WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI
POLITECHNIKI WROCŁAWSKIEJ

PEER-TO-PEER WEB OBJECTS CACHING PROXY

TOMASZ DRWIĘGA

Praca magisterska napisana
pod kierunkiem
dra Mirosława Korzeniowskiego

WROCŁAW 2013

Spis treści

Wstęp

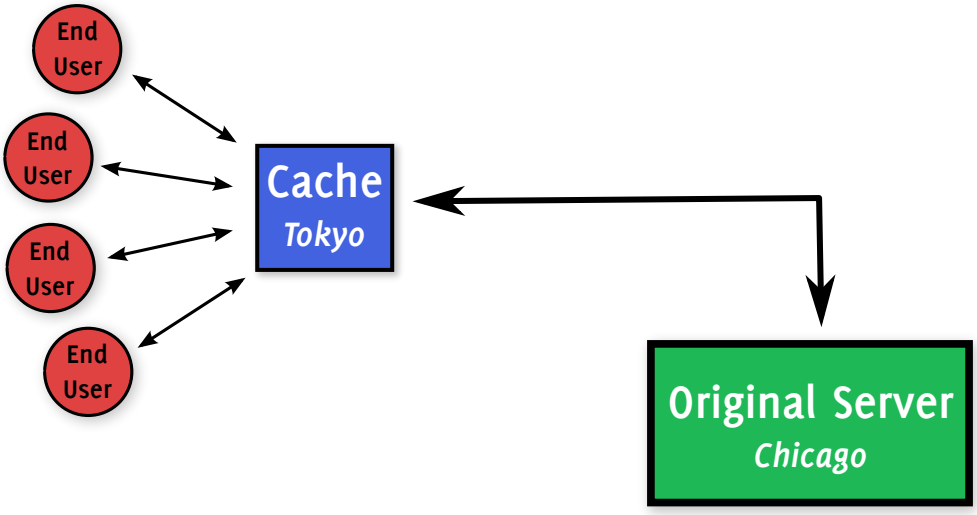
TODO: Opis problemu cachowania (oryginalnego - hot spots), nasilenie związane z efektem slash-dot, multimedia

1 Historia cache'owania

Wraz z rosnącą liczbą użytkowników Internetu w latach dziewięćdziesiątych zaczęły pojawiać się problemy z dostępem do pewnych zasobów. Duża liczba odwołań do popularnych stron w krótkim czasie powodowała znaczące obciążenie serwerów, które dany zasób posiadały. Czasami z powodu nadmiernego zalania[1] żądaniami serwer nie był w stanie obsłużyć wszystkich, przez co strona stawała się niedostępna.

1.1 Serwery cache'ujące

Jako rozwiązanie problemu „gorących punktów" (ang. *hot spots*) pojawiły się serwery cache'ujące. Rysunek 1 przedstawia wprowadzenie transparentnego, lokalnego cache'a i jego relację z serwerem docelowym. Żądania zasobów wychodzące od użytkowników końcowych są w pierwszej kolejności obsługiwane przez serwer cache'ujący, który odpowiada zapamiętanym zasobem lub kontaktuje się z serwerem docelowym i zapamiętuje odpowiedź.



Rysunek 1: Przykład lokalnego serwera cache'ującego. Zamiast wielokrotnie odwoływać się do docelowego serwera, który zawiera daną stronę możemy zapamiętać ją na serwerze cache'ującym. Takie rozwiązanie niesie ze sobą zalety zarówno dla administratorów serwera w Chicago, jak i użytkowników sieci w Japonii: mniej żądań skutkuje mniejszym obciążeniem serwera, a „lokalność" serwera cache'ującego zmniejsza opóźnienia i zwiększa szybkość transferu zasobu.

Wprowadzenie cache'owania niesie ze sobą szereg zalet nie tylko dla administratorów serwerów zawierających zasoby. Serwery znajdujące się na granicy sieci lokalnej z Internetem (jak na rysunku

[1] ang. *flooded*, *swamped*

1) oferują użytkownikom tej sieci lepszy transfer i mniejsze opóźnienia w dostępie do popularnych zasobów. Ponieważ łącze wychodzące z sieci ma ograniczoną przepustowość, cache pozwala również na jego oszczędniejsze wykorzystanie, a tym samym poprawę jakości dostępu do Internetu.

