AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

dHTTP – Rozproszony system wsparcia serwerów sieci web

dHTTP – Distributed companion for central-server based web

Autor: *Dominik Adamiak*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2018

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję promotorowi za wolność działania, a rodzinie i przyjaciołom za wiarę w to, że się uda.

Spis treści

1. Wprowadzenie

Dzisiejsza informatyka jak nigdy stoi przed wyzwaniami związanymi z wydajnością, wymuszanymi poprzez miliony klientów z szerokopasmowym dostępem do sieci. Ich rozwiązywanie nie jest już możliwe przez wzmacnianie podzespołów pojedynczych komputerów – wymaga odpowiedniego sposobu projektowania, który skupia się na możliwościach maksymalnego rozłożenia i łatwego skalowania obciążenia. Ta praca proponuje rozwiązanie, które wpisuje się w ten motyw i w przystępny klientom sposób może znacznie odciążyć rozwijające się części sieci web.

1.1. Cele pracy

Celem tej pracy jest stworzenie protokołu i oprogramowania pozwalającego na rozproszony i bezpieczny dostęp do współdzielonych zasobów, w warstwie użytkowej odpowiadającego obecnemu Hypertext Transfer Protocol (HTTP). W toku pracy, system określany będzie skrótem **dHTTP** – distributed HTTP.

Głównym założeniem projektu jest rozłożenie obciążenia stron, które nagle zyskują popularność – często konieczna dla właścicieli takich stron jest inwestycja w szybsze łącza czy więcej sprzętu, wiążąca się ze sporymi kosztami. Kosztami które mogą zresztą nie znaleźć uzasadnienia – dodatkowy sprzęt pomoże w chwilach szczytowego ruchu, będzie jednak przez większość czasu leżeć odłogiem, wciąż generując koszty łącz i prądu.

Częściowym rozwiązaniem tego problemu są chmury i udostępniany przez nie *autoscaling* – użytkownicy, zamiast utrzymywać własną infrastrukturę, mogą nie tylko zdać się na serwery zarządzane przez zewnętrzny podmiot, ale także wykorzystać mechanizmy automatycznego rozszerzania i zmniejszania ilości urządzeń czy wykorzystywanego łącza, w zależności od obciążenia całego klastra. To pozwala na drastyczną minimalizację kosztów i jest rozwiązaniem coraz chętniej stosowanym także przez podmioty o gigantycznym ruchu sieciowym, jak serwis streamingowy Netflix ([1]).

Wciąż jednak są to rozwiązania obciążające właściciela witryny, co może stanowić problem w przypadku projektów hobbystycznych czy krajów rozwijających się – koszt autoscalingu w czasach szczytu może okazać się nieproporcjonalnie wysoki w stosunku do czasów spokojnego ruchu.

System zaprezentowany w poniższej pracy – dHTTP – ma posłużyć jako samoskalująca się, rozproszona alternatywa do tego podejścia. Bierze on pod uwagę optymalne rozłożenie danych,

szybki i możliwie najmniej scentralizowany sposób współdzielenia informacji o stanie systemu (*kto jest online i może dać mi plik* `logo.png` *z hosta* `example.com`*?*) i aktualizacjach zawartości, oraz stronę kliencką, będącą w tym przypadku wtyczką do przeglądarki Google Chrome, nakładającą warstwę dHTTP na polecenia pobierania zasobów. Z punktu widzenia użytkownika system jest *przezroczysty*: zostanie węzłem i klientem sieci dHTTP polega wyłącznie na instalacji wtyczki, działającej autonomicznie w tle. Oprócz aplikacji klienckiej, dHTTP udostępniony jest także w trybie *headless* – węzeł może zostać częścią klastra w sposób automatyczny, bez użycia przeglądarki.

1.2. Zawartość

Rozdział 1. stanowi krótkie wprowadzenie i definicję celu pracy.

Rozdział 2. zawiera rozważania na temat historycznych i teoretycznych podstaw systemów rozproszonych, oraz przykłady najpopularniejszych narzędzi je implementujących. Wprowadza także do konceptualnych i narzędziowych podstaw implementacji systemu dHTTP.

Rozdział 3. jest szczegółowym opisem wymagań, projektu, modułów, a także implementacji samego systemu, w szczegółach omawiającym funkcjonalności, biblioteki i narzędzia pozwalające na działanie aplikacji, napotkane w toku projektowania problemy i ich rozwiązania, rys architektoniczny sposobu przechowywania danych czy ich propagacji, z uwzględnieniem implementacji niezależnego węzła (*headless mode*) i rozwiązania dla klienta końcowego.

Rozdział 4. to próba walidacji systemu dHTTP – analiza użytkowości systemu, zbiór badań nad wydajnością (z uwzględnieniem różnych mechanizmów propagacji danych) i propozycji na jej dalsze optymalizacje.

Rozdział 5. zawiera krótkie podsumowanie, podkreślające osiągnięte cele i potencjalne kierunki dalszego rozwoju projektu.

2. Droga do rozproszenia

Prawo Moore'a ([2]), wspominające o podwajaniu ilości tranzystorów w procesorach, często parafrazowane jako podwajanie ich mocy obliczeniowej, przestaje działać, podczas gdy złożoność problemów i ilość użytkowników szerokopasmowego internetu rośnie. Próba skalowania wertykalnego – polegającego na wzmacnianiu pojedynczych węzłów – przestaje zdawać próbę czasu.

Problem posiada także drugą stronę – nie wszyscy użytkownicy internetu mogą pozwolić sobie na łącze szerokopasmowe. Szacuje się, że w roku 2017 dostęp do internetu posiada ponad połowa populacji świata; sam wzrost ilości użytkowników sieci Web z Afryki od roku 2000 do 2017 wynosi aż 8503.1% ([3]) – znaczną część tych połączeń stanowią jednak połączenia starych generacji sieci komórkowej, o znacznie ograniczonej przepustowości i ogromnych latencjach. Każdy kolejny węzeł niezbędny do połączenia użytkownika z serwerem końcowym może dokładać cennych milisekund czasu odpowiedzi.

Autorzy IPFS zauważają w swojej pracy ([4]) także inne słabości obecnego internetu – sieć oparta o HTTP jest w prawdzie zdecentralizowana, jako iż treści rozdzielane są pomiędzy miliony węzłów, od gigantycznej sieci Amazon Web Services aż po mikroserwery stojące w domach pasjonatów; brakuje jej jednak faktycznego rozproszenia: ta infrastruktura nie jest gotowa *by design* na przyjmowanie gigantycznego ruchu, nie jest w stanie efektywnie przechowywać i udostępniać wielkich zestawów danych; jest również podatna na znikanie danych, jako iż awaria pojedynczego dysku twardego może zatrzymać udostępnianie całej witryny.

2.1. Zarys historyczny

W świecie informatyki szybko wyewoluowały **rozwiązania współbieżne**. Istnienie wielu wątków – niezależnych od siebie logicznie toków wywołań, operujących na wspólnej pamięci – rozwiązywało takie problemy, jak nierówny rozmiar żądań – działający sekwencyjnie serwer blokowałby się przy poleceniach zajmujących więcej czasu. Dzięki przeplataniu wątków można uniknąć tej sytuacji, a także pozwolić na iluzję symultaniczności – mimo korzystania z jednego procesora, polecenia wykonywane są *pseudorównolegle*, co pozwala między innymi na responsywność interfejsów użytkownika.

Upowszechnienie się komputerów wieloprocesorowych i procesorów wielordzeniowych zaowocowało rozwojem **obliczeń równoległych**. Istotne jest rozgraniczenie tych dwóch pojęć – współbieżność jest raczej paradygmatem, sposobem strukturyzacji oprogramowania w sposób,

który pozwala na wykonywanie wielu poleceń niezależnie i jednocześnie; równoległość z kolei to możliwość uruchamiania tego typu oprogramowania w tym samym czasie, dzięki mnogiej liczbie procesorów ([5]).

Współbieżność i równoległość rozwiązują problemy obciążenia, ułatwiając operacje na współdzielonej pamięci. Istnieją całe klasy algorytmów naturalnie podatnych zrównoleglaniu: mapowanie, grupowanie, przeszukiwanie czy sortowanie są jednymi z przykładów. Choć w niektórych przypadkach adaptacja algorytmu wymaga jego konkretnej modyfikacji, możliwość rozwiązywania składowych problemu jako niezależnych wątków znacznie przyspiesza wykonanie w środowiskach wieloprocesorowych ([6]).

Skalowanie rozwiązań tego typu może jednak trafić na ścianę kosztów – niezbędne są duże ilości pamięci RAM i szybkie dyski; cierpi też niezawodność systemu, gdyż jego działanie pozostaje kompletnie uzależnione od utrzymania przy życiu konkretnego komputera. Jeśli komputer musi też propagować dane w sieci, cała komunikacja spoczywa na nim – i wydajność jego łącza może zawieść w przypadku szczytów obciążenia.

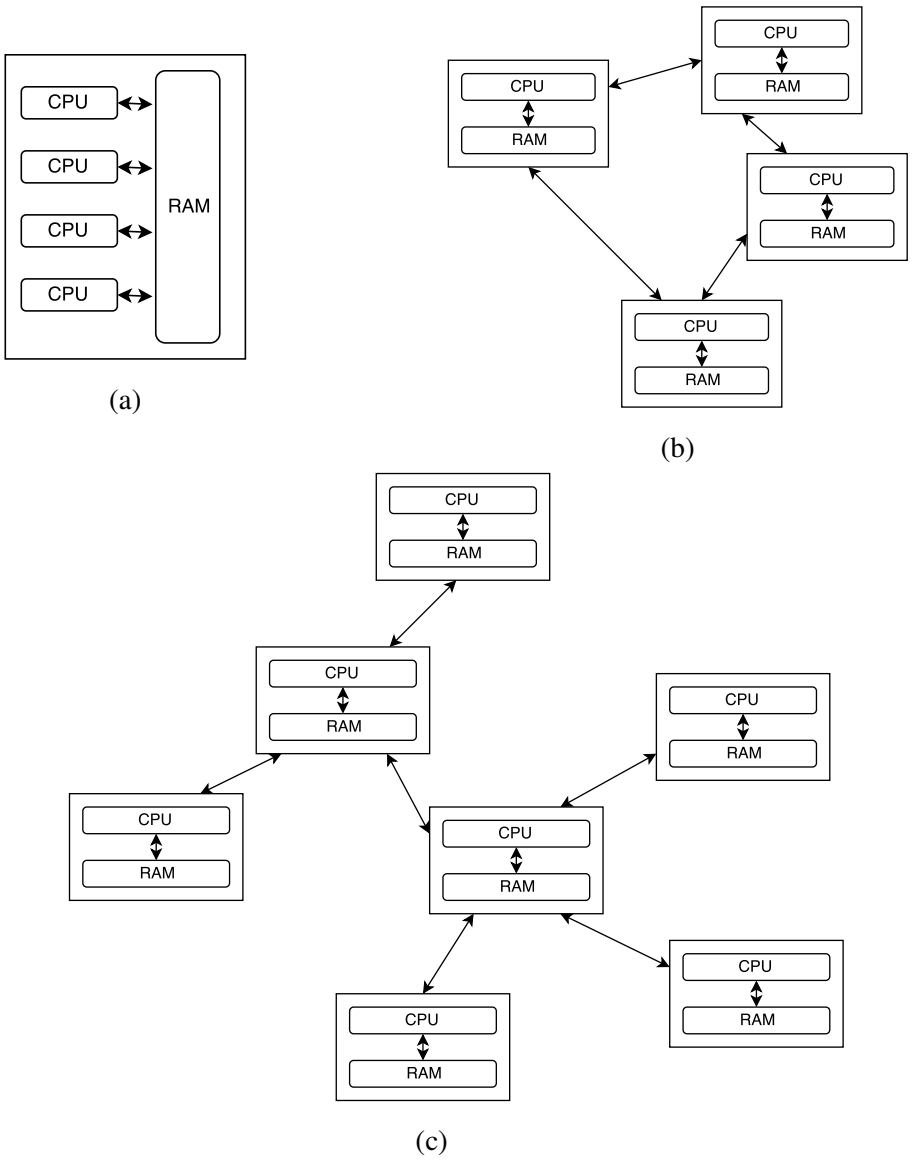
Remedium dla powyższych problemów jest próba rozdzielenia pracy pomiędzy wiele komputerów – tu pojawia się **decentralizacja**, podział problemu na podproblemy, którymi zarządzać mogą mniejsze węzły główne, mające dostęp do węzłów roboczych. To rozwiązanie stawia też nowe wyzwania przed projektantami oprogramowania, którzy nie mogą już polegać na współdzielonej pamięci – dane muszą być przekazywane jakąś sieciową metodą komunikacji.

Obliczenia decentralizowane uznać można za podklasę **obliczeń rozproszonych**. Dla potrzeb tej pracy, obliczenia rozproszone zdefiniowane zostaną jako prowadzone przez w pełni autonomiczne węzły, które dzielą między sobą pracę i radzą sobie z pojawianiem się i znikaniem kolejnych elementów sieci; innymi słowy, w warstwie logicznej system rozproszony powinien zachowywać się identycznie zarówno przy istnieniu zaledwie jednego węzła sieci, jak i przy milionie – różnicą powinien być wyłącznie spadek wydajności systemu.

2.2. Narzędzia w rozproszeniu danych

Najbardziej istotnym problemem z dziedziny rozproszenia dla systemu dHTTP jest rozproszenie danych – próba propagowania informacji pomiędzy wiele węzłów sieci, w celach takich jak podniesienie wydajności, obniżenie kosztów, zwiększenie dostępności czy zapewnienie trwałości informacji.

Dobrym przykładem rozwiązania rozproszonego w operacjach na danych jest system kontroli wersji `git`. Został stworzony przez Linusa Torvaldsa, autora jądra Linux, jako otwarta alternatywa dla systemu BitKeeper, który wycofał z dystrybucji darmową wersję swojej aplikacji projektom open source. `git` jest projektem czysto rozproszonym – każdy użytkownik posiada kompletną kopię repozytorium, z informacjami o wszystkich zmianach i ich autorach, na której może wykonywać operacje w trybie offline; nie jest uzależniony od istnienia centralnego serwera (choć jego stosowanie jest powszechne; każdy użytkownik o aktualnej wersji



Rys. 2.1. Wizualizacja różnicy pomiędzy systemem równoległym (a), rozproszonym (b) i zdecentralizowanym (c).

repozytorium mógłby jednak pełnić jego rolę), a zmiany mogą być propagowane na różne sposoby, takie jak email czy przekazywanie łat w formie tekstowej. Dzięki swojej wydajności i silne zdecentralizowanemu designowi, `git` jest chętnie wybieranym rozwiązaniem nie tylko w środowiskach programistów, a w środowiskach profesjonalnych wypiera konkurencyjne rozwiązania, takie jak Apache Subversion czy CVS ