Wykład nr 5: 31-01-2005
Temat: Pośredniki spamiętujące — komunikacja

1 Motywacje

Podstawowe motywacje związane ze stosowaniem pośredników:

- równoważenie obciążenia serwera internetowego,
- zmniejszenie obciążenia sieci przy ściąganiu stron WWW,
- przyspieszenie czasu odpowiedzi przy ściąganiu stron WWW.

Podstawowy schemat architektury sieci pośredników wygląda tak:

2 Podstawowe zagadnienia

Podstawowe zagadnienia związane z sieciami pośredników:

- Jak pośredniki powinny być rozmieszczone?
- Jakie dane powinny zapamiętywać?
- Jakie dane powinny być wymieniane pomiędzy pośrednikami?
- Jakie dane powinny być „wypychane” do sieci?
- W które miejsce dane powinny być „wypychane”?
- Jak usuwać dane z pamięci pośrednika?
- Jak zapewnić spójność danych w pośredniku?

3 Podstawowe protokoły wymiany informacji między pośrednikami

3.1 Protokoły wymiany bezpośredniej

Do protokołów wymiany bezpośredniej — czyli działających na takiej zasadzie: najpierw zapytanie o zasób, a potem odpowiedź być może bezpośrednio z zasobem — należą ICP (ang. Internet Cache Protocol), [WC97] oraz HTCP (ang. Hyper-Text Caching Protocol) [VW00].

Podstawowy sposób działania:

- pośrednik chce się dowiedzieć, czy któryś z jego sąsiadów nie ma poszukiwanego przez niego zasobu;
- w tym celu wysyła do sąsiadów żądanie ICP lub HTCP;
- sąsiad odpowiada, informując, czy posiada dany zasób;
- jeśli jest odpowiednio skonfigurowany, to razem z odpowiedzią wędruje sam zasób (o ile się zmieści w pakiecie odpowiedzi);
- jeśli sąsiad nie jest odpowiednio skonfigurowany lub jeśli zasób jest zbyt duży, to pytający pośrednik wysyła żądanie HTTP do sąsiada z prośbą o zasób.

Główna wada tego rodzaju protokołów to mała skalowalność — nie można w takiej sieci mieć dużej liczby pośredników.

3.2 Protokoły wymiany oparte na filtrach Blooma

Protokoły te działają w taki sposób, że okresowo publikują listę dostępnych u nich zasobów. Ta lista jest podobna trochę do tablicy haszującej, ale ma tę własność, że nie ma gwarancji, że faktycznie dana strona się u pośrednika znajduje, ale są duże szanse, że tak faktycznie jest. Idea ta jest opisywana za pomocą pojęcia filtru Blooma [Blo70] i jej użycie w kontekście sieci komputerowych zostało zasugerowane przez Marais i Bharat [MB97].

Przykładowy protokół stosujący tę zasadę to Cache-Digest [HRW98].

3.2.1 Filtry Blooma

Na *filtr Blooma* składa się para: wektor v złożony z m bitów oraz k niezależnych funkcji haszujących $h_1, \ldots, h_k : U \to \{0, \ldots, m-1\}$ (przy czym rozmiar U — przestrzeni danych wejściowych — jest dużo większy od rozmiaru m filtru Blooma).

Na filtrze Blooma można wykonywać następujące operacje:

- *Inicjalizacja:* polega na ustawieniu wszystkich bitów filtra na 0.
- *Wstawienie elementu:* dla wstawianego zasobu $x \in U$

  - obliczamy wartości funkcji haszujących $h_1(x), \ldots, h_k(x)$ i
  - ustawiamy wskazane przez funkcje haszujące bity wektora v na 1 ($v[h_1(x)] := 1, \ldots, v[h_k(x)] := 1$).

- *Sprawdzenie obecności w filtrze*: aby sprawdzić, czy element $x \in U$ jest w filtrze v:
  - obliczamy wartości funkcji haszujących $h_1(x), \ldots, h_k(x)$ i
  - jeśli dla któregoś i mamy $v[h_i(x)] = 0$, to wartość x nie była wstawiona do filtra,
  - jeśli dla wszystkich i mamy $v[h_i(x)] = 1$, to wartość x znajduje się w filtrze i z dużym prawdopodobieństwem została wstawiona do filtra.

Przykład Dla $m = 5$ i $k = 2$ oraz wartości funkcji haszujących

- $h_1(x) = x \mod 5$
- $h_2(x) = (2x + 3) \mod 5$

mamy

- *Inicjalizacja*

  v : | 0 | 0 | 0 | 0 | 0 |

- *Wstawienie elementu:*

		h_1	h_2	
9	$\mapsto$	4	1	0 1 0 0 1
11	$\mapsto$	1	0	1 1 0 0 1

- *Sprawdzenie obecności elementu:*

		h_1	h_2	
15	$\mapsto$	0	3	nie ma
16	$\mapsto$	1	0	jest (!)

