erst ein entsprechend niedriger *Threshold* genügend Kandidaten.

SearchDepth: Die Erhöhung der Suchtiefe verringert die Verzögerung genauso, wie sie mehr verschwendet Bandbreite zur Folge hat. Eine Suchtiefe von 4 scheint sinnvoll, da es darüber kaum noch Änderungen gibt.

PrefixDepth: Die Erhöhung der *PrefixDepth* verringert ebenso die Verzögerung und erhöht die Verschwendung von Bandbreite, jedoch ist dieser Effekt stärker ausgeprägt, wenn der *Threshold* hoch ist. Generell hilft zusätzlicher Kontext.

2.2.3 Potential von Prefetching

Aus Abbildung 2 läßt sich folgendes erkennen:

- Der große Unterschied zwischen *attempted* und *prefetched* weist darauf hin, daß es meist nicht zum Prefetching aller Kandidaten kam, da dafür meist die Idle-Zeit der Benutzer zu kurz war.
- Das Verhältnis zwischen *used* und *prefetched* entspricht der Genauigkeit der PPM-Konfiguration. Sie schwankt zwischen 40% (2; 4; 0,125) und 73% (1; 1; 0,5).

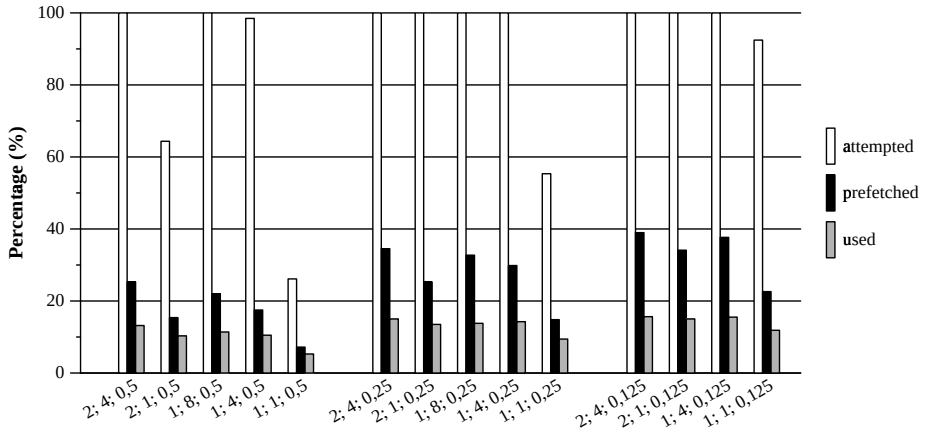


Abbildung 2: Gesamtanzahl an vorgeschlagenen Kandidaten (*attempted*), Anzahl der durchgeführten Prefetching-Vorgänge (*prefetched*), Anzahl der Zugriffe auf Objekte, die per Prefetching in den Browser-Cache kopiert wurden (*used*). Dabei entsprechen 100% der Gesamtanzahl von Anfragen im Trace. Konfigurationen des PPM-Algorithmus dargestellt als Tupel (*PrefixDepth*, *SearchDepth*, *Threshold*).

- Eine Verringerung des *Threshold* verringert immer die Genauigkeit.
- Es gibt keine direkte Korrelation zwischen der Genauigkeit und der Verzögerungsverringerung.

Empfehlungen von [Fan et al., 1999]:

- Ist Verzögerungsverringerung das höchste Ziel, so ist (2; 4; 0,125) die beste Konfiguration.
- Ist Verzögerungsverringerung bei niedriger Bandbreitenverschwendung das Ziel, so ist (2; 4; 0,5) die beste Wahl.
- Ist die Kapazität des Proxys bezüglicher der History-Struktur beschränkt, so sind (2; 1; 0,25) und (1; 1; 0,125) eine gute Wahl.

Aufschlüsselung von prefetched nach Dateitypen

- Ca. 48% GIFs
- Ca. 21% JPEGs

- Ca. 15,5% HTML-Dateien
- Rest andere oder anhand der Traces nicht identifizierbare Dateitypen

Dies legt nahe, daß es sich um eingebettete Grafiken der entsprechenden Webseiten handelte.

Diese Objekte können laut Fan et al. (1999) auch durch andere Methoden, wie z. B. dem Parsen von HTML-Dokumenten gut gefunden werden.

Allerdings macht dies nur dann Sinn, wenn es sich um eingebette Objekte in für Prefetching vorgesehene HTML-Dokumente handelt, da Browser normalerweise eingebette Objekte sofort nach deren Entdecken anfragen und somit für Prefetching uninteressant machen.

Aufschlüsselung der Genauigkeit nach Dateitypen

Ca. 65% GIFs

Ca. 58% JPEGs

Ca. 35% HTML- und andere Dateien

Da PPM aber (wie leicht klar wird) stark vom Umfang der History abhängt und die oben genannten Werte auf Traces von nur 5 Tagen basieren, sollte sich die Genauigkeit verbessern, sofern man auf Daten von mehreren Wochen oder Monaten zurückgreifen kann.

2.3 Varianten von Proxy-Initiated Prefetching

Um möglichst gute Vorhersagen treffen zu können, sollte der Proxy alle Anfragen eines Benutzers mitbekommen, auch die, die im lokalen Browser-Cache (sowohl RAM- als auch Platten-Cache!) landen. Dies wird um browserunabhängig zu bleiben meist über einen lokalen Proxy und einen ausgeschalteten Browser-Cache realisiert.

Weiter sollte der Proxy den Cache-Inhalt jedes Clients kennen, um nicht Dokumente zum Prefetching vorzuschlagen, die bereits im Browser-Cache gespeichert sind. Dies zu handhaben ist jedoch nicht trivial und wird deshalb meist gemieden. Es gibt hierzu aber auch eine Alternative:

2.3.1 Proxy-Suggested Prefetching via HTTP Request Piggyback

Der (Remote-) Proxy teilt dem lokalen Proxy (oder Browser) seine Empfehlungen für Prefetching als Header in der Antwort auf einem HTTP-Request mit (*engl.* „Piggyback“ = „Huckepack“), dieser veranlaßt ein Prefetching nur dann, wenn sich das Dokument nicht im lokalen Cache befindet.

2.3.2 Auswirkungen von vereinfachten Implementationen

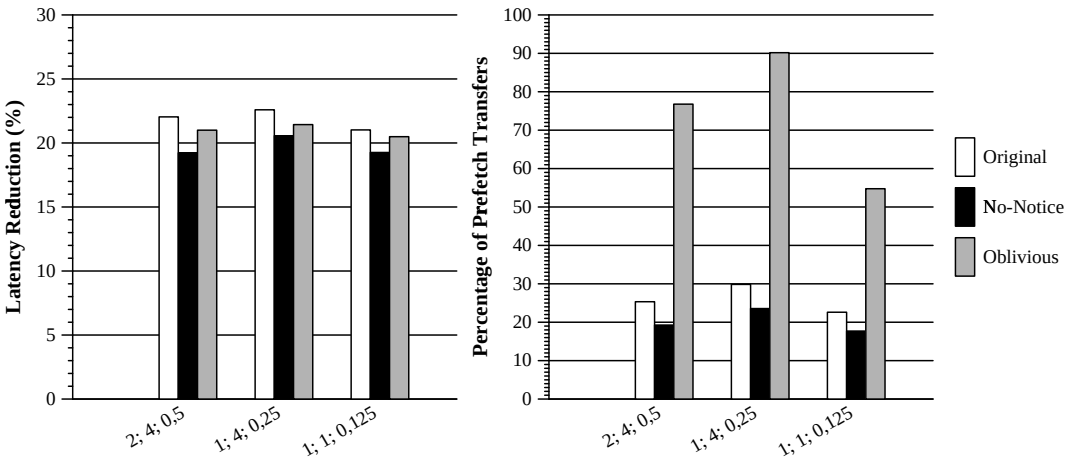


Abbildung 3: Vergleich des ursprünglichen Prefetchings mit Vereinfachungen. Konfigurationen des PPM-Algorithmus dargestellt als Tupel (*PrefixDepth*, *SearchDepth*, *Threshold*).

In Abbildung 3 sind die Auswirkungen zweier verschiedener Vereinfachungen des Prefetchings mit dem PPM-Algorithmus graphisch dargestellt.

Keine Benachrichtigung über Hits im Browser-Cache („No-Notice“)

Grund: Wesentlich einfacher zu implementieren.

Effekt: Prefetching wird weniger häufig initiiert.

Kein Wissen über Inhalt des Browser-Caches („Oblivious“)

Grund: Um das unnötiges Prefetching von Dokumenten zu verhindern, die bereits im Browser-Cache liegen, sollte der Proxy über den Cache-Inhalt aller seiner Clients Bescheid wissen. Dies ist aber aufgrund des zusätzlichen Speicherverbrauchs als auch von der Implementation her schwierig zu handhaben.

Effekt: Es werden wesentlich mehr Dokumente per Prefetching zum Client übertragen.

Folgerung: Ist also die Verschwendung von Bandbreite kritisch, so muß der vorhersagende Proxy den Inhalt des Browser-Caches kennen oder dem Browser die endgültige Entscheidung überlassen, welche der vorgeschlagenen Dokumente wirklich per Prefetching geladen werden soll (HTTP Request Piggyback).

2.4 Zusammenfassung

Prefetching in Verbindung mit großem Browser-Cache und Δ-Encoding zwischen Client und Proxy-Cache kann Wartezeiten merkbar verzögern.

Dazu sollte der Proxy allerdings auch über die Hits im Browser-Cache benachrichtigt werden und die Entscheidung über Prefetching vorgeschlagener Dokumente vom Inhalt des Browser-Caches abhängig sein. Letzteres läßt sich am einfachsten durch Übermitteln der Vorschläge als HTTP-Header in der Antwort auf einen normalen HTTP-Request implementieren.

3 Web Proxy Caching

Caching mittels Web Proxy (z. B. Squid) reduziert einerseits die Verzögerung von häufig angefragten Daten und andererseits das Datenvolumen der Anbindung des ISP zum Internet, da sie meist nur innerhalb eines LANs übertragen werden müssen. Dazu tragen u. a. folgende HTTP-Header bei [Fielding, Gettys, Mogul, Frystyk & Berners-Lee, 1997]:

Cache-Control: Cache oder No-Cache

Pragma: Cache oder No-Cache

ETag: Hash-Wert eines Dokumentes, sollte innerhalb eines Servers eindeutig sein

Expires: Angabe über die Dauer der Gültigkeit eines Dokumentes (Datumsangabe, keine Dauer)

Last-Modified: Angabe über Datum der letzten Änderung. Wird bei dynamischen Dokumenten (dazu gelten SSI-Dokumente, die Kopf- und Fußzeile einfügen lassen!) meist *nicht* gesendet, auch wenn vielleicht auf der Webseite ein Datum der letzten Änderung (z. B. sehr leicht zu realisieren mittels SSI) steht!

3.1 Cooperative Web Proxy Caching

Cooperative Web Proxy Caching ist die Zusammenarbeit von zwei oder mehreren Web Proxy Caches. Grundgedanke hierbei: Es kann immer noch schneller sein, seinen Cache-Kollegen zu fragen, als den eigentlich angefragten Server.

Wolman et al. (1999) haben versucht, anhand Auswertung von gleichzeitigen Traces an der *University of Washington* und bei *Microsoft* die Nützlichkeit von *Cooperative Web Proxy Caching* in der Praxis zu ermitteln.

Dabei wurden vor allem zwei Dinge analysiert:

1. Die Kooperation lauter kleiner Caches in jeder der ca. 200 Teilorganisationen der *University of Washington* (Institute, Fachbereiche, Verwaltung, etc.) mit jeweils kleinen Populationen (bis 1000 Benutzern).
2. Die Kooperation zwei großer Organisationen (eben der *University of Washington* und *Microsoft*) mit jeweils sehr großen Populationen (jeweils mehreren 10 000 Benutzer).

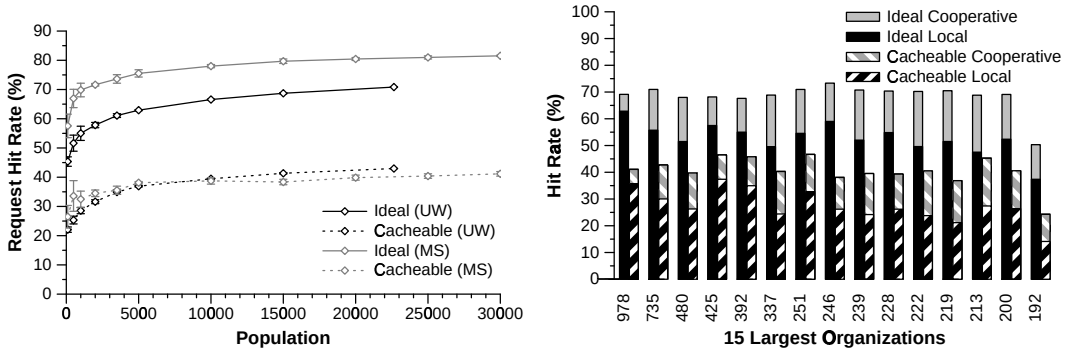


Abbildung 4: Proxy Anfrage-Trefferquoten als Funktion der Größe der Benutzerpopulation sowie lokale und globale Proxy Anfrage-Trefferquoten für die 15 größten Suborganisationen der *University of Washington*

Sie kamen dabei zu folgenden Ergebnissen:

- *Cooperative Web Proxy Caching* ist nur für kleine Organisationen nützlich (in der linkem Grafik von Abbildung 4 sieht man schön das „Knie“ bei ca. 2500 Benutzern) und liegt damit (bei den verwendeten Traces) in einem Bereich in dem es technisch (aber eben meist nicht politisch oder geographisch) auch ein einzelner, großer Proxy-Cache täte.
- Bei großen Organisationen reduziert *Cooperative Web Proxy Caching* die Verzögerung nur marginal, da es eine große Überschneidung zwischen den beliebtesten Dokumenten in den Organisationen (im Beispiel 33% der ersten 1000 Dokumente) gibt und „unbeliebte Dokumente universell unbeliebt sind“.

Dabei sind folgende Unterschiede zwischen dem Benutzerverhaltens an der *University of Washington* und dem bei *Microsoft* aufgefallen:

- Die graue Idealkurve (*Microsoft*) ist um ca. 13% höher als die schwarze (*University of Washington*). D. h., daß das „Document Sharing“ der Benutzer bei *Microsoft* wesentlich höher ist als an der *University of Washington*, sich also die Benutzer bei *Microsoft* wesentlich homogener verhalten als die der *University of Washington* (was auch nicht sehr verwunderlich ist).
- Auf der anderen Seite unterscheiden sich die beiden „Cacheable“-Kurven nur sehr wenig, was zusammen mit der vorherigen Beobachtung den Schluß zuläßt, daß die Anfragen bei *Microsoft* wesentlich häufiger (hier 49%) nicht-cachebare Dokumente (hauptsächlich dynamische Seiten) betreffen als an der *University of Washington* (40%). Das bedeutet, daß die Cache-Trefferquoten bei *Microsoft* trotz der größeren Benutzeranzahl nicht besser sind als die der *University of Washington*.

