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

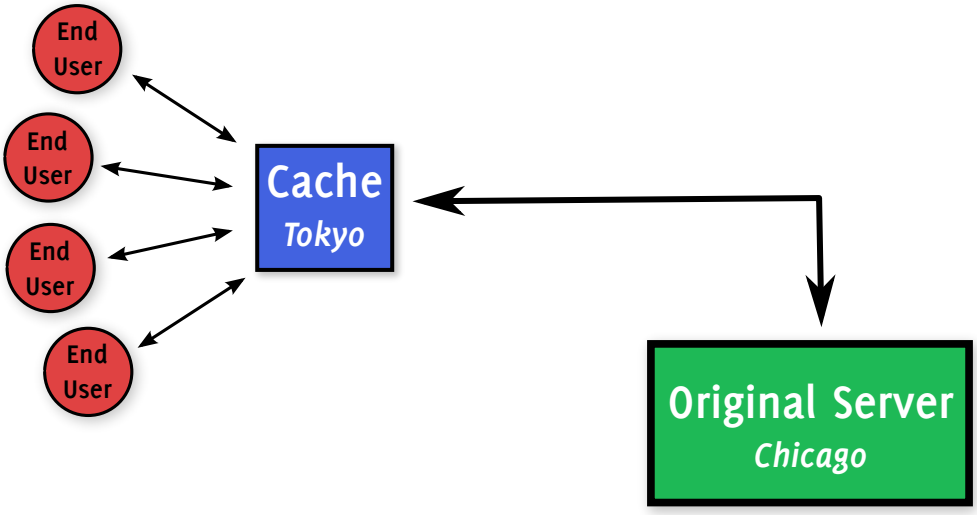
TODO: Opis problemu cachowania (oryginalnego - hot spots), nasilenie związane z efektem slash-dot, multimedia

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 **??** 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*

??) 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 **??** 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 [**?**] 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 [**?**] A. Chankhunthod i inni [**?**] 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 [**?**].

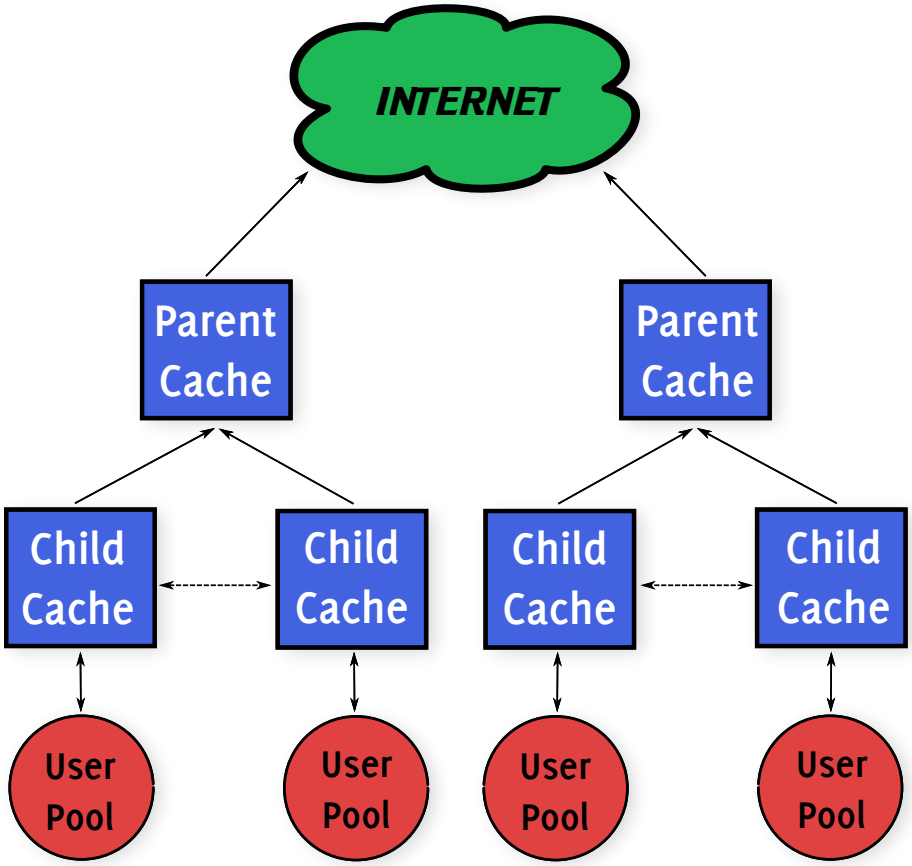
Kolejnym zaproponowanym udoskonaleniem jest cache rozproszony. Rozwiązanie zachowuje hierarchiczną strukturę w formie drzewa, ale tym razem jedynie serwery w liściach zajmują się cache'owaniem zasobów, a pozostałe serwery stanowią indeks, w którym zapamiętywana jest informacja czy zasób znajduje się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

Jak pokazały eksperymenty [**?**] 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 **??** i **??** 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:

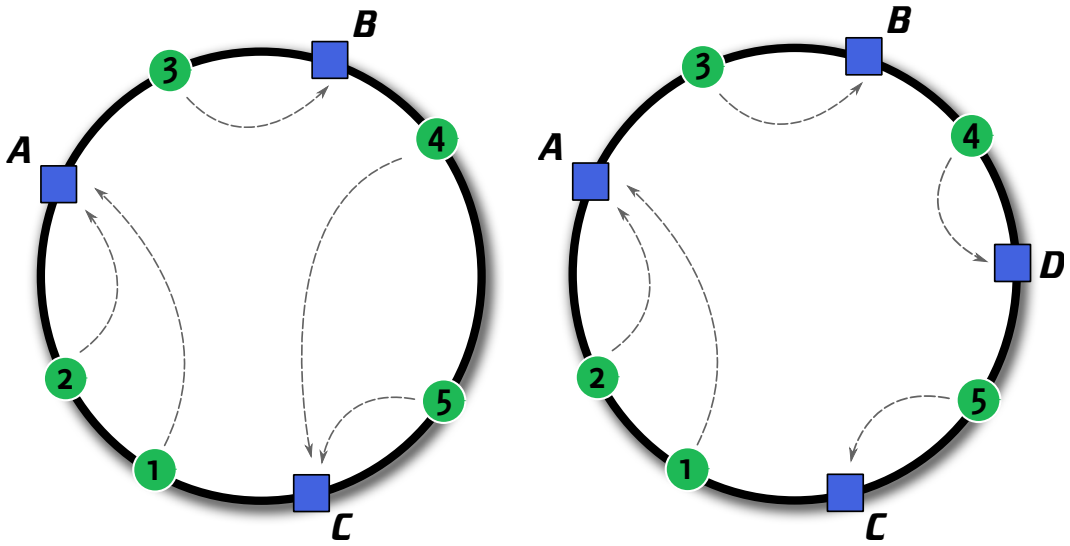
$$S \equiv hash(R) \bmod n,$$

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 [**?**].

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 **??** 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 [**?**].

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 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 możliwość dzielenia się plikami, np. *Napster*, *Gnutella* i *Freenet*. Sieci te na różne sposoby 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 promieniu, 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 **??**.

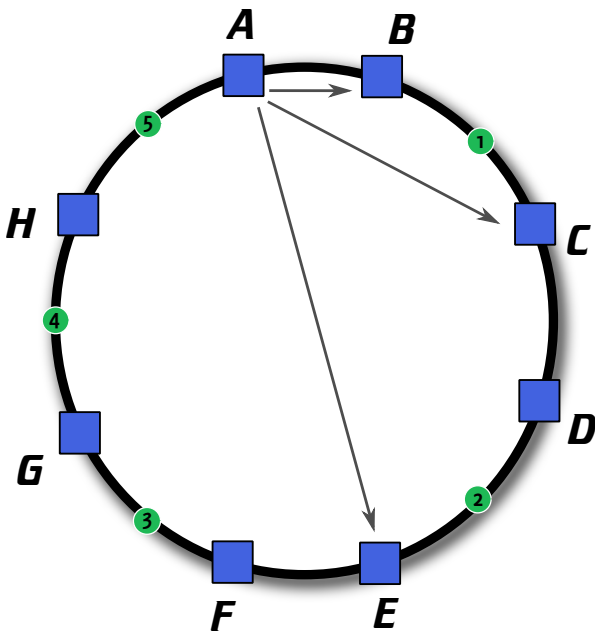
2.1 Chord

Sieć *Chord* opracowana przez Stoicę, Morrisa, Kargera, Kaashoeka i Balakrishnana [**?**] 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 **??**. 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 **??**. 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[**?**], Tapestry [**?**] i Pastry[**?**].



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 **??**). 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 już do 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.

TODO: kolejny akapit jest gówniany, bo nagle z dupy zaczyna rzucać jakimiś procedurami.

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.

2.2 Podsumowanie

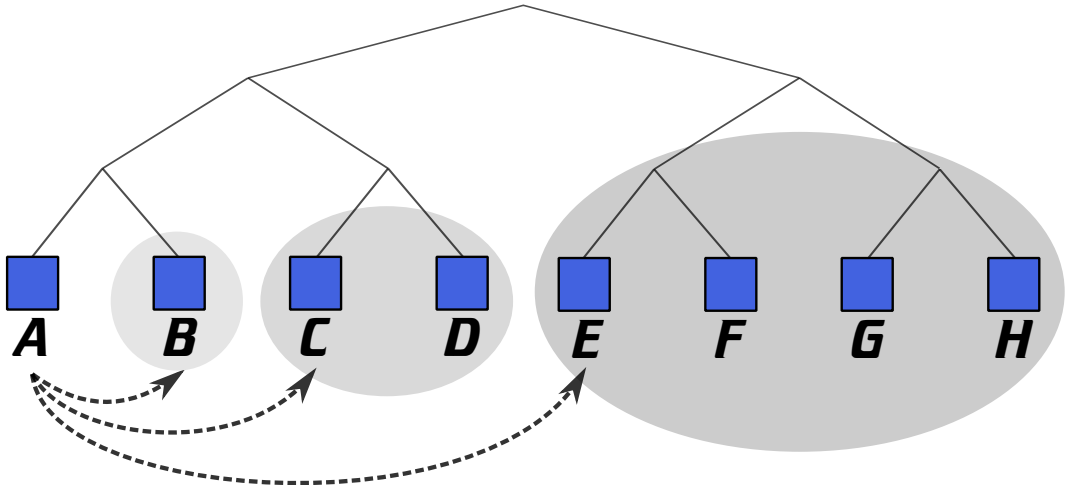
TODO: Krótkie podsumowanie Chorda?

2.3 Kademlia

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

W 2002 roku Maymounkov i Mazieres opracowali nową sieć P2P, która wykorzystywała zalety pierwszych sieci oraz uwzględniała nie tylko aspekty teoretyczne, ale również wykorzystanie sieci w praktyce.

Kademlia [**?**], podobnie jak pierwsze powstałe sieci peer-to-peer, pozwala na odnalezienie zasobu w czasie logarytmicznym od liczby węzłów w sieci. Zaletą Kademli jest fakt, że każdy pakiet wymienia-



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?

ny pomiędzy węzłami niesie ze sobą dodatkowe, użyteczne informacje o pozostałych węzłach, dzięki czemu nie jest konieczne wysyłanie dodatkowych komunikatów, niezbędnych do działania systemu. Wykorzystanie tej własności umożliwia użycie metryki odległości opartej na funkcji XOR, co więcej sieć pozwala na równoległe odpytywanie wielu węzłów w celu zmniejszenia opóźnień użytkownika. Szczegóły działania routingu Kademli zostały opisane w rozdziale **??**.