Dla filtrów Blooma zachodzi

$$Pr(\text{filtr pomylił się po } n \text{ operacjach dodawania}) \approx \left(1 - e^{\frac{-k \cdot n}{m}}\right)^k$$

3.2.2 Realizacja Cache-Digest

Używamy tutaj 4 funkcji haszujących dających 32-bitowe wyniki. Funkcje te uzyskane są w ten sposób, że bierzemy MD5 z URL-a zasobu. Wynik jest 16-bajtowy. Dzielimy go na 4 bloki. Oznacza to, że dla tego protokołu

$k = 4$
$m = 2^3 2$

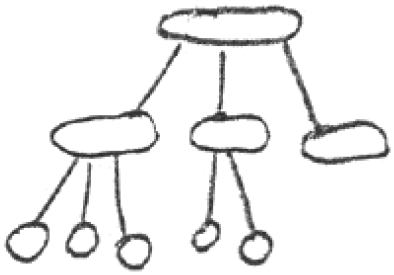
Można policzyć, że zadowalające wyniki wydajnościowe uzyskujemy dla liczby operacji $n \approx 744261117$.

4 Losowe drzewa — unikanie zalania

Problem Czasami jeden zasób staje się nagle bardzo popularny. Jak powinien zostać zorganizowany system pośredników, aby nie spowodować zablokowania dostępu do serwera go oferującego?

Idea Należy rozproszyć zlecenia przez losowe odwoływanie się do pośredników [KLL+97].

Jak to konkretnie zrealizować? Pośredniki będą wypełniać sieć o topologii drzewa, które ma C wierzchołków (C to liczba pośredników powiększona o 1 wierzchołek odpowiadający serwerowi) o stopniu rozgałęzienia d. Wszystkie poziomy drzewa są w pełni zapełnione za wyjątkiem, być może, ostatniego poziomu. Oto przykład takiej sieci dla $C = 9$ i $d = 3$.



Nasz protokół dla ustalonego URL-a przypisuje wierzchołkom drzewa adresy pośredników. Realizowane to jest przez umieszczenie we wszystkich aktywnych (przeglądarkach i pośrednikach) elementach sieci funkcji haszującej $h : U \times \{2, \ldots, C\} \to \mathcal{C}$, gdzie $\mathcal{C}$ jest zbiorem adresów pośredników, zaś U zbiorem URL-i.

Dodatkowo jeszcze wprowadzamy parametr q — pośrednik zapamiętuje zasób, gdy ujrzy go q razy.

Czynności wykonywane w ramach działania protokołu wyglądają tak:

Przeglądarka: Gdy przeglądarka chce się odwołać do zasobu z, to wybiera losowo dowolny liść w drzewie i przypisuje wierzchołkom drzewa konkretne maszyny ze zbioru $\mathcal{C}$ zgodnie z pewną funkcją haszującą h, a następnie wysyła żądanie do maszyny przypisanej wybranemu liściowi. Żądanie składa się z:

- adresu przeglądarki,
- adresu ściąganego zasobu z,
- ścieżki od wybranego liścia do korzenia (implicite),
- wyniku działania h dla z i drzewa (implicite).

Pośrednik: Gdy pośrednik otrzymuje żądanie, to sprawdza, czy zasób, którego żądanie dotyczy znajduje się w pamięci oraz, czy właśnie jest ściągany. Gdy zachodzi pierwsza sytuacja, to podaje zapamiętany zasób. Gdy zachodzi druga, to czeka na ściągnięcie zasobu i podaje otrzymany zasób.

Jeśli zasobu nie ma w pamięci, to zwiększa licznik zasobu. Jeśli zasób nie jest on ściągany, to prosi następną maszynę na ścieżce o z. Gdy licznik osiągnie q, to zasób po ściągnięciu jest zapisywany w pamięci pośrednika. W każdym wypadku zasób jest przekazywany do komputera, który go zażądał.

Serwer: Po otrzymaniu żądania zasobu odsyła go do maszyny, która o niego prosiła.

4.1 Własności protokołu drzew losowych

- Opóźnienie przy dostarczaniu zasobu jest nie większe niż

$$2\delta\lceil \log_d C\rceil,$$

  gdzie δ to ograniczenie górne na opóźnienie w komunikacji między poszczególnymi maszynami.