1.2 Rozproszony i hierarchiczny cache

Rozwiązanie przedstawione w rozdziale 1.1 pozwala na odciążenie serwera docelowego. Zastanówmy się jednak co będzie się działo w przypadku zwiększania liczby użytkowników korzystających z serwera cache'ującego. Duża liczba żądań może spowodować dokładnie taką samą sytuację jak w przypadku serwera docelowego - cache zostanie zalany i nie będzie w stanie obsłużyć wszystkich zapytań.

W 1995 roku Malpani i inni [6] zaproponowali metodę, w której wiele serwerów cache'ujących współpracuje ze sobą w celu zrównoważenia obciążenia. W ich propozycji klient wysyła żądanie do losowego serwera należącego do systemu. W przypadku, gdy serwer posiada określony zasób odsyła go w odpowiedzi, w przeciwnym razie rozsyła to żądanie do wszystkich innych serwerów. Jeżeli żaden z cache'ynie zawiera zasobu, to żądanie jest przesyłane do serwera oryginalnego. Niestety wraz ze wzrostem liczby serwerów, należących do systemu liczba przesyłanych między nimi wiadomości bardzo szybko rośnie i cały system staje się zawodny.

W ramach projektu Harvest [1] A. Chankhunthod i inni [2] stworzyli cache hierarchiczny. Rozwiązanie to pozwala łączyć kilka serwerów w system, który można skalować w zależności od liczby użytkowników, których ma obsługiwać. Grupy użytkowników łączą się z serwerami, będącymi liśćmi. Przychodzące żądania są obsługiwane najpierw przez serwery na najniższym poziomie, w przypadku gdy te serwery nie mają danego zasobu w cache'u decydują czy pobrać go z serwera docelowego, czy odpytać serwery z tego samego i wyższego poziomu. Decyzja podejmowana jest na podstawie opóźnień do obu serwerów.

W praktyce, w takim systemie serwery na wyższych poziomach muszą obsługiwać dużą liczbę żądań od dzieci przez co stają się wrażliwe na zalanie oraz składują duże ilości danych [8].

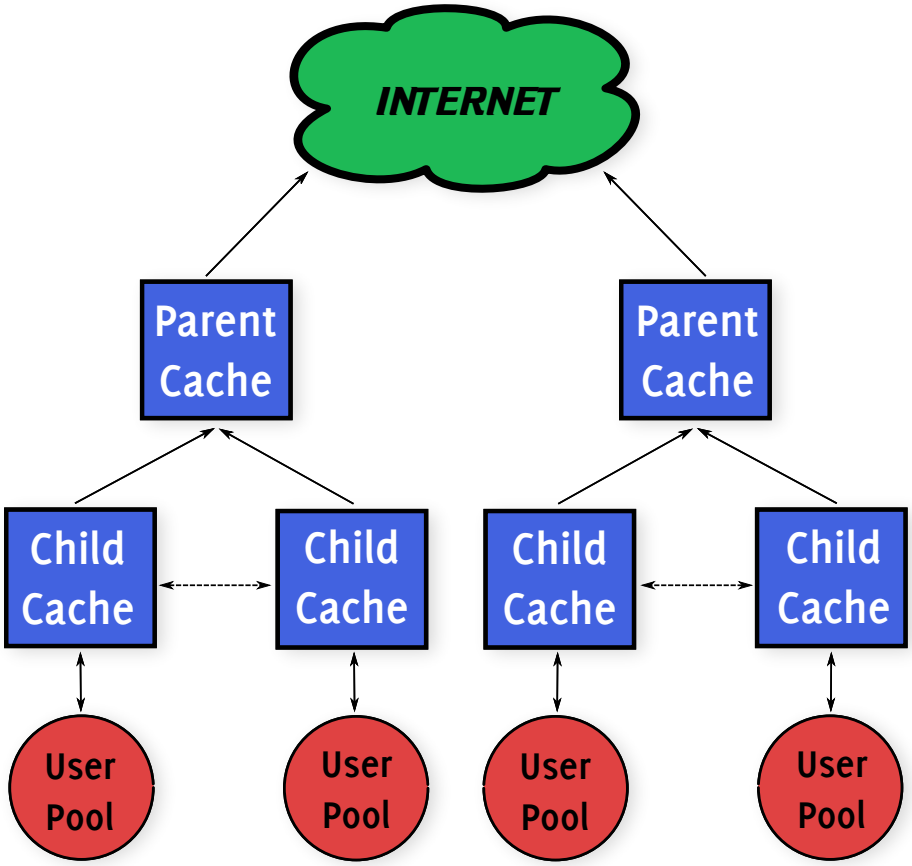
Kolejnym zaproponowanym udoskonaleniem jest cache rozproszony. Rozwiązanie zachowuje hierarchiczną strukturę w formie drzewa, ale tym razem jedynie serwery w liściach zajmują się cache'owaniem zasobów, a pozostałe serwery stanowią indeks, w którym zapamiętywana jest informacja czy zasób znajduje się w systemie oraz jaki serwer go przechowuje.

