WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI
POLITECHNIKI WROCŁAWSKIEJ

CACHE'OWANIE STRON WWW BAZUJĄCE NA SIECIACH PEER-TO-PEER

TOMASZ DRWIĘGA

Praca magisterska napisana
pod kierunkiem
dra Mirosława Korzeniowskiego

WROCŁAW 2013

Spis treści

1. Historia cache'owania **3**
- 1.1. Serwery cache'ujące . . . 3
- 1.2. Cacherozproszony i hierarchiczny . . . 4
- 1.3. Dynamiczna restrukturyzacja . . . 5
- 1.4. *Consistent Hashing* . . . 6

2. Rozproszone tablice haszujące (DHT) **6**
- 2.1. Chord . . . 8
- 2.2. Kademlia . . . 9

3. Rozproszony cache z użyciem DHT **11**
- 3.1. Motywacja . . . 11
- 3.2. Poprzednie prace . . . 12
- 3.3. Architektura . . . 13

4. Testy **15**
- 4.1. Dane testowe . . . 15
- 4.2. Cache'owanie w pamięci . . . 16
- 4.3. Cache'owanie w sieci P2P . . . 16
- 4.4. Cache'owanie dwupoziomowe . . . 20
- 4.5. Wnioski . . . 21

5. Implementacja **22**
- 5.1. Wybór technologii . . . 22
  - 5.1.1. Plugin do przeglądarki . . . 22
  - 5.1.2. Plugin *Native Client* . . . 23
- 5.2. Proxy cache'ujące . . . 23
- 5.3. Symulator . . . 23
- 5.4. Skrypty pomocnicze . . . 24
- 5.5. Instalacja i uruchomienie . . . 24

6. Podsumowanie **24**

A. Załączniki **25**
- A.1. Kod pluginu do *Chrome* . . . 25
- A.2. Opcje konfiguracyjne aplikacji . . . 25

Wstęp

Problem cache'owania pomimo wielu prac na jego temat, wciąż pozostaje tematem niewyczerpanym. Wraz ze zwiększającym się udziałem multimediów wysokiej jakości oraz coraz większych rozmiarów stron internetowych, cache'owanie zasobów na różnych etapach realizacji zapytania staje się niezwykle istotne.

Intensywne badania na temat cache'owania doprowadziły do powstania wielu ciekawych rozwiązań, takich jak cache hierarchiczny czy mechanizm *Consistent Hashing*. Ten ostatni z kolei zapoczątkował zupełnie odrębną gałąź badań: sieci peer-to-peer. W niniejszej pracy proponujemy wykorzystanie sieci peer-to-peer w celu cache'owania zasobów internetowych. System taki pozwala wykorzystać potencjalnie nieograniczone zasoby sieci w celu składowania obiektów, które zostały pobrane z Internetu. W pracy porównujemy wpływ rozmiaru cache'u oraz algorytmów zastępowania elementów w pełnym cache'u na liczbę trafień (ang. *Cache hits*). Rozważane są również różne połączenia cache'owania zasobów lokalnie oraz w sieci peer-to-peer. W celu realizacji sieci peer-to-peer używany jest protokół Kademlia, który dzięki równoległym zapytaniom w trakcie routingu oraz prostej implementacji świetnie nadaje się do zastosowań praktycznych.

TODO: Coś bardziej rozwinąć wstęp?

W pierwszym rozdziale opisany został problem cache'owania oraz krótka historia rozwoju cache'owania , doprowadzająca do powstania *Constistent Hashing*. Rozdział 2 opowiada o rozproszonych tablicach haszujących (ang. *DHT*). W rozdziale 3 opisujemy wykorzystanie DHT w celu cache'owania zasobów internetowych. Rozdział 4 opisuje wykonane testy oraz przedstawia ich wyniki, a w rozdziale 5 opisane zostały próby implementacji rozwiązania w formie pluginu do przeglądarki oraz wersja wykonana ostatecznie, czyli proxy cache'ujące.

1. Historia cache'owania

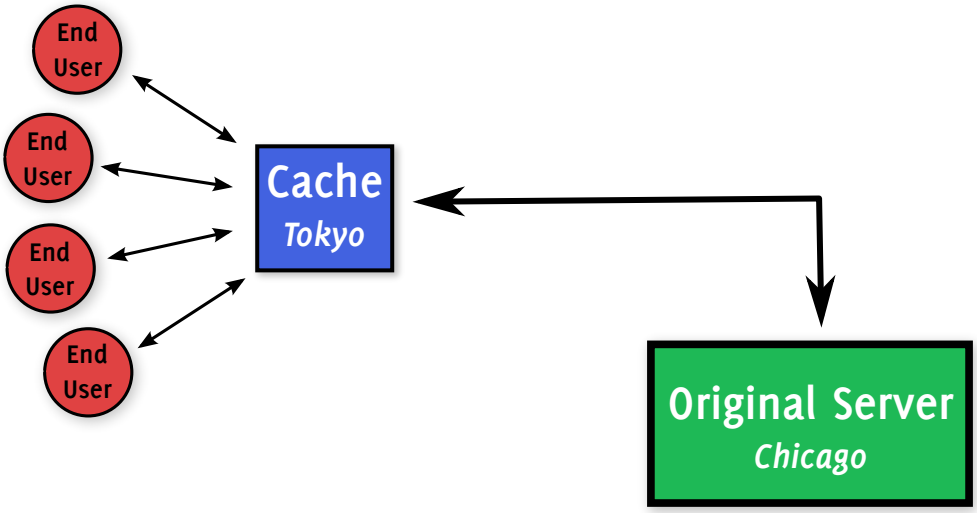
Wraz z rosnącą liczbą użytkowników Internetu w latach dziewięćdziesiątych zaczęły pojawiać się problemy z dostępem do pewnych zasobów. Duża liczba odwołań do popularnych stron w krótkim czasie powodowała znaczące obciążenie serwerów, które dany zasób posiadały. Czasami, z powodu nadmiernego zalania[1] żądaniami, serwer nie był w stanie obsłużyć ich wszystkich, przez co strona stawała się niedostępna. Serwery, posiadające takie zasoby noszą nazwę „gorących punktów" (ang. *hot spots*).

1.1. Serwery cache'ujące

Jako rozwiązanie problemu „gorących punktów" pojawiły się serwery cache'ujące. Rysunek 1 przedstawia schemat sieci, w której wprowadzony został transparentny, lokalny cache oraz 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 jego odpowiedź.

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) 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] ang. *flooded*, *swamped*



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.

1.2. Cacherozproszony i hierarchiczny

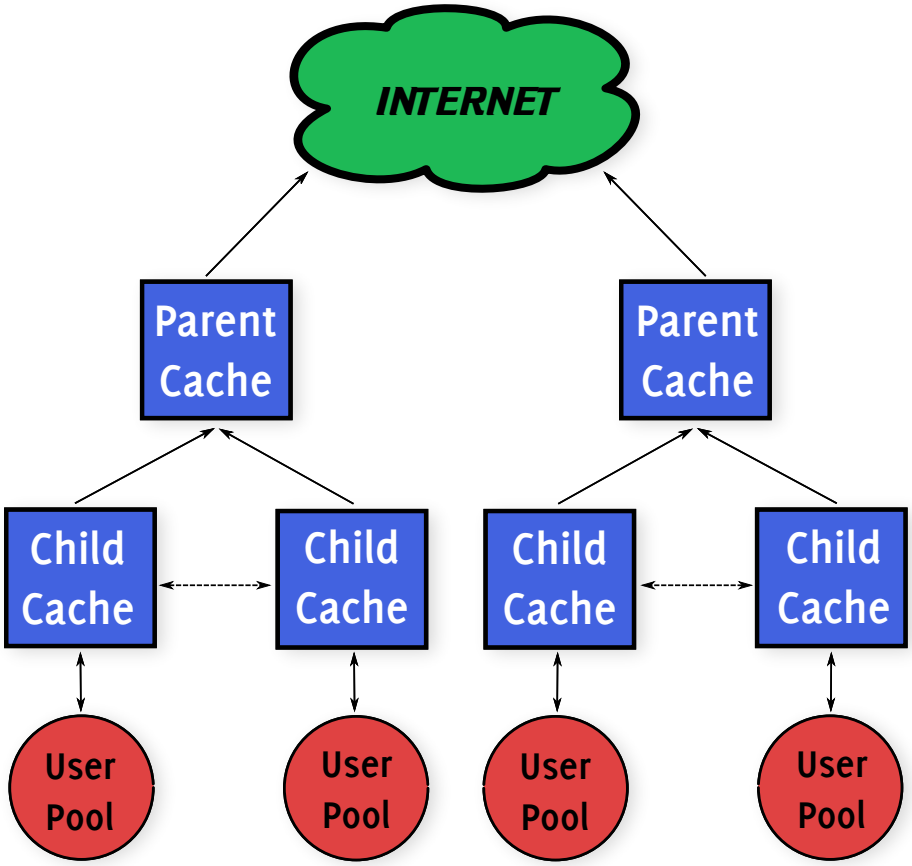
Rozwiązanie przedstawione w rozdziale 1.1 pozwala na odciążenie serwera docelowego. Zastanówmy się jednak co będzie się działo w przypadku zwiększania liczby użytkowników korzystających z serwera cache'ującego. Duża liczba żądań może spowodować dokładnie taką samą sytuację jak w przypadku serwera docelowego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

W 1995 roku Malpani i inni [11] 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'y nie 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 [3] 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 w konstruowanej strukturze drzewa. 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, co oznacza że stają się wrażliwe na zalanie oraz składują duże ilości danych [14].

Kolejnym zaproponowanym udoskonaleniem jest cache rozproszony. Rozwiązanie zachowuje hie-



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.

rarchiczną 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 przeprowadzone przez [14], 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 których 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 można 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 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 $(n+1)$-go serwera powoduje, że wszystkie zasoby zostaną przypisane w inne miejsca.

1.4. *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'u były przesuwane pomiędzy „starymi" serwerami. Taką własność gwarantuje mechanizm *Consistent Hashing*, opracowany przez Kargera i Lehmana [8].

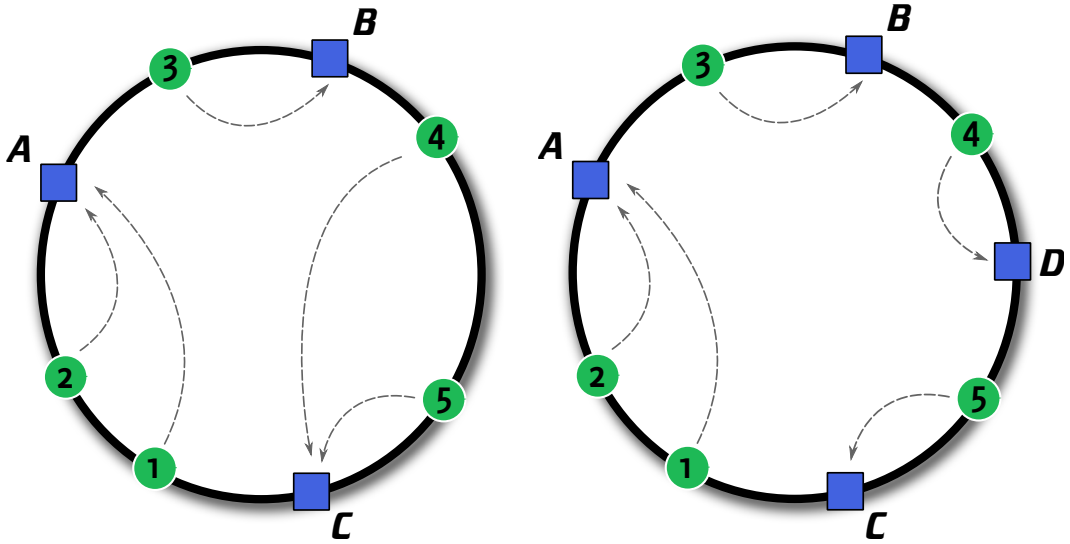
Załóżmy, że dysponujemy funkcją haszującą w odcinek $[0,1]$, za pomocą której każdemu zasobowi i serwerowi przyporządkowujemy punkt na tym odcinku. Dodatkowo, traktujemy odcinek jako okrąg o obwodzie równym 1. W podanym przez Kargera i Lehmana schemacie dany zasób jest obsługiwany przez najbliższy serwer 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.

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 z dużym prawdopodobieństwem zasoby będą równomiernie przyporządkowane do serwerów [9].

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 zbudo-



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.

wanej 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:

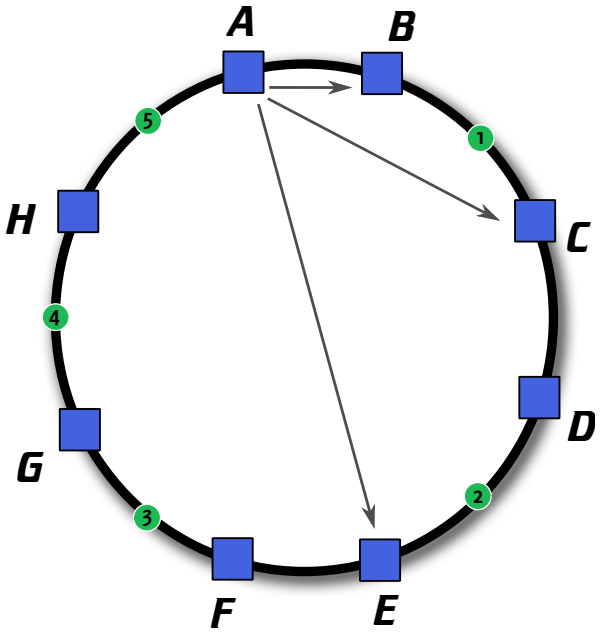
- w pełni rozproszony, skalowalny system; brak centralnego zarządzania,
- odporność na awarie węzłów,
- deterministyczny algorytm routingu, z dowiedzioną poprawnością, pozwalający na efektywne wyszukiwanie.

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

2.1. Chord

Sieć *Chord* opracowana przez Stoicę, Morrisa, Kargera, Kaashoeka i Balakrishnana [21] 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 1.4. 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).

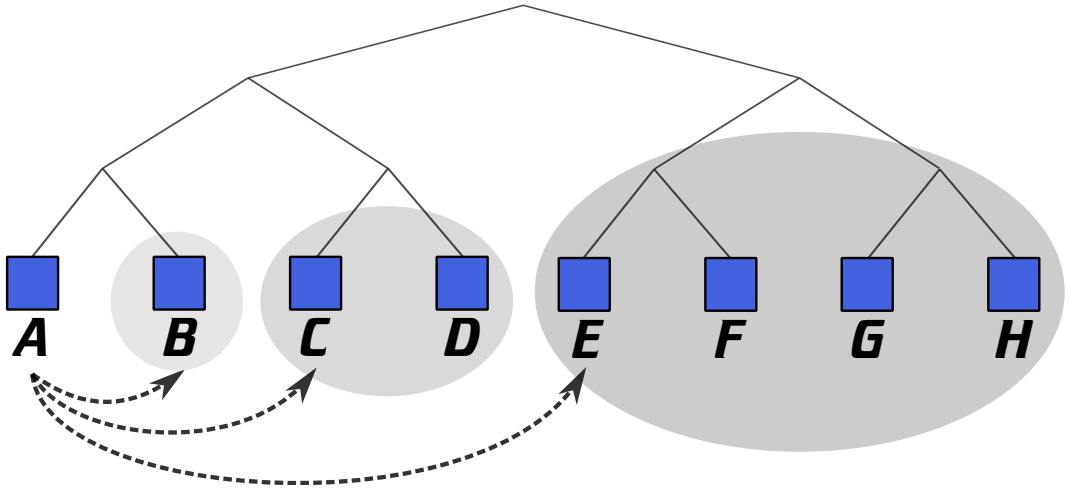


Rysunek 4: Kontakty (*fingers*) węzła A w sieci Chord. Dla ustalonej długości klucza m, węzeł n przechowuje kontakty: $successor(n+2^{i-1})$ dla każdego $i = 1,2,\ldots,m$; gdzie $successor(k)$ oznacza serwer odpowiedzialny za klucz k.

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 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 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

[2]Pozostałe to CAN[16], Tapestry [23] i Pastry[17].



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.

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 stabilizacyjnej, odpowiadającej za uaktualnienie (poprawienie) informacji o poprzedniku i następniku dla każdego węzła. Dodatkowo również cyklicznie wywoływana jest procedura, która odpowiada za uaktualnianie tabel kontaktów.

Chord stanowi implementację mechanizmu *Constistent Hashing* w sieci peer-to-peer. Dzięki dobrym właściwościom (logarytmiczny routing) oraz prostocie działania sieć ta jest bardzo często wykorzystywana do opisu bardziej zaawansowanych algorytmów, takich jak równoważenie obciążenia [15], usprawnienia routingu [12] i tym podobnym. Jednakże w praktycznych zastosowaniach, kiedy część z węzłów może nie odpowiadać (lub może odpowiadać wolno), algorytm routing w *Chordzie* podczas jednego kroku kontaktuje się tylko z jednym węzłem i musi czekać na jego odpowiedź, co może powodować znaczne opóźnienia.