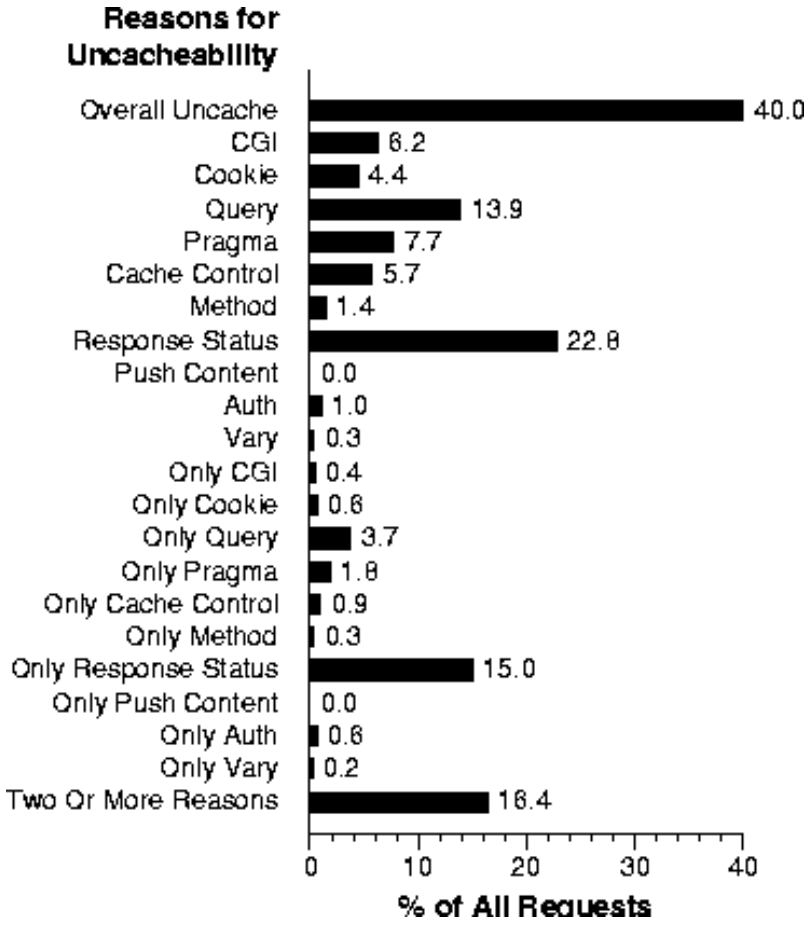
- Dla dowolnego N prawdopodobieństwo, że wskazany pośrednik dostanie żądań więcej niż

$$\rho\Big(2\log_d C + O\Big(\frac{\log N}{\log\log N}\Big)\Big) + O\Big(\frac{dq\log N}{\log\big(\frac{dq}{\rho}\log N\big)} + \log N\Big)$$

  jest mniejsze niż $\frac{1}{N}$, gdzie $\rho = \frac{R}{C}$

5 Powody niespamiętywania zasobów

Oto wykres przedstawiający powody niezapamiętywania zasobów przez pośredniki [WVS+99]:



6 Co robić, aby zapamiętywać więcej?

1. Zapamiętywanie aktywnych danych:
   - dane dynamiczne zmieniają się raz na jakiś czas,
   - zwykle tylko część strony zawiera dynamiczne dane,
   - zapamiętywanie w pośredniku par: dane wejściowe, wynik.
2. Aktywne pośredniki:
   - pośrednik zapamiętuje i wykonuje cały serwlet w Javie.
3. Wzmocnienie mechanizmów zapewniania spójności:
   - wysyłanie unieważnień,
   - systemy wynajmowania zasobów.
4. Sprowadzanie z wyprzedzeniem.

Najważniejsze jest jednak, aby budzić świadomość potrzeby budowania serwisów, które są nastawione na spamiętywanie zawartości.

7 Zachowywanie spójności danych

1. Odpowiednie ustawianie czasu życia.
2. Unieważnianie (problemy ze skalowalnością).
3. Wynajmowanie zasobów (konieczne odświeżanie świadectwa wynajmu).

Ciekawy przykład: Skalowalna architektura spójnego WWW [YBS99].

- Pośredniki są zorganizowane w drzewo (a właściwie w DAG).
- W dół drzewa przesyłane są okresowo „tyknięcia”.
- Okres wynajmu T jest mierzony w „tyknięciach”, np. może wynosić 5 „tyknięć”.
- Każdy węzeł okresowo sprawdza czas. Jeśli od ostatniego „tyknięcia” upłynęło więcej niż T czasu, to wszystkie strony pochodzące od węzła, od którego pochodzą te „tyknięcia” są unieważniane.
- Razem z „tyknięciami” przesyłane są w dół drzewa informacje o nieważnych stronach.
- Serwer dołącza się w dowolnym miejscu grafu.
- Pośredniki przekazują informacje o dołączeniu serwera w górę grafu.
- Raz na jakiś czas (razem z „tyknięciami”) serwer wysyła informacje o nieważności stron.
- Ta informacja jest propagowana w górę hierarchii.
- Jeśli za długo ($> T$) nie przychodzą żadne wiadomości z serwera, to jest on usuwany ze struktur danych i unieważniane są jego wszyskie strony.

Literatura

[Blo70] Burton Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, July 1970.

[HRW98] Martin Hamilton, Alex Rousskov, and Duane Wessels. Cache digest specification - version 5. `http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt`, December 1998. distributed by Squid development team.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM Press, 1997.

[MB97] J. Marais and K. Bharat. Supporting cooperative and personal surfing with a desktop assistant. In *Proceedings of ACM UIST'97*, October 1997. Available on-line at `ftp://ftp.digital.com/pub/DEC/SRC/publications/marais/uist97paper.pdf`.

[VW00] P. Vixie and D. Wessels. Hyper Text Caching Protocol (HTCP/0.0). RFC 2756, January 2000. distributed by IETF.

[WC97] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. RFC 2186, September 1997. distributed by IETF.

[WVS+99] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proc. of the 2nd USENIX Conf. on Internet Technologies and Systems*, October 1999.

[YBS99] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 163–174. ACM Press, 1999.