Jak pokazały eksperymenty [8] rozproszony cache oferuje zbliżoną wydajność do cache'u hierarchicznego rozwiązując dodatkowo problemy związane z obsługą dużej liczby żądań przez serwery na wyższych poziomach.

1.3 Dynamiczna restrukturyzacja

Przedstawione w rozdziałach 1.1 i 1.2 metody cache'owania nie są przystosowane do działania w warunkach, w którym obciążenie czy liczba użytkowników obsługiwanych przez system się zmienia. Wraz z rosnącą liczbą żądań nie jest możliwe łatwe poprawienie wydajności przez dołożenie dodatkowych serwerów cache'ujących. Wprowadzenie zmian w systemie wymaga zmian w konfiguracji poszczególnych serwerów. Dodatkowo awarie pojedynczych węzłów mogą spowodować nieprawidłową pracę cache'u.

W celu ułatwienia dodawania i usuwania serwerów do systemu możemy spróbować innego podejścia do problemu. Podstawowym zadaniem węzłów wewnętrznych w systemie cache'u rozproszonego jest utrzymywanie indeksu przechowywanych zasobów i lokalizacja serwera, który dany zasób obsługuje. Zatem jeżeli klient mógłby określić, który serwer cache'ujący jest odpowiedzialny za obsługę



Rysunek 2: Połączenie kilu serwerów cache'ujących w system hierarchiczny. Każdy z serwerów znajdujących się w liściach obsługuje określoną liczbę użytkowników. Zapytania o zasoby, które nie znajdują się w cache'u wysyłane są do rodzica.

obiektu, wtedy cała infrastruktura związana z indeksowaniem byłaby niepotrzebna.

Naiwnym rozwiązaniem stosującym takie podejście jest przypisanie zasobów do konkretnych serwerów. Przyjmijmy za identyfikator zasobu jego adres. Możemy teraz przypisać wszystkie zasoby w obrębie jednej domeny do pewnego serwera. Na przykład dysponując 24 serwerami możemy przypisać wszystkie domeny zaczynające się na „a" do serwera pierwszego, na „b" do drugiego itd. Oczywiście pozwoli to na łatwe określenie, który serwer należy odpytać o dany zasób, ale rozwiązanie to ma dwie poważne wady. Po pierwsze rozwiązanie nie jest skalowalne - niezdefiniowane jest jakie zasoby miałby obsługiwać dodatkowy serwer, a po drugie obciążenie na serwerach jest nierównomierne.

Możemy jednak spróbować rozwiązać problem przydziału zasobów do serwerów nieco inaczej. Niech n oznacza liczbę serwerów, a R adres pewnego zasobu, wtedy:

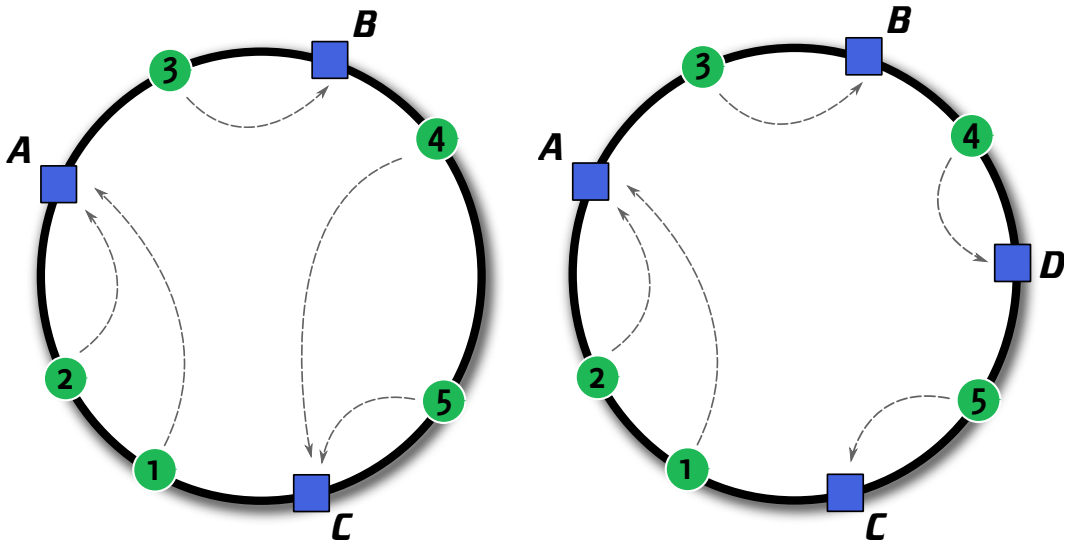
$$S \equiv hash(R) \bmod n,$$

gdzie S oznacza indeks serwera odpowiedzialnego za zasób R, a *hash* jest funkcją haszującą. Dzięki właściwościom funkcji haszujących gwarantujemy, że zasoby każdy z serwerów jest odpowiedzialny za równą część składowanych zasobów. Niestety dalej nie jest rozwiązany problem skalowalności. Dodanie $n+1$ serwera powoduje, że wszystkie zasoby zostaną przypisane w inne miejsca.

1.3.1 *Consistent Hashing*

Podczas dodawania nowego serwera do systemu oczekujemy, że przejmie on od pozostałych równą część zasobów, aktualnie znajdujących się w cache'u. Dodatkowo nie chcemy, aby wartości w cache'ubyły przesuwane pomiędzy „starymi" serwerami. Taką własność gwarantuje mechanizm *Consistent Hashing*, opracowany przez Kargera i Lehmana [4].

Załóżmy, że dysponujemy funkcją haszującą w odcinek $[0, 1]$, za pomocą której każdemu zasobowi i serwerowi przyporządkowujemy punkt na tym odcinku. Dodatkowo, traktujemy odcinek jako okrąg o obwodzie równym 1. W podanym przez Kargera i Lehmana schemacie dany zasób jest obsługiwany przez najbliższy serwer idąc zgodnie z ruchem wskazówek zegara. Rysunek 3 przedstawia przykładowe mapowanie zasobów i ich przyporządkowanie do poszczególnych serwerów. W drugiej części rysunku przedstawiona została zmiana przyporządkowania wynikająca z dodania nowego serwera *D*.



Rysunek 3: Idea *Consistent Hashing*. Zasoby, reprezentowane przez zielone punkty, oraz serwery cache'ujące (kwadraty) mapowane są na punkty na okręgu o jednostkowym obwodzie. W praktyce do reprezentowania punktów na okręgu najczęściej wykorzystywane są klucze binarne o ustalonej długości *m*. Każdy serwer jest odpowiedzialny za obsługę zasobów znajdujących się pomiędzy jego punktem i punktem jego poprzednika. Dodanie nowego serwera *D* powoduje wyłącznie zmianę przydziału zasobu 4.

W praktyce, przesłanie informacji dotyczących zmian serwerów do wszystkich klientów byłoby bardzo kosztowne, ale sytuacja, w której klienci mają różne „widoki" (wiedzą tylko o pewnym podzbiorze serwerów) systemu nie stanowi problemu po wprowadzeniu prostej modyfikacji. Zauważmy, że może dojść do sytuacji, w której jeden z serwerów, z powodu niepełnej wiedzy klientów, będzie obsługiwał więcej żądań od pozostałych. W celu rozwiązania nierównego obciążenia autorzy zaproponowali stworzenie wirtualnych kopii serwerów, tzn. każdy z nich zmapowany jest nie do jednego, lecz kilku punktów na okręgu. Dzięki temu zagwarantowane jest równomierne przyporządkowanie zasobów do serwerów [5].

Pomimo, że pierwotną motywacją powstania *Consistent Hashing* był problem cache'owania zasobów internetowych, schemat znalazł zastosowanie w innych obszarach. W szczególności, leży on u podstaw rozproszonych tablic haszujących, które omówione są w rozdziale 2.