2.2. Kademlia

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.

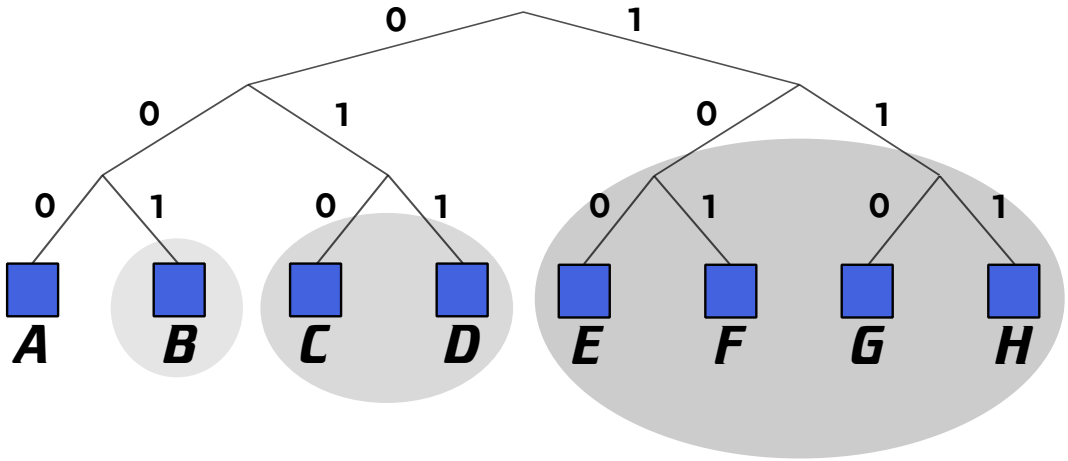
Kademlia [13], podobnie jak pierwsze powstałe sieci peer-to-peer, pozwala na odnalezienie zasobu w czasie logarytmicznym względem liczby węzłów w sieci. Zaletą *Kademli* jest fakt, że każdy pakiet wymieniany pomiędzy węzłami niesie ze sobą dodatkowe, użyteczne informacje o pozostałych węzłach, dzięki 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 2.2.

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 6 przedstawiona została przykładowa sieć dla kluczy długości 3 oraz kontakty węzła o kluczu 000.

Bazując na zapisach logów zebranych przez Saroiu i innych [19] 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 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 5. 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.

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

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

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.

Kademlia, dzięki zastosowaniu metryki XOR pozwala na znaczące uproszczenie zarówno analizy poprawności i złożoności routingu, jak i implementacji. Dodatkowo możliwość zastosowania równoległych, asynchronicznych zapytań do wielu węzłów naraz niesie ze sobą duże zalety praktyczne. Podejście to pozwala na zmniejszenie opóźnień spowodowanych nieodpowiadającymi serwerami. Ponieważ, w przeciwieństwie do *Chorda*, dwa różne zapytania o te same klucze od pewnego miejsca będą zbiegały wzdłuż tej samej ścieżki możliwe jest powielanie popularnego zasobu na węzłach leżących na tej ścieżce. Dzięki temu klienci, którzy obsługują często wyszukiwane zasoby nie zostaną zablokowani z powodu nadmiernej ilości zapytań, ponieważ część odpowiedzi będzie udzielanych już na wcześniejszym etapie routingu.

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 1.4 cache'owanie zasobów leżało u podstaw powstania mechanizmu *Consistent Hashing*, który z kolei zapoczątkował rozwój DHT.

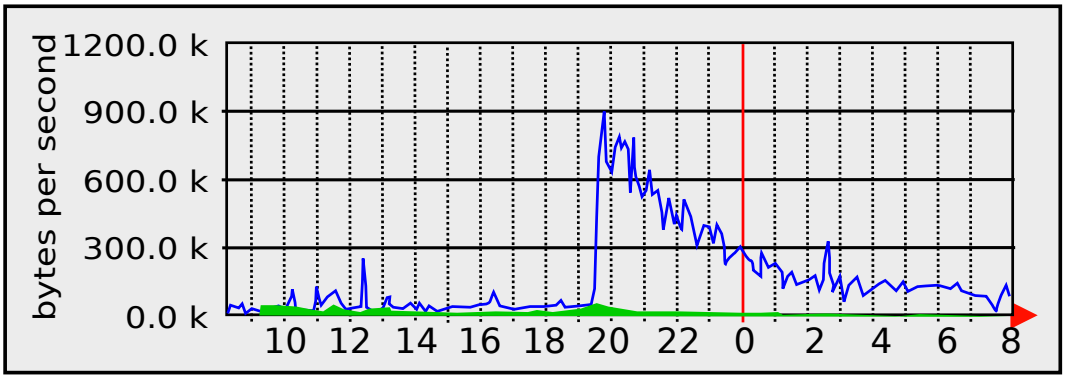
3.1. Motywacja

Pod koniec lat dziewięćdziesiątych, kiedy opracowane zostało *Constistent Hashing* z internetu korzystało 7% światowej populacji (31% w krajach rozwiniętych)[22]. 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[22]. 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 popu-

larności serwisów, które zajmują się agregowaniem treści (jak *reddit*[4] oraz *Wykop*[5]) coraz większym problemem staje się tzw. *Slashdot 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 7 przedstawiony został wzrost ilości przesyłanych danych spowodowany przez *Slashdot Effect*.



Źródło: `https://en.wikipedia.org/wiki/File:SlashdotEffectGraph.svg`

Rysunek 7: Wykres ilości danych przesyłanych przez serwer. Około godziny 20:00 można zaobserwować olbrzymie zwiększenie ruchu spowodowane efektem *Slashdot*

W dzisiejszych czasach problemem jest nie tylko chwilowa zwiększona ilość ruchu, spowodowana efektem *Slashdot*, ale przede wszystkim objętość przesyłanych danych. Porównując dane z roku 2000 oraz 2007 możemy zaobserwować ponad trzykrotny wzrost rozmiaru danych przesyłanych z internetu[18]. Na rysunku 8, przedstawiającym wykres średniego rozmiaru strony[6], ponownie widoczna jest zdecydowana tendencja wzrostowa. Na podstawie wykresu możemy wyciągnąć wniosek, że średni rozmiar strony podwaja się co około trzy lata. Dodatkowo, obecna sieć pełna jest obiektów o dużym rozmiarze (filmy, multimedia), które rzadko są cache'owane z powodu ograniczonych zasobów na scentralizowanych serwerach cache'ujących [18].

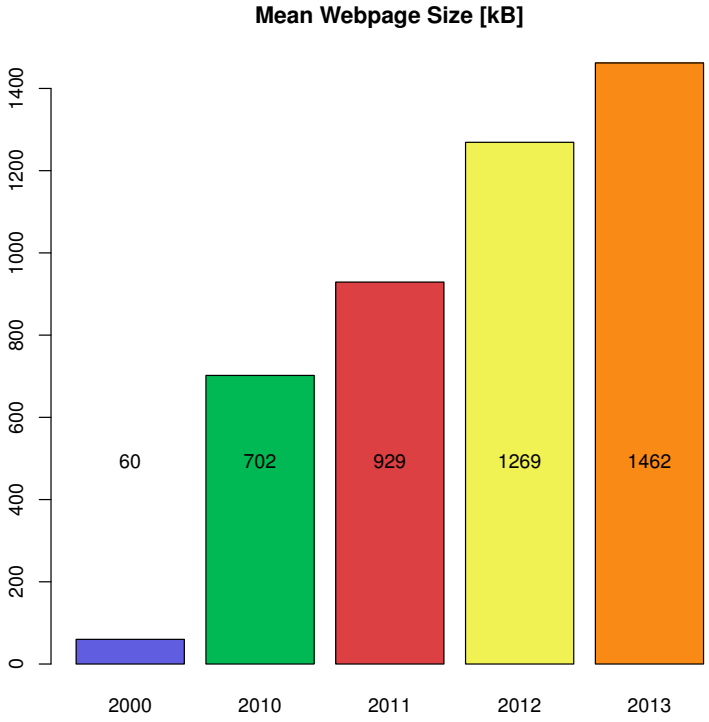
System cache'ujący oparty o sieć peer-to-peer, proponowany w tej pracy stara się adresować przedstawione problemy. W celu zwiększenia pojemności i umożliwienia cache'owania multimediów system wykorzystuje rozproszone zasoby uczestników sieci w celu odciążenia serwerów i zmniejszenia opóźnień użytkowników. Szczegóły dotyczące architektury proponowanego rozwiązania opisane zostały w rozdziale 3.3.