Obwohl die Theorie diese Ergebnisse (daß eben die Trefferquote eines Cache logarithmisch mit der Anzahl der Anfragen und der Anzahl seiner Benutzer steigt) teilweise schon vorausgesagt hatte, sind sie bezüglich der Nützlichkeit von *Cooperative Web Proxy Caching* mit Vorsicht zu genießen:

- Wie wir gerade gesehen haben, hängen Cache-Trefferquoten sehr von der Homogenität der Benutzer sowie ihrem Verhalten ab. Dies läßt vermuten, daß es auch zu anderen Ergebnissen kommen kann als denen, die hier gezeigt wurden.
- Nach eigener Aussage war [Wolman et al., 1999] die erste praxisnahe Untersuchung von *Cooperative Web Proxy Caching* in großem Umfang. Weiter gingen die Traces nur über eine knappe Woche (6 Tage und 6 Stunden). Die statistische Signifikanz dieser Ergebnisse ist also nicht sehr hoch.

- Diese Ergebnisse gehen von optimalen Bedingungen aus (keine Verwendung des Browser Caches während der Traces, beliebig viel Speicherplatz in allen Proxy Caches). Allerdings wurde ein Unterschied zwischen idealem Caching (alle Dokumente können im Cache gespeichert werden) und Caching von Dokumenten, bei denen Caching möglich ist, gemacht.

3.2 Caching Dynamischer Dokumente

Dynamische Dokumente oder Dokumente, bei denen der Server das am besten zum Client passende Dokument sendet (*Content Negotiation*), machen Probleme beim Caching. Bezüglich *Content Negotiation* ist dieses Problem aber unter zusätzlicher Zuhilfenahme der folgenden HTTP-Header bereits weitestgehend gelöst:

Vary: z. B. Encoding, Type, Language

Content- & Accept-Type: z. B. text/plain, text/enriched, text/html, text/vnd.wap.wml, application/pdf, application/ms-word, etc.

Content- & Accept-Encoding: z. B. gzip

Content- & Accept-Language: z. B. de, en, fr, dk, nl, no

Wirklich dynamische Dokumente sind jedoch außerdem noch von mindestens einem der folgenden Faktoren abhängig:

- Datum
- Uhrzeit (z. B. Wettervorhersage, Nachrichten)
- Cookies (z. B. viele Online-Shops, benutzerabhängige Portalseiten wie Slashdot, MyOpera, MyYahoo! und wie die ganzen MyWeißgottwas-Seiten alle heißen.$\Longrightarrow$ Dies sind insbesondere viel sehr beliebte Seiten.)
- Referer-Header
- lokalen Daten(banken) auf dem Server

Dementsprechend können sie normalerweise nicht im Cache gespeichert werden. Einen Ansatz zur Lösung dieses Problems zeigt der folgenden Abschnitt:

3.2.1 HTML Macro-Preprocessing (HPP) to Support Dynamic Document Caching [Douglis, Haro & Rabinovich, 1997]

Wie bereits gezeigt wurde, sind dynamische Dokumente schlecht cachebar. Allerdings sind auch innerhalb von dynamischen Dokumenten viele statische Elemente (Layout von Tabellen, Menüs, Kopf- und Fußzeilen, etc.), so daß in vielen Fällen das Offenlegen der internen Struktur dynamischer Seiten[8] hilft, deren statische Elemente zu cachen.

Dann müssen nur noch die sich ändernden Daten vom Server geholt werden. Dabei könnte das Macro-Processing bereits im Proxy oder erst im Browser passieren, abhängig davon, ob der Browser HPP unterstützt. Daß dies bei einem Client oder Proxy der Fall ist, kann dem Server z. B. in dem HTTP-Header „Accept-Encoding“ übermittelt werden.