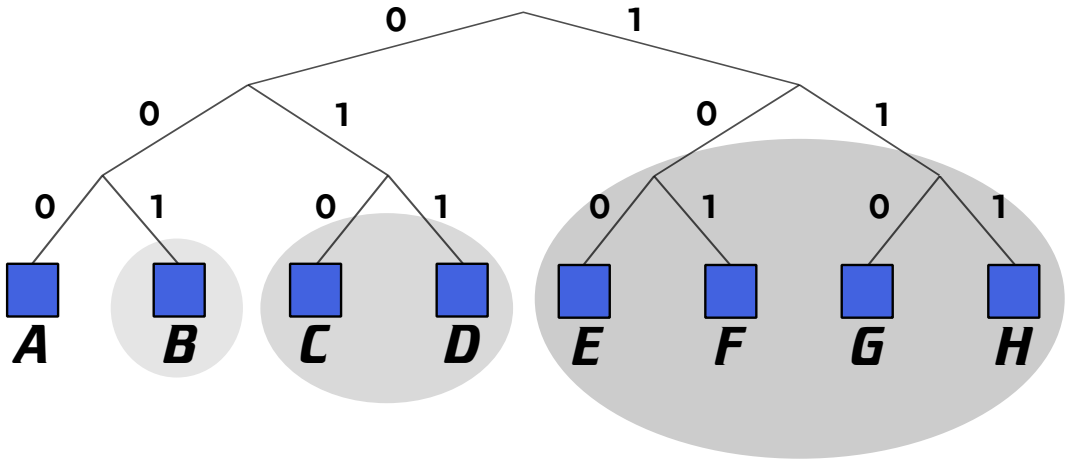
W celu identyfikacji zasobów i węzłów w sieci Kademlia używane są 160-bitowe klucze (np. hash SHA-1 identyfikatora). Zasoby są składowane w węzłach, których klucz jest blisko identyfikatora zasobu. W celu określenia "bliskości" elementów wykorzystywana jest metryka XOR, tzn. im więcej początkowych bitów dwóch kluczy jest zgodnych tym bliżej siebie się znajdują. Dzięki temu, że funkcja XOR jest symetryczna każde zapytanie, które dociera do danego węzła, może nieść ze sobą informacje o nowym kontakcie, którym jest nadawca komunikatu.

Każdy węzeł utrzymuje swoją tablicę kontaktów wykorzystywaną podczas wyszukiwania zasobów. Tablica dla węzła o identyfikatorze X składa się z k-pojemników (ang. *k-buckets*), które dla każdego $i \in [0, 160)$ przechowują adresy (maksymalnie) k węzłów, które znajdują się w odległości od 2^i do 2^{i+1} od danego węzła. Zatem klucze w i-tym pojemniku mają z X wspólny prefiks długości $160-(i+1)$, ale różnią się na bicie $160-i$. Parametr k jest stały i ustalony dla danej sieci (autorzy Kademli proponują wartość $k = 20$). Na rysunku **??** przedstawiona została przykładowa sieć dla kluczy długości 3 oraz kontakty węzła o kluczu 000.

Bazując na zapisów logów zebranych przez Saroiu i innych [**?**] z działającej sieci Gnutella, zaobserwowano, że prawdopodobieństwo, że węzeł który znajduje się w sieci od x minut będzie się w niej znajdował również w czasie $x+60$ rośnie wraz ze wzrostem x. Własność ta została wykorzystana w Kademli w celu ustalenia kolejności w k-pojemnikach. Węzły są uszeregowane według malejącego czasu ostatniej widoczności[3]. W momencie kiedy do danej listy ma trafić kolejny węzeł do ostatniego węzła na liście wysyłany jest komunikat w celu sprawdzenia czy jest wciąż aktywny. W przypadku braku odpowiedzi węzeł jest usuwany z listy, a nowy węzeł wstawiany jest na początek. W przeciwnym razie węzeł jest przesuwany na początek listy, a nowy kontakt jest ignorowany. Dzięki takiemu podejściu sieć dodatkowo staje się częściowo odporna na ataki, w których adwersarz wprowadza wiele

[3]Czas ten jest uaktualniany w momencie otrzymywania zapytań lub odpowiedzi na zapytania od danego węzła.

nowych węzłów, które mają na celu "wypchnięcie" prawdziwych węzłów z tabel kontaktów.



Rysunek 6: Tablica kontaktów węzła 000 w przykładowej sieci Kademli. Szare elipsy oznaczają kolejne *k*-pojemniki. Rysunek ten bardzo przypomina rysunek **??**. W przypadku Chorda węzeł znał co najwyżej jeden węzeł z każdego poddrzewa. Dzięki zastosowaniu metryki XOR w Kademli węzły bez dodatkowych komunikatów mogą dowiedzieć się o pozostałych węzłach, które należą do danego przedziału. Informacja ta wykorzystywana jest w celu przyspieszenia routingu w praktycznych zastosowaniach.