3.2. Poprzednie prace

Problem cache'owania zasobów opisywany był w literaturze wielokrotnie. Jako przykłady warto ponownie przytoczyć prace [11, 3, 14] omawiane już w rozdziale 1.2. Niezwykle ważne w historii cache'owania są prace Kargera i innych [8, 9] wprowadzające mechanizm *Consistent Hashing*.

Idea użycia potencjalnie nieograniczonych zasobów sieci peer-to-peer w celu cache'owania zasobów internetowych pojawia się w pracy [7, 4], przedstawiającej system *Squirrel*. *Squirrel* oparty jest o sieć Pastry[17]. Na podobnym pomyśle bazuje *Tuxedo* [20], który dodatkowo stara się cache'ować

[4] `http://reddit.com`
[5] `http://wykop.pl`
[6] Strony rozumianej jako dokument HTML, obrazki, skrypty itd.



Opracowane na podstawie: `http://httparchive.org` *oraz* `http://www.pantos.org/atw/35654.html`.

Rysunek 8: Wykres średniego rozmiaru strony w poszczególnych latach. W ciągu ostatnich 3 lat średni rozmiar strony podwoił się.

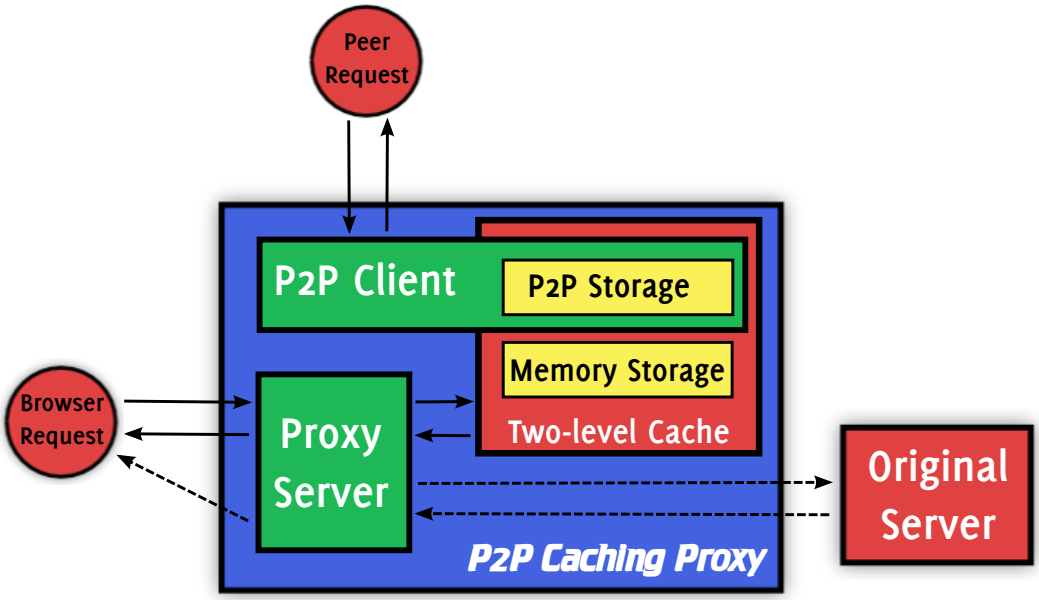
również zasoby, które są spersonalizowane, jednak wymaga scentralizowanej konfiguracji i nie jest przystosowany do użycia przez użytkowników końcowych. Kolejną siecią opartą na unikalnej implementacji DHT jest *Kache*[10], używający opracowanej na własne potrzeby sieci *Kelips*. Ciekawy pomysł zastosowania sieci peer-to-peer w celu przyspieszania strumieniowania multimediów został opisany w [5].

Pomimo podobnej tematyki przytoczonych prac żadna z nich nie wykorzystuje sieci *Kademlia*, oferującej pewne zalety istotne w zastosowaniach praktycznych, w szczególności równoległe zapytania w trakcie routingu.

3.3. Architektura

Opracowana aplikacja składa się z dwóch głównych części: serwera proxy oraz klienta sieci peer-to-peer. Oba komponenty połączone są ze sobą przez zarządcę cache'u.

Serwer proxy odpowiada za obsługę żądań przychodzących od przeglądarki. Następnie odpytywany jest zarządca cache'u. W zależności od wybranego typu cache'owania sprawdzany jest cache w pamięci lub zapytanie wysyłane jest do cache'u rozproszonego w sieci peer-to-peer. Po otrzymaniu odpowiedzi z sieci P2P jeżeli zasób został znaleziony to jego wartość zwracana jest do klienta, a w przeciwnym przypadku serwer proxy wysyła zapytanie do serwera oryginalnego. Po otrzymaniu



Rysunek 9: Przekrój architektury systemu. Składowa *Two-level cache* odpowiada za zarządzanie cache'em. W pierwszej kolejności odpytywany jest lokalny cache w pamięci, w przypadku gdy poszukiwana wartość nie została znaleziona zapytanie wysyłane jest do sieci P2P. Jeżeli zasób nie zostanie znaleziony w żadnym z dostępnych cache'y serwer proxy wyśle żądanie do serwera oryginalnego, a następnie umieści odpowiedź serwera w obu typach cache'u.

odpowiedzi zostanie ona umieszczona w cache'u.

Dzięki modularnej budowie aplikacji każdy z komponentów może zostać łatwo podmieniony. Dostępne są trzy implementacje polityk usuwania elementów z cache'u w przypadku przepełnienia: *First In First Out(FIFO)*, *Least Recently Used (LRU)* oraz *Least Frequently Used (LFU)*. Pierwszy z algorytmów, jeżeli brakuje miejsca na nowy element, usuwa z cache'u element, który dodany został do niego jako pierwszy. *LRU* z kolei usuwa ten, który był używany najdawniej. *LFU* usuwa element, który używany był najrzadziej (w przypadku takiej samej częstotliwości użycia usuwany jest element zgodnie z polityką *FIFO*). Opisane polityki mogą zostać użyte jako algorytm zarządzania cache'em w pamięci.

Aplikacja pozwala na wybranie jako implementację DHT jednej z dwóch z sieci: Chord lub Kademlia. Obie sieci w celu składowania zasobów mogą użyć dowolnego z opisanych algorytmów cache'owania.

W przeciwieństwie do większości tego typu systemów opisywanych w literaturze, w przypadku dołączania i odchodzenia węzłów do sieci nie są podejmowane żadne akcje mające na celu albo wypełnienie lokalnego cache'u nowo dołączonego klienta, albo przesłanie swoich zasobów pozostałym węzłom w przypadku odchodzenia. Pomimo, że takie podejście może zwiększać liczbę zapytań, które będą musiały zostać obsłużone z serwerów oryginalnych, to pozwala na uniknięcie efektu zwanego *churn*. *Churn* to nadmierna komunikacja wewnątrz sieci, która spowodowana jest częstym i szybkim dochodzeniem i odchodzeniem węzłów.

4. Testy

Do analizy prezentowanej aplikacji stworzony został symulator pozwalający na przetestowanie działania różnych możliwych konfiguracji. Symulator pozwala na wczytanie danych dotyczących węzłów w sieci i zapytań, które wysyłają w określonym czasie.

W celu jak najlepszego oddania realnych warunków pracy systemu w trakcie symulacji dla każdego klienta uruchamiany jest osobny proces systemowy z działającym programem. Następnie do tego procesu wysyłane są adresy i czasy, w których klient ma wysłać zapytania na dany adres. Dzięki takiemu podejściu kompleksowo testowana jest niemal cała architektura przedstawiona na rysunku 9. Jedynym komponentem, który nie bierze udziału w symulacji jest serwer proxy, ponieważ zapytania są tworzone bezpośrednio przez procesy klientów.

Do uruchomienia symulacji wymagany jest plik zawierający listę wszystkich klientów, oraz osobny plik dla każdego klienta zawierający zapytania, które ma wykonać w trakcie symulacji. Symulator tworzy procesy dla wszystkich klientów podczas startu, każdy z nich dołącza też do sieci P2P na samym początku. Jednak po wysłaniu wszystkich swoich zapytań klient opuszcza sieć bez informowania żadnego z węzłów.

4.1. Dane testowe

Jako dane wejściowe do symulatora użyte zostały realne dane pochodzące z działających centralnych serwerów cache'ujących. Niestety znalezienie aktualnych danych do testów jest obecnie niezwykle trudne ze względu na kwestie bezpieczeństwa i anonimowości użytkowników takich serwerów. Niedostępne są też dane, które używane są w podobnych pracach (wspomnianych w rozdziale 3.2) co powoduje, że porównanie systemu z innymi staje się trudne.