Außerdem können `while`- und `foreach`-Schleifen zur Komprimierung der statischen Daten beitragen, z. B. bei homogenen Auflistungen von Suchergebnissen oder bei Newstickern wie Slashdot.

3.2.2 Resource Description Framework (RDF)

Auch wenn es noch Welten von HPP entfernt ist, so geht das Resource Description Framework (RDF), das von Netscape für sein Portal myNetscape entwickelt wurde und seit 1999 vom World Wide Web Consortium (W3C) weitergeführt wird[9] einen Schritt in diese Richtung: Es erlaubt, Daten bzw. Meta-Daten von Webseiten in einem kompakten, XML-basierten Format anzubieten, die dann z. B. auf einer Webseite nur noch layoutet werden müssen.

Neben Sitemaps, Content Ratings und Bookmarks sind auch die neusten Schlagzeilen eines Newstickerdienstes (wie z. B. Slashdot oder dem Heise-Newsticker) eine sehr beliebte Anwendung des RDF-Formate: Viele Newsticker bieten die Titel sowie die URLs (und gegebenenfalls noch wenige weitere Daten) ihrer Nachrichten in einem reinen Datenformat an, so daß jeder sie in eine andere Webseite (oder auch ein Programm mit Internet-Anschluß) einbinden kann.

Wer dies einmal in Aktion sehen möchte, kann sich z. B. das „Slashdot Cheesy Portal“[10] oder www.GeekBoys.org ansehen. Letztere Webseite bietet nicht anderes als

[8] Durch Darstellung der Abhängigkeiten dynamischer Elemente mittels üblicher Programmierkonstrukte wie `if-then-else`, `while`, `foreach`, etc.

[9] Siehe http://www.w3.org/RDF/ und http://developer.netscape.com/tech/rdf/.

[10] Siehe http://slashdot.org/cheesyportal.shtml

die Schlagzeilen von über 200 verschiedene, RDF-basierte Newsticker an, darunter die aktuellsten Programme bei Freshmeat, die neuesten Geschichten bei SegFault, die Versionsnummern der aktuellsten Linux-Kernels sowie die Schlagzeilen vieler „richtiger“ Newsticker wie Slashdot, Heise/c't, HotWired, etc.

Da (wie GeekBoys.org auf eindrucksvolle Weise zeigt) auf diese Art sehr viele Newstickerdienste auch die Schlagzeilen anderen solcher Dienste anzeigen, ist es nur noch ein kleiner Schritt weiter, innerhalb einer solchen Seite zu sagen: „So, und hier setze bitte die Nachrichten aus `slashdot.rdf` in folgendem Format ein...“ Aktuell muß sich aber noch der derjenige, der diese Daten auf seiner Seite anzeigen möchte, erst die RDF-Datei herunterladen und deren Daten dann eingebunden anbieten.

4 Literatur

Douglis, F., Haro, A. & Rabinovich, M. [1997]. „HTML Macro-Preprocessing to Support Dynamic Document Caching.“ In *Proceedings of USENIX'97.* (Available from http://www.research.att.com/~douglis/)

Fan, L., Cao, P., Lin, W. & Jacobson, Q. [1999]. „Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performances.“ In .

Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H. & Berners-Lee, T. [1997, Januar]. *RFC 2068: Hypertext Transfer Protocol – HTTP/1.1.*

Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. [1999, June]. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.*

Mogul, J. C., Douglis, F., Feldmann, A. & Krishnamurthy, B. [1997]. „Potential benefits of delta encoding and data compression for http.“ In *Proceedings of ACM SIGCOMM'97.* (Available from http://www.research.att.com/~douglis/)

Wolman, A., Voelker, G. M., Sharma, N., Cardwell, N., Karlin, A. & Levy, H. M. [1999]. „On The Scale and Performance of Cooperative Web Proxy Caching.“ In *Operating Systems Review* (Vol. 34, Seite 16-31).