2.3.1 Routing

Protokół Kademli oparty jest o cztery procedury: `STORE`, `PING`, `FIND_NODE` oraz `FIND_VALUE`. Pierwsza procedura wysyła prośbę o zapamiętanie pary (klucz, wartość). `PING` używany jest do ustalenia czy dany węzeł powinien zostać usunięty z pewnego *k*-pojemnika lub czy dalej jest aktywny. Dwie kolejne procedury stanowią najważniejszą część sieci. Obie używają tego samego algorytmu routingu w celu odnalezienia węzła lub wartości, która reprezentowana jest przez zadany klucz.

W celu odnalezienia węzła o kluczu X przez węzeł S na podstawie tablicy kontaktów określanych jest k najbliższych do klucza X węzłów sieci. W kolejnym kroku S przesyła zapytanie o k najbliższych węzłów do węzłów wybranych w poprzednim kroku. Dzięki zastosowanej metryce XOR i jej własnościom możliwe jest wysyłanie równoczesnych zapytań do kilku węzłów. Liczbę równoległych żądań określa parametr α, dla którego Maymounkov i Mazieres proponują wartość 3. Dzięki odpytywaniu wielu węzłów w tym samym czasie możemy uzyskać krótszy czas dostępu do zasobów, w przypadku gdy część węzłów nie odpowiada na komunikaty węzła S. Procedura `FIND_VALUE` obsługiwana jest dokładnie w ten sam sposób, dopóki jeden z węzłów nie posiada wartości odpowiadającej danemu kluczowi. Wtedy węzeł zamiast zwracać k najbliższych węzłów zwraca po prostu wartość, znajdującą się pod podanym kluczem.

2.4 Podsumowanie

TODO: Krótkie podsumowanie kademli; Może nie robić z tego osobnego akapitu?

Rysunek 7: TODO: jakiś wykres związany z slashdot

3 Rozproszony cache z użyciem DHT

Obecne zastosowania DHT skupiają się głównie na systemach wymiany plików takich jak Kad czy BitTorrent. W systemach tych DHT są używane w celu odnalezienia komputerów, które przechowują dany plik. W tej pracy proponujemy użycie DHT w celu cache'owania zasobów internetowych. Jak opisane zostało w rozdziale **??** cache'owanie zasobów leżało u podstaw powstania mechanizmu *Consistent Hashing*, który z kolei zapoczątkował rozwój DHT.

3.1 Motywacja

Pod koniec lat dziewięćdziesiątych, kiedy opracowane zostało *Constistent Hashing* z internetu korzystało 7% światowej populacji (31% w krajach rozwiniętych). Problemy, które stały wówczas przed twórcami mechanizmu rozpatrywane były głównie z punktu widzenia serwerów, których zalanie nadmierną liczbą zapytań powodowało czasowe problemy z dostępem do zasobów. Rozwiązania miały na celu stworzenie infrastruktury, która pozwalałaby na odciążenie serwerów i w efekcie zwiększenie dostępności danych.

Obecnie, około 13 lat później, Internet i jego użycie zmieniło się diametralnie. Liczba użytkowników Internetu podwoiła się w krajach rozwiniętych (do poziomu 77%) oraz wzrosła ponad pięciokrotnie (do 39%) biorąc pod uwagę całą populację. Wraz z rozwojem sieci rozwijała się też technologia. Obecne serwery mogą sprostać o wiele większemu obciążeniu. Pojawiły się również dodatkowe rozwiązania pozwalające na łatwiejsze skalowanie dużych stron internetowych takie jak CDN (*Content Delivery Network*) oraz *Cloud-based hosting*.

Jakkolwiek oryginalna motywacja cache'owania straciła obecnie na znaczeniu, to jednak współczesny Internet stawia szereg nowych problemów, którym musimy sprostać. Wraz ze wzrostem popularności serwisów, które zajmują się agregowaniem treści (jak *reddit*[4] oraz *Wykop*[5]) coraz większym problemem staje się tzw. *slash-dot effect*. Efekt obserwowany jest kiedy strona ciesząca się pierwotnie niewielkim lub umiarkowanym zainteresowaniem trafia do agregatora, dzięki któremu w krótkim czasie zyskuje olbrzymi wzrost liczby wizyt. W rezultacie serwer, który nigdy nie był przygotowany do obsługi tak wielu zapytań zostaje zalany i staje się niedostępny. Na wykresie **??** przedstawiona została liczba wizyt spowodowana efektem slash-dot.

TODO: Akapit o średnim rozmiarze strony oraz prędkościach internetu. Jakis wykres mówiący o średnim rozmiarze strony (może zestawione ze średnią prędkością łącza?)

Rozproszony cache oparty o DHT wykorzystuje moc obliczeniową współczesnych maszyn klienckich w celu odciążenia serwerów i zmniejszenia opóźnień użytkowników.

3.2 Poprzednie prace

TODO: Squirrel p2p web cache [**?**]

4 Analiza różnych metod cachowania

TODO: Cache wielopoziomowy:

[4] `http://reddit.com`
[5] `http://wykop.pl`

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

W niniejszym rozdziale opisana została implementacja rozproszonego cache'u z użyciem DHT. W rozdziale **??** przedstawione zostały podjęte próby innej formy implementacji, a w rozdziale **??** opisane zostało ostatecznie wybrane rozwiązanie. Rozdział **??** prezentuje informacje o instalacji i uruchomieniu stworzonego programu.

6.1 Wybór technologii

Wybór technologii był silnie zależny od celu, który chcieliśmy osiągnąć. Początkowo istotnym aspektem, był aspekt praktyczny tworzonego rozwiązania, priorytetem było maksymalne uproszczenie i ułatwienie procesu instalacji cache'u, tak aby jak najwięcej osób było chętnych przetestować system. Celem było zebranie danych do testów oraz analiza działania i uzyskiwanych opóźnień podczas normalnej pracy programu. Z tego powodu przed implementacją ostatecznego rozwiązania opisanego w rozdziale **??** podjęte zostały próby utworzenia pluginu do przeglądarki, opisane w rozdziałach **??** i **??**.

6.1.1 Plugin do przeglądarki

Pierwszą podjętą próbą implementacji było stworzenie rozszerzenia do przeglądarki Chrome[6], które korzystałoby z nowego API dla połączeń Peer-to-peer, zaproponowanego w HTML5[7].

Jako *proof-of-concept* stworzony został plugin, którego kod znajduje się na listing **??**.

```
(function() { 'use strict';
var cowImg;
chrome.webRequest.onBeforeRequest.addListener(function(details){
  // Don't redirect multiple times
   if (loldogs.indexOf(details.url) !== -1) {
      return;
   }

  // Redirect to cow
```

[6] http://chrome.google.com

[7] http://www.w3.org/TR/2008/WD-html5-20080122/#peer-to-peer, http://www.w3.org/TR/webrtc

```
    var url = cowImg;
  // or to loldogs
    if (Math.random() < .5) {
      url = loldogs[Math.round(Math.random() * loldogs.length)];
    }
    console.log("Redirecting", details.url, " to ",
     (url === cowImg ? "COW" : url));

  // Return redirection url
    return {
      redirectUrl: url
    };
}, {
    urls: ['<all\_urls>'],
    types: ['image']
}, ["blocking"]);

cowImg = "...";
}());
```

Rozwiązanie korzysta z API `chrome.webRequest.onBeforeRequest`[8], które pozwala nasłuchiwać oraz modyfikować wszystkie żądania, wychodzące z przeglądarki. Nasłuchiwane są tylko żądania o obrazki, z dowolnymi adresami URL. Następnie każde żądanie z prawdopodobieństwem $1/2$ zostaje zastąpione obrazkiem `cowImg`, który znajduje się w pliku źródłowym w zakodowanej postaci (`base64`) oraz z prawdopodobieństwem $1/2$ jednym z obrazków z tablicy `loldogs`. Dzięki przekierowaniu na zasób w postaci `base64` nie jest konieczne wykonywanie dodatkowego żadania przez przeglądarkę i obrazek jest wyświetlany od razu.

Niestety stworzenie tego pluginu pokazało poważną wadę tego podejścia. Żeby przechwytywać żądania w trybie `onBeforeRequest`, czyli przed wykonaniem zapytania przez przeglądarkę musimy korzystać z opcji `blocking`, która powoduje, że żądania muszą być przetwarzane synchronicznie. Zatem próby pobrania zasobu z sieci P2P podczas przechwytywania zapytania będą skutkowały zwiększonymi opóźnieniami w dostępie do zasobów, co podważa podstawową ideę rozwiązania.

6.1.2 Plugin *Native Client*

Kolejnym podejściem było stworzenie pluginu dla przeglądarki Chrome w technologii *Native Client(NaCL)*. *NaCL* pozwala na kompilację programów, które uruchamiane są w piaskownicy (ang. *sanboxed environment*) w celu zapewnienia bezpieczeństwa użytkownika. Główną zaletą technologii jest wydajność, która jest bliska wydajności natywnych aplikacji.

Pomimo ogłoszenia przez Google gotowości *Native Client* do produkcyjnego użycia, plugin nie został stworzony z powodu słabej dokumentacji oraz braku wersji odpowiednich bibliotek na platformie *NaCL*.

TODO: Ostateczne rozwiązanie?

6.2 Proxy cache'ujące

Po nieudanych próbach stworzenia rozszerzenia do przeglądarki stworzone zostało rozwiązanie, które wymaga dodatkowej instalacji i konfiguracji systemu, ale bazuje na znanych i dobrze przetestowanych komponentach. Ostateczną aplikacją jest proxy cache'ujące, w którym każde zapytanie (wykluczając

[8] `http://developer.chrome.com/extensions/webRequest.html`

Rysunek 8: TODO: Rysunek przedstawiający dwie części aplikacji i request, który jest obsługiwany z cacha lub nie.

ruch `https`) jest przekazywane do oryginalnego serwera tylko w momencie, gdy nie zostanie znalezione w cache'u.

Program został wykonany w języku `Python` z użyciem bibliotek `Twisted`[9] oraz `Entangled`[10]. Aplikacja składa się z dwóch części:

Proxy - server proxy, który odpowiada za przekazywanie zapytań oraz odpytywanie cache'u o zasoby,

P2P - klient sieci Peer-to-Peer, pozwalający na odszukiwanie zasobów, które w tej sieci się znajdują.

Na rysunku **??** przedstawione zostało przykładowe zapytanie i jego droga przez poszczególne komponenty aplikacji.

TODO: Cos o Twisted i jego zaletach, jakiś fragment o tym, że entangled też jest w twisted? TODO: Możliwość łatwego podmieniania komponentów (np. jak budowany jest cache)

6.3 Symulator

W celu wykonania testów, których wyniki przedstawione zostały w rozdziale **??** wykonany został symulator bazujący na stworzonej aplikacji.

Symulator pozwala na wczytanie danych o klientach i żądaniach, które mają wykonać. Każdy z klientów jest osobnym procesem, który uruchamiany jest przez proces główny (`simulator_main.py`). Wyniki symulacji zapisywane są do pliku z logami i zawierają identyfikator cache'u, z którego otrzymano zasób (lub -1 w przypadku, gdy zasób nie został znaleziony w żadnych cache'u) oraz opóźnienie, czyli czas, który upłynął od wysłania zapytania do otrzymania odpowiedzi.

6.4 Skrypty pomocnicze

Aby ułatwić przetwarzania plików z logami zostały wykonane pomocnicze skrypty dla środowiska `R`[11].

Pierwszy ze skryptów (`read.data.r`) odpowiedzialny jest za wczytanie danych testowych i przetworzenie ich na format, które oczekuje symulator. Skrypt wczytuje więc plik z logami z serwera, który zawiera wpisy dla wszystkich żądań i dzieli go na pliki z zapytaniami dla poszczególnych klientów. Dodatkowo czas zapytań zamieniany jest z absolutnego na relatywny, czyli pierwsze zapytanie ma czas 0, a czas pozostałych to opóźnienie w stosunku do pierwszego żądania.

Skrypt `read.logs.r` pozwala na wczytanie danych wyjściowych z symulatora i opracowanie na ich podstawie wykresów odpowiedzi z poszczególnych cache'y. Skrypt zapisuje też uproszczone wyniki symulacji, które używane sa przez ostatni ze skryptów (`combine.logs.r`). Ponieważ wyniki symulacji są w pewnym stopniu losowe (co wynika z użycia protokołu UDP dla sieci Kademlia oraz różnego obciążenia maszyny, na której uruchomione były symulacje) ostatni skrypt zbiera wyniki z wielu uruchomień symulatora i prezentuje wykresy z uśrednionymi wynikami.

6.5 Instalacja i uruchomienie

[9] `http://twistedmatrix.com`
[10] `http://entangled.sourceforge.net`
[11] `http://www.r-project.org`