Dane, na których przeprowadzane były testy pochodzą z 2007 roku i są danymi zebranymi przez serwer cache'ujący IRCache[7]. Do testów użyte zostały trzy zbiory danych `sj.sanitized20070110`, `sv.sanitized20070110` oraz `uc.sanitized20070109`. Wszystkie użyte pliki zawierają zapytania wykonywane przez klientów w ciągu jednej doby. Liczba wysyłanych zapytań oraz liczba klientów dla poszczególnych plików przedstawiona została w tabeli 1. W tabeli 2 przedstawiona zostały statystyki dotyczące liczby wysyłanych zapytań przez pojedynczego klienta w przedstawionych zbiorach danych. Wartości w kolejnych kolumnach oznaczają odpowiednio minimalną liczbę zapytań wykonaną przez pewnego klienta, medianę i średnią z liczby zapytań wszystkich klientów oraz maksymalną liczbę zapytań przypadającą na klienta w tym zbiorze danych.

Plik z danymi	Liczba klientów	Łączna liczba zapytań	Liczba zapytań `GET`
`sj.sanitized`	114	348 189	346 405
`sv.sanitized`	91	447 696	447 621
`uc.sanitized`	160	544 370	527 355

Tabela 1: Szczegóły dotyczące użytych danych testowych.

Na potrzeby testów z danych zostały usunięte zostały zapytania inne niż `GET`, a czas został przeskalowany o czynnik 1/18, tak aby pojedyncza symulacja trwała około 1.5 godziny.

[7] `ftp://ftp.ircache.net/Traces/DITL-2007-01-09/`

	Liczba zapytań na klienta				
Plik z danymi	1. kwartyl	Mediana	Średnia	3. kwartyl	Max
`sj.sanitized`	17	150	2 273	1414	45 910
`sv.sanitized`	11	120	2 976	703	167 000
`uc.sanitized`	24	199	3 230	1766	87 390

Tabela 2: Szczegóły dotyczące liczby zapytań przypadających na pojedynczego klienta.

4.2. Cache'owanie w pamięci

Pierwszym testowanym podczas symulacji komponentem jest lokalny cache. Cache'owanie takie utrzymywane jest w pamięci w celu szybkiego dostarczenia danych, które były już pobierane wcześniej.

Przeprowadzone testy uwzględniały różne wielkości cache'u oraz różne algorytmy decydujące o tym, który element należy wyrzucić kiedy cachejest już pełny, a powinien do niego trafić nowy wpis.

W trakcie testów cache'owania w pamięci całkowicie wyłączony został komponent odpowiedzialny za cache'owanie w sieci P2P, zatem dla każdego z węzłów dane cache'owane były niezależnie.

Przetestowane zostały trzy algorytmy opisane w rozdziale 3.3 o wielkościach: 1024, 2048, 4096, 8192 oraz 16384. Wyniki testów dla poszczególnych zbiorów danych znajdują się na rysunku 10. Uśrednione wyniki dla wszystkich zbiorów danych testowych przedstawione zostały na rysunku 12.

4.3. Cache'owanie w sieci P2P

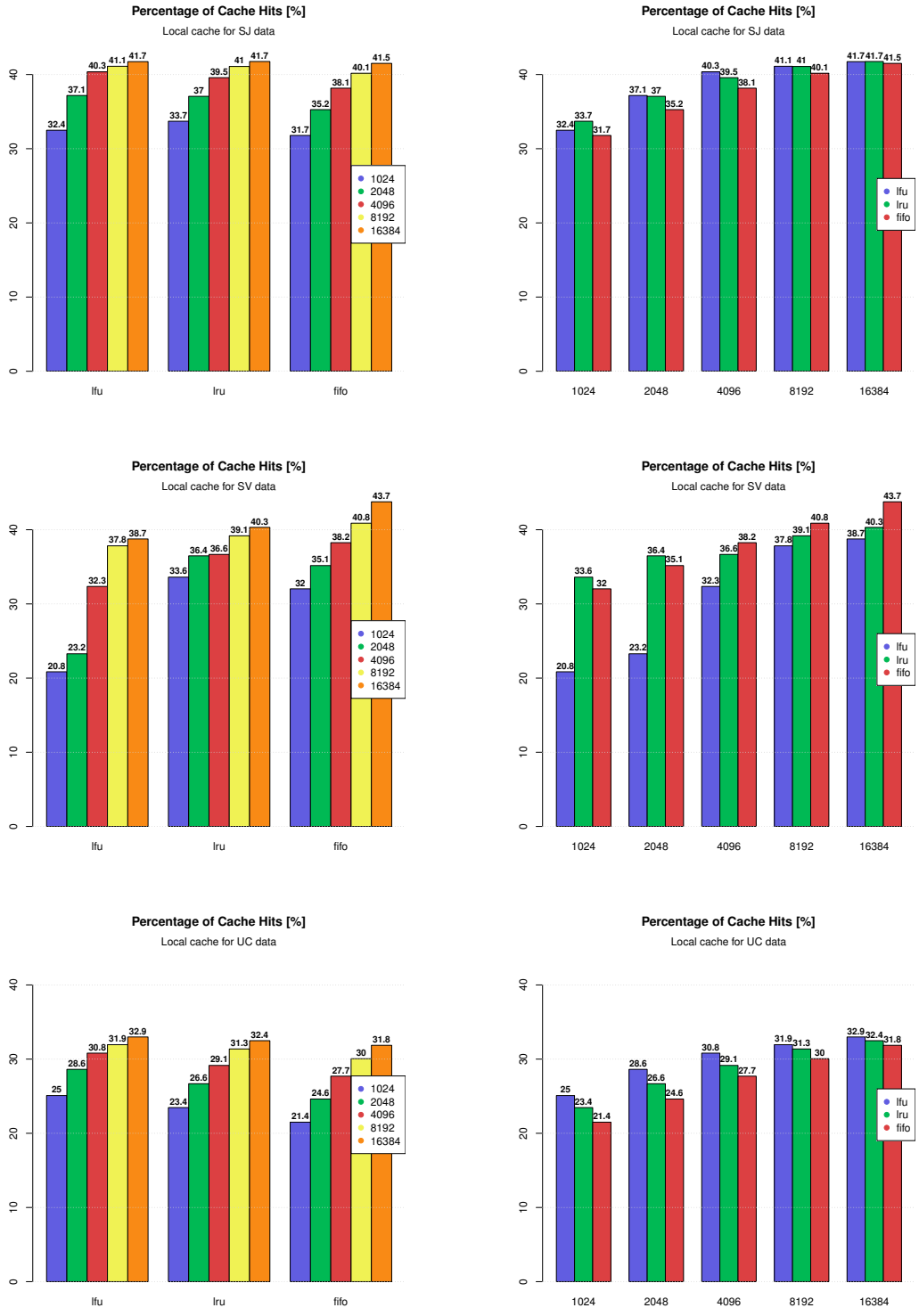
Drugim przedmiotem testów było działanie cache'u, który wykorzystuje sieć peer-to-peer. Celem testów jest określenie w jakim procencie dane, które pobierane są przez jednego użytkownika mogą zostać wykorzystane przez pozostałych.

Ponownie testy zostały przeprowadzone z uwzględnieniem różnej wielkości cache'u oraz różnych algorytmów wymiany danych w przypadku pełnego cache'u. Podczas testów wyłączony został zupełnie cache w pamięci, aby zapytania o wszystkie zasoby były wyszukiwane w sieci peer-to-peer.

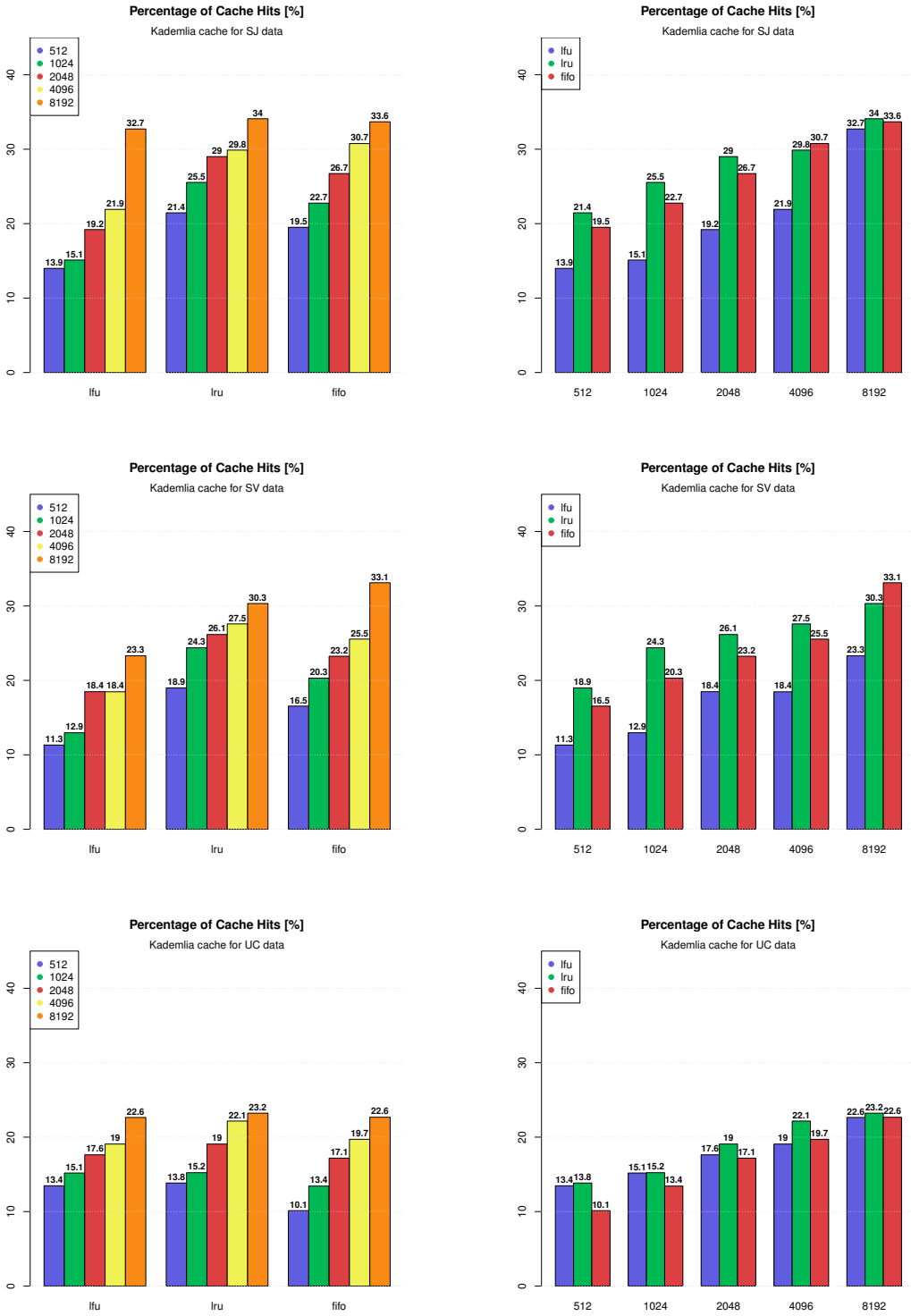
Ponieważ symulator, w celu odwzorowania realnego środowiska pracy, tworzy procesy, które komunikują się ze sobą przez sieć, znacząco obciąża to komputer, na którym testy są uruchomione. Dodatkowo *Kademlia* używa do komunikacji między węzłami protokołu `UDP`, który jest protokołem stratnym. Z powyższych powodów nie wszystkie wysłane zapytania są poprawnie obsłużone - część z wysyłanych pakietów nie dociera do celu, co skutkuje przekroczeniem limitów czasów zdefiniowanych w protokole *Kademli* i pomimo, że zasób może znajdować się w sieci istnieje szansa, że w czasie testów wyszukiwanie zasobu zakończy się niepowodzeniem. Dla przedstawionych wyników testów średnio 91% zapytań została obsłużona poprawnie. Pozostałe zakończyły się niepowodzeniem z powodu utraty pakietu `UDP` lub z powodu upłynięcia limitu czasu przewidzianego na odpowiedź ustalonego na 5 sekund.

Podobnie jak w przypadku testów w pamięci (rozdział 4.2) przetestowane zostały trzy algorytmy cache'owania , ale o wielkościach 512, 1024, 2048, 4096 i 8192.

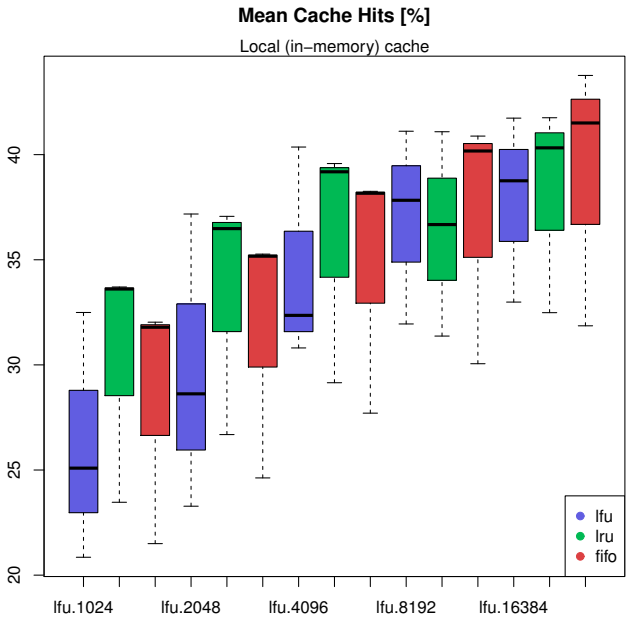
Na rysunku 11 przedstawione zostały średnie wartości procentu zasobów, które były cache'owane dla poszczególnych zbiorów danych. Wyniki pochodzą z kilku niezależnych uruchomień symulatora na różnych maszynach. Rysunek 13 przedstawia zebrane wyniki dla wszystkich zbiorów danych.



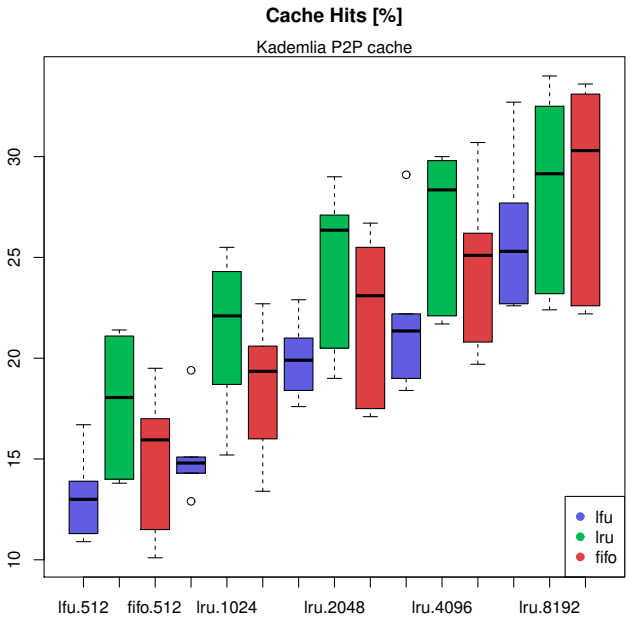
Rysunek 10: Wyniki testów cache'u lokalnego (w pamięci) dla przetestowanych zbiorów danych.



Rysunek 11: Uśrednione wyniki testów cache'u w sieci peer-to-peer dla przetestowanych zbiorów danych.



Rysunek 12: Zbiorczy wykres wyników testów cache'u w pamięci dla wszystkich zbiorów danych.



Rysunek 13: Zbiorczy wykres wyników testów cache'u w sieci peer-to-peer dla wszystkich zbiorów danych.

4.4. Cache'owanie dwupoziomowe

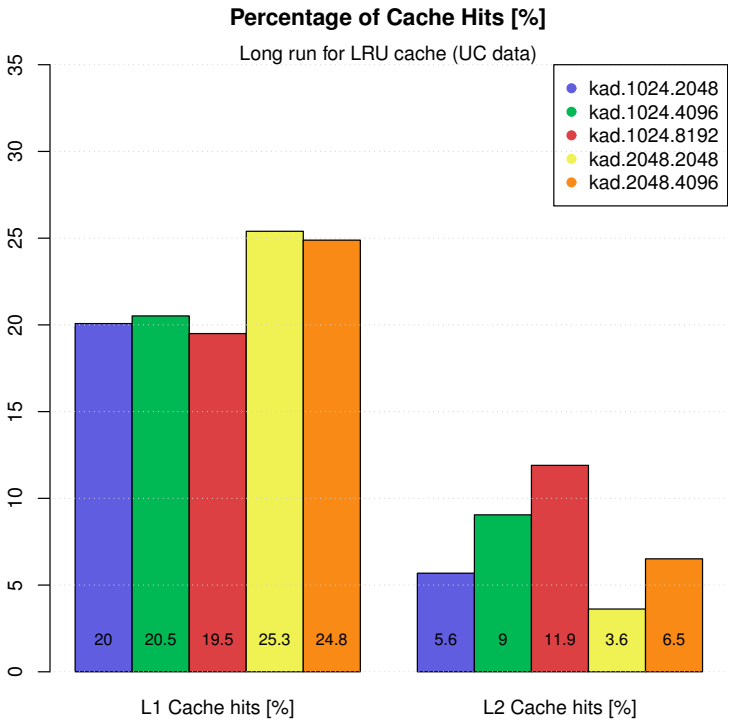
Ostatnie z przeprowadzonych testów dotyczą połączenia lokalnego cache'owania w pamięci z cache'owaniem w sieci peer-to-peer. W tym celu wykorzystywane jest cache'owanie dwupoziomowe - w pierwszej kolejności sprawdzany jest cache lokalny (cache pierwszego poziomu), a dopiero w przypadku, gdy zasób nie zostaniem znaleziony w pamięci, zapytanie wysyłane jest do sieci peer-to-peer (cache drugiego poziomu).

W celu uruchomienia testów i zebrania danych odpowiednich do przedstawienia wyniku opóźnień (rysunek 15) w tym przypadku testy zostały wykonane na danych, w których czas zapytań został przeskalowany o czynnik ⅙. Dzięki temu wyniki są bardziej miarodajne, ponieważ obciążenie komputera, na którym są uruchomione testy jest mniejsze.

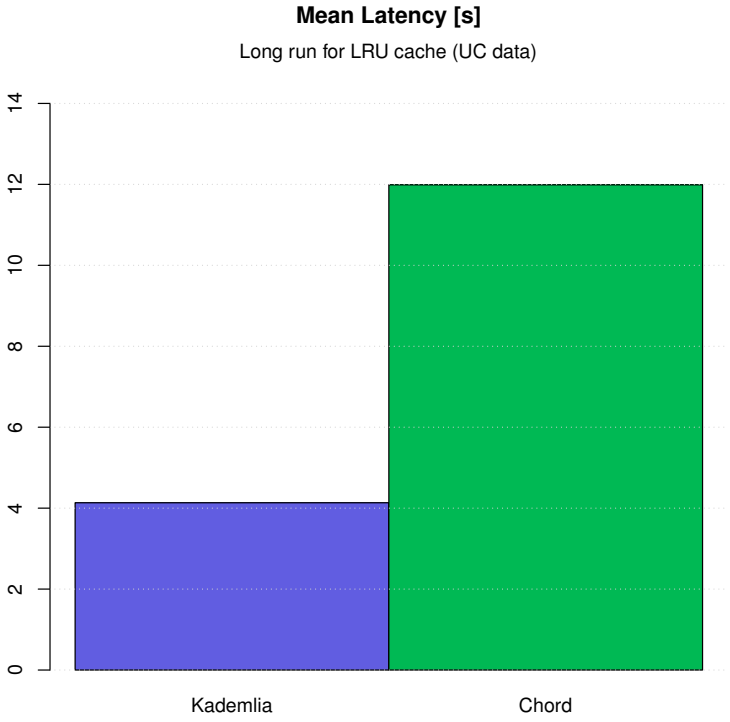
Testy zostały przetestowane dla kilku różnych kombinacji parametrów. We wszystkich przypadkach jako polityka cache'owania używany był algorytm LRU. Na wykresie została użyta następująca notacja:

$$network.mem\_size.p2p\_size,$$

gdzie *network* oznacza implementację DHT (`kad` - Kademlia, `chord` - Chord), *mem_size* wielkość cache'u lokalnego (w pamięci), a *p2p_size* oznacza wielkość cache'u w sieci peer-to-peer. Na przykład: `kad.1024.2048` oznacza symulację, dla której wielkość cache'u w pamięci to 1024, w sieci peer-to-peer 2048, a jako implementacja DHT użyta została Kademlia. Wyniki testów przedstawione zostały na rysunku 14.



Rysunek 14: Porównanie liczby trafień w przypadku użycia cache'u dwupoziomowego.



Rysunek 15: Porównanie średniego opóźnienia pomiędzy Kademlią i Chordem.

4.5. Wnioski

Wyniki testów dla cache'u lokalnego znacząco różnią się w zależności od zbioru danych. Oczywiście w przypadku wszystkich algorytmów cache'owania zwiększanie rozmiaru cache'u powoduje wzrost liczby trafień (ang. *Cache hit*). Dla wszystkich danych można zaobserwować ciekawą zależność: dla wielkości powyżej 2048 liczby trafień dla wszystkich polityk cache'owania się wyrównują. Związane jest to z tym, że dla większości klientów liczba zapytań jest większa od 2^{11} i mniejsza od 2^{12}.

Dla danych `UC` najlepsze wyniki daje cache LFU. Podobnie jest dla zbioru `SJ`, chociaż tam wyniki poszczególnych algorytmów są bardziej wyrównane. W przypadku `SV` do wspomnianej już wartości granicznej 2048 najlepiej radzi sobie polityka LRU, a dla pozostałych wartości lepsze wyniki daje zastosowanie FIFO. Wyniki są o tyle ciekawe, że LRU i FIFO działają bardzo podobnie, LRU pozwala jedynie na uaktualnienie pozycji w przypadku odwołania do elementu, który znajduje się już w cache'u. Jednak w specyficznym przypadku danych `SV` własność ta obniża liczbę trafień. Wartości liczby trafień są podobne dla zbiorów `SJ` i `SV`, LRU pozwala na uzyskanie od 33% dla cache'u wielkości 1024 do 41% trafień dla cache'u wielkości 16384. Dla zbioru `UC` wyniki wynoszą odpowiednio 23% i 32%.

Uśrednione wyniki dla wszystkich zbiorów danych przedstawione na rysunku 12 pokazują, że w średnim przypadku najlepiej zachowuje się polityka LRU. Wynik ten potwierdza poprzednio uzyskane wyniki [2] i uzasadnia popularność tego algorytmu w systemach cache'owania.

W przypadku cache'owania w sieci peer-to-peer ponownie zwiększenie rozmiaru cache'u bezpośrednio wpływa na liczbę trafień. Również tutaj najlepsze wyniki oferowane są przez politykę LRU.

Główną różnicą w wynikach dla sieci peer-to-peer są wartości dla liczby trafień. Wyniki dla LRU dla zbiorów `SV` i `SJ` wynoszą od około 20% w przypadku cache'u o wielkości 512 do 30% dla wielkości 8192. Dla zbioru `UC` wyniki, podobnie jak w przypadku cache'owania w pamięci, są nieco gorsze od pozostałych zbiorów i kształtują się na poziomie od 14% do 23%.

Wyniki przedstawione na wykresie 14 pokazują, że użycie cache'u dwupoziomowego pozwala na poprawienie wyników uzyskanych w przypadku cache'u w pamięci o około ¼ (np. w przypadku testów `kad.2048.4096` z 25% procent do 31%). Bardzo wyraźnie widać też wzrost liczby trafień w cache drugiego poziomu w przypadku obniżenia rozmiaru cache'u w pamięci.

Na wykresie 15 można zaobserwować ogromną dysproporcję pomiędzy średnim opóźnieniem na obsługę zapytania w przypadku Kademli i Chorda. Jedną z przyczyn takiej różnicy jest brak równoległych zapytań w przypadku Chorda - jeżeli węzeł, którego chcemy odpytać nie jest już dostępny to protokół musi poczekać na upłynięcie ustalonego limitu czasowego i dopiero wtedy rozpocząć odpytywanie innego węzła. Drugi powodem jest słaba jakość biblioteki użytej do obsługi Chorda[8], wszystkie procedury są tam zaimplementowane w sposób blokujący, więc w przypadku, gdy wiele węzłów opuści sieć zapytania blokowane są przez oczekiwanie na odpowiedź z poprzednio wysłanych żądań.

5. Implementacja

W niniejszym rozdziale opisujemy implementację rozproszonego cache'u z użyciem DHT. W rozdziale 5.1 przedstawimy podjęte próby innej formy implementacji, a w rozdziale 5.2 opiszemy ostatecznie wybrane rozwiązanie. Rozdział 5.5 prezentuje informacje o instalacji i uruchomieniu stworzonego programu.

5.1. 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, 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 5.2 podjęte zostały próby utworzenia pluginu do przeglądarki, opisane w rozdziałach 5.1.1 i 5.1.2.

5.1.1. Plugin do przeglądarki

Pierwszą podjętą próbą implementacji było stworzenie rozszerzenia do przeglądarki Chrome[9], które korzystałoby z nowego API dla połączeń Peer-to-peer, zaproponowanego w HTML5[10].