2 Rozproszone tablice haszujące (DHT)

Rozproszone tablice haszujące (ang. *Distributed Hash Tables*, DHT) to systemy pozwalające na przechowywanie par *(klucz, wartość)*, podobnie jak zwykłe tablice haszujące, w dynamicznej sieci zbudowanej z uczestniczących węzłów (*peer-to-peer*). Odpowiedzialność za przetrzymywanie mapowania kluczy do wartości jest podzielona pomiędzy węzły, w taki sposób aby zmiany w liczbie uczestników nie powodowały reorganizacji całości systemu, ale tylko jego drobnej części. Dodatkowo rozproszone tablice haszujące oferują efektywne wyszukiwanie kluczy w systemie.

Początkowe badania nad DHT były motywowane istniejącymi systemami peer-to-peer, oferującymi głównie możliwośćdzielenia się plikami, np. *Napster*, *Gnutella* i *Freenet*. Sieci te na różne sposóby realizowały wyszukiwanie zasobów, ale każdy z nich miał swoje wady. *Napster* korzystał z centralnego katalogu, do którego każdy serwer wysyłał swoją listę plików i który zarządzał wyszukiwaniem plików. W *gnutelli* w celu odnalezienia zasobu, zapytanie wysyłane było do każdego węzła w sieci w określonym promienu, co skutkowało mniejszą szybkością i mnóstwem przesyłanych wiadomości oraz podobnie jak w przypadku sieci *Freenet*, w której znajdowanie klucza odbywało się za pomocą heurystycznej metody, brak gwarancji odnalezienia klucza.

Idea rozproszonych tablic haszujących ma na celu rozwiązanie wszystkich przedstawionych problemów:

- w pełni rozproszony, skalowalny system; brak centralnego zarządzania,
- odporność na awarie węzłów,
- deterministyczny algorytm routingu, z dowiedzioną poprawnością, pozwalający na efektywne wyszukiwanie.

Jedną z pierwszy sieci DHT, bazującej na schemacie *Consistent Hashing* jest *Chord*, opisany w rozdziale 2.1.

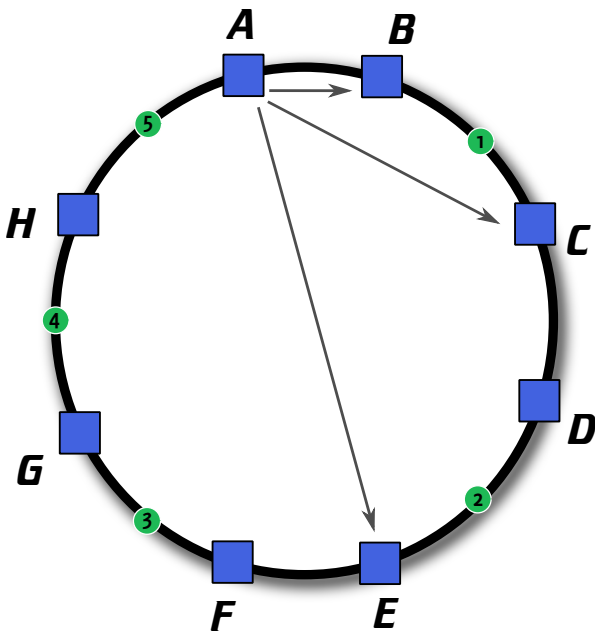
2.1 Chord

Sieć *Chord* opracowana przez Stoicę, Morrisa, Kargera, Kaashoeka i Balakrishnana [11] to jedna z czterech pierwszych powstałych rozproszonych tablic haszujących[2]. Zapewnia szybki i odporny na błędy protokół routingu, znajdujący węzeł odpowiedzialny za dany klucz w czasie logarytmicznym w stosunku do liczby węzłów w sieci.

U podstaw *Chorda* leży mechanizm *Constistent Hashing* opisany w rozdziale 1.3.1. W przypadku *Constistent Hashing*, aby zlokalizować węzeł odpowiedzialny za dany klucz wymagana była znajomość wszystkich węzłów w sieci. Takie rozwiązanie pozwala na szybkie odnalezienie węzła, ale nie daje możliwości skalowania na dowolną liczbę węzłów. W uproszczonej wersji *Chorda*, w celu zapewnienia efektywnego skalowania, autorzy wymagają wyłącznie znajomości przez węzeł swojego następnika. Dzięki temu, niezależnie od wielkości sieci każdy z węzłów przechowuje informacje tylko o jednym sąsiedzie, ale czas odnalezienia dowolnego klucza jest liniowy w stosunku do liczby węzłów (wiadomości są przesyłane po pierścieniu aż dotrą do docelowego serwera).

Bardziej efektywna wersja routingu zakłada, że każdy z węzłów utrzymuje tabelę z kontaktami w określonych odległościach od niego samego. Kolejne kontakty są wykładniczo coraz bardziej odległe tak, jak przedstawione to zostało na rysunku 4. Takie podejście pozwala na odnalezienie węzła odpowiedzialnego za zadany klucz w czasie $O(\log n)$. Aby łatwiej zauważyć tę własność, możemy

[2]Pozostałe to CAN[9], Tapestry [12] i Pastry[10].



Rysunek 4: Kontakty (*fingers*) węzła A w sieci Chord. Dla ustalonej długości klucza m, węzeł n przechowuje kontakty: $successor(n+2^{i-1})$ dla każdego $i = 1,2,\ldots,m$; gdzie $successor(k)$ oznacza serwer odpowiedzialny za klucz k.

przedstawić kontakty w formie drzewa (rysunek 5). Każdy krok w routingu *Chorda* skraca wtedy pozostały dystans co najmniej o połowę.

Podczas dołączania do sieci węzeł n musi znać adres co najmniej jednego węzła, który należy jużdo sieci, nazwijmy go n'. Procedura dołączania polega na poproszeniu n' o znalezienie następnika dla n. W kolejnym kroku n buduje tablicę kontaktów wysyłając zapytania o $successor(n+2^{i-1})$ dla każdego kolejnego i. Po zakończeniu tego procesu n posiada pełną listę kontaktów, ale tylko jego następnik wie o jego istnieniu.

Pozostałe węzły mają możliwość uwzględnienia nowych podczas cyklicznego wykonywania procedury *stab*??????????????TODO, odpowiadającej za uaktualnienie (poprawienie) informacji o poprzedniku i następniku dla każdego węzła. Dodatkowa procedura $fix\_fingers$, która również jest wywoływana cyklicznie, odpowiada za uaktualnianie tabel kontaktów.

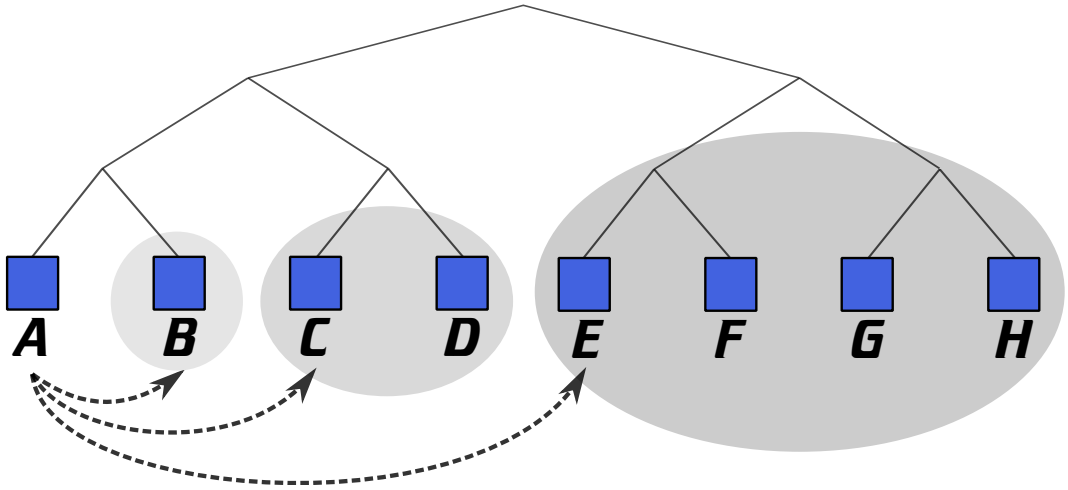
TODO: Krótkie podsumowanie Chorda?

2.2 Pastry

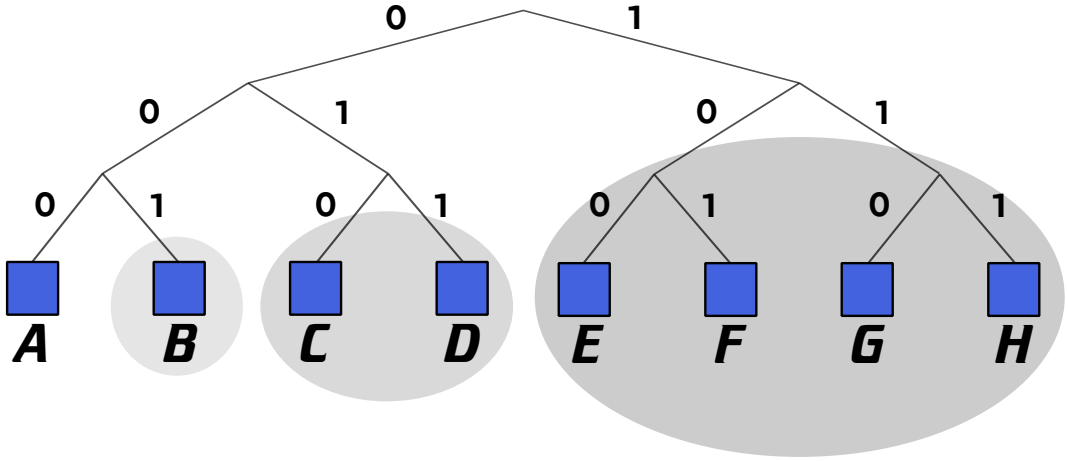
TODO: Opis pastry w celu porównania Squirrela?

2.3 Kademlia

TODO: Opis Kademli[7]: Routing, Asynchroniczność wywołań routingu - aspekty praktyczne.



Rysunek 5: Kontakty węzła *A* w sieci Chord w postaci drzewa. Węzeł ma możliwość kontaktowania się z pojedynczymi węzłami w kolejnych poddrzewach, o co raz większej wysokości. TODO: Pełniejszy opis i powiązanie z Kademlią?



Rysunek 6: Kontakty w Kademli. Szare elipsy oznaczają kolejne *k-pojemniki*. TODO: Pełniejszy opis

3 Rozproszony cache z użyciem DHT

3.1 Poprzednie prace

TODO: Squirrel p2p web cache [3]

3.2

TODO: Opis połączenia keszowania z DHT - zastosowanie w celu realizacji pierwotnych idei. TODO: Zmiany w sieci w stosunku do „tamtych czasów” :) TODO: Opis efektu slash-dot (przykłady stron w stylu reddit, wykop, etc)

4 Analiza różnych metod cachowania

TODO: Cache wielopoziomowy:

1. w pamięci,
2. na dysku,
3. w sieci P2P (te dane również w pamięci, na dysku)

Różne strategie przechodzenia między poziomami.

5 Testy

5.1 Spreparowane dane

5.2 Wdrożenie

TODO: testy opóźnień?

6 Implementacja

6.1 Wybór technologii

Tutaj próby zrobienia tego w JS jako plugin do przeglądarki.

6.2 Biblioteki

Twisted, Entangled.

6.3 Instalacja i uruchomienie

Literatura

[1] C Mic Bowman, Peter B Danzig, Darren R Hardy, Udi Manber, Michael F Schwartz, and D Wessels. The harvest information discovery and access system. *Internet Research Task Force-Resource Discovery, http://harvest. transarc. com*, 1994.

[2] Anawat Chankhunthod, Peter B Danzig, Chuck Neerdaels, Michael F Schwartz, and Kurt J Worrell. A hierarchical internet object cache. Technical report, DTIC Document, 1995.

[3] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222. ACM, 2002.

[4] 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 *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[5] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[6] Radhika Malpani, Jacob Lorch, and David Berger. Making world wide web caching servers cooperate. In *Proceedings of the Fourth International World Wide Web Conference*, pages 107–117, 1995.

[7] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[8] Dean Povey, John Harrison, et al. A distributed internet cache. *Australian Computer Science Communications*, 19:175–184, 1997.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[10] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[11] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

[12] Ben Yanbin Zhao, John Kubiatowicz, Anthony D Joseph, et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. 2001.