Jako *proof-of-concept* stworzony został plugin, którego kod znajduje się na listingu 1 w załączniku A.1.

Rozwiązanie korzysta z API `chrome.webRequest.onBeforeRequest`[11], 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 ½ zostaje zastąpione obrazkiem `cowImg`, który znajduje się w pliku źródłowym w zakodowanej postaci

[8] `http://code.google.com/p/pyrope/`
[9] `http://chrome.google.com`
[10] `http://www.w3.org/TR/2008/WD-html5-20080122/#peer-to-peer`, `http://www.w3.org/TR/webrtc`
[11] `http://developer.chrome.com/extensions/webRequest.html`

(`base64`) oraz z prawdopodobieństwem $1/2$ jednym z obrazków z tablicy `loldogs`. Dzięki przekierowaniu na zasób w postaci `base64` nie jest konieczne wykonywanie dodatkowego żądania 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, należy korzystać z opcji `blocking`, której włączenie 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.

5.1.2. 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. *sandboxed 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*.

5.2. 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 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`[12] oraz `Entangled`[13]. Aplikacja składa się z dwóch części:

Proxy - serwer 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 9 w rozdziale 3.3 przedstawione zostało przykładowe zapytanie i jego droga przez poszczególne komponenty aplikacji.

5.3. Symulator

W celu wykonania testów, których wyniki przedstawione zostały w rozdziale 4 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 symulatora (skrypt `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.

[12] `http://twistedmatrix.com`
[13] `http://entangled.sourceforge.net`

5.4. Skrypty pomocnicze

Aby ułatwić przetwarzania plików z logami stworzono zestaw pomocniczych skryptów dla środowiska `R`[14].

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 bezwzględnego 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 są 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.

5.5. Instalacja i uruchomienie

Aplikacja, jej źródła oraz niniejsza praca dostępne są na dołączonej płycie CD oraz publicznym repozytorium[15]. W celu uruchomienia aplikacji wymagany jest `Python` w wersji `2.7` oraz pakiety `python2.7-twisted` i `python2.7-simplejson`. Aby pobrać źródła aplikacji można skorzystać z polecenia `$ git clone git@bitbucket.org:tomusdrw/mgr.p2p.proxy.git`.

Uruchomienie aplikacji odbywa się przez wywołanie pliku `main.py`, znajdującego się w głównym katalogu. Listing 2 w załączniku A.2 przedstawia dostępne opcje konfiguracyjne.

Wywołanie symulatora możliwe jest przez uruchomienie programu `simulator_main.py`. Symulator przyjmuje podobne opcje co główna aplikacja. Dodatkowo do pracy symulatora potrzebne są dane wejściowe, które powinny znajdować się w katalogu `simulator/data/`. Przykładowe dane zostały załączone w repozytorium.

Dodatkowo w celu uruchomienia wielu symulacji utworzone zostały pomocnicze skrypty dla konsoli `bash`: `run_nomem_simulations.sh` oraz `run_nop2p_simulations.sh`. Pozwalają one na uruchomienie symulacji odpowiednio cache'u lokalnego i cache'u w sieci peer-to-peer dla różnych algorytmów cache'owania i wielkości cache'y.

6. Podsumowanie

Jak pokazały wyniki testów opracowany w pracy system, dzięki połączeniu cache'owania w pamięci z cache'owaniem w sieci peer-to-peer, pozwala na zwiększenie liczby trafień o około 25%. Dodatkowo zastosowanie sieci Kademlia pozwala na uzyskanie niewielkich opóźnień w stosunku do Chorda.

Zarówno na podstawie wyników testów cache'u lokalnego, jak i testów cache'u w sieci peer-to-peer możemy uznać, że w średnim przypadku najlepsze wyniki pod względem liczby trafień daje algorytm LRU.

W momencie pisania tej pracy nie udało się zaimplementować uproszczonego rozwiązania, które chętnie byłoby instalowane przez użytkowników. Dostępne API i dokumentacja nie są jeszcze wystarczająco dojrzałe do stworzenia takiego systemu. Trwają jednak prace nad kolejnymi wersjami

[14] `http://www.r-project.org`

[15] `https://bitbucket.org/tomusdrw/mgr.p2p.proxy`

specyfikacji nowych możliwości przeglądarek, które dają nadzieję na stworzenie podobnej aplikacji w przyszłości.

Kolejnymi krokami, które możemy podjąć w celu dalszej analizy przedstawionego rozwiązania jest na przykład opracowanie algorytmu decydującego kiedy dany zasób powinien być składowany w cache'u lokalnym, a kiedy warto wysłać go do sieci peer-to-peer (obecnie zasób zawsze składowany jest na obu poziomach). Kolejnym z możliwych usprawnień jest składowanie zasobów w postaci "paczek", na podstawie bliskości wysyłanych zapytań w czasie - pobierając stronęinternetową użytkownik w krótkim czasie powinien też pobrać obrazki i skrypty znajdujące się na tej stronie. Umieszczenie paczki, która potencjalnie zawiera całą stronę internetową, w sieci peer-to-peer pozwoli na zmniejszenie opóźnień i liczby zapytań wewnątrz sieci. Oczywiście warte odnotowania byłoby porównanie innych implementacji DHT, takich jak Pastry, Tapestry CAN czy Skipnet [6].

A. Załączniki

A.1. Kod pluginu do *Chrome*

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

Listing 1: Kod rozszerzenia dla przeglądarki Chrome, realizujący *proof-of-concept*.

A.2. Opcje konfiguracyjne aplikacji

```
usage: main.py [-h] [--version] [-P port] [--no-proxy] [--spawn N]
               [--bootstrap HOST:PORT] [--known-nodes filename]
               [--log {info,debug,warn}] [--no-p2p]
               [--p2p-network {kademlia,chord}] [--p2p-port P2P PORT]
```

```
                [--mem-algo {lru,lfu,fifo}] [--mem-size MEM_SIZE] [--no-mem]
                [--p2p-algo {lru,lfu,fifo}] [--p2p-size P2P_SIZE]

Run and test distributed caching proxy.

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  -P port               Proxy port
  --no-proxy            Create only p2p node without proxy.
  --spawn N             Spawn some nodes that will be in P2P network
  --bootstrap HOST:PORT
                        Provide known bootstrap node for P2P network.
  --known-nodes filename
                        Provide filename with known nodes in P2P network.
  --log {info,debug,warn}
                        Change logging mode
  --no-p2p              Disable connecting to p2p network and p2p caching
  --p2p-network {kademlia,chord}
                        Change p2p network implementation
  --p2p-port P2P PORT   Port for P2P network node. In case of spawn it would
                        be start port.
  --mem-algo {lru,lfu,fifo}
                        Change memory cache algorithm
  --mem-size MEM_SIZE   Change memory cache queue size
  --no-mem              Disable memory cache (use only P2P)
  --p2p-algo {lru,lfu,fifo}
                        Change p2p cache algorithm
  --p2p-size P2P_SIZE   Change p2p cache queue size
```

Listing 2: Dostępne opcje konfiguracyjne głównej aplikacji.

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] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

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

[4] Florence Clévenot and Philippe Nain. A simple fluid model for the analysis of the squirrel peer-to-peer caching system. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1. IEEE, 2004.

[5] Lei Guo, Songqing Chen, and Xiaodong Zhang. Design and evaluation of a scalable and reliable p2p assisted proxy for on-demand streaming media delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 18(5):669–682, 2006.

[6] Nicholas JA Harvey, Michael B Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, volume 4, pages 9–9, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] Prakash Linga, Indranil Gupta, and Ken Birman. Kache: Peer-to-peer web caching using kelips. *ACM Transactions on Information Systems (under submission)*, 2004.

[11] 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.

[12] Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 54–63. ACM, 2004.

[13] 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.

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

[15] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured p2p systems. In *Peer-to-Peer Systems II*, pages 68–79. Springer, 2003.

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

[17] 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.

[18] Ramin Sadre and Boudewijn R Haverkort. Changes in the web from 2000 to 2007. In *Managing Large-Scale Service Deployment*, pages 136–148. Springer, 2008.

[19] Stefan Saroiu, P Krishna Gummadi, and Steven D Gribble. Measurement study of peer-to-peer file sharing systems. In *Electronic Imaging 2002*, pages 156–170. International Society for Optics and Photonics, 2001.

[20] Weisong Shi, Kandarp Shah, Yonggen Mao, and Vipin Chaudhary. Tuxedo: A peer-to-peer caching system. In *PDPTA'03: Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.

[21] 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.

[22] Wikipedia. Global internet usage, 2013. [Online; accessed 17-July-2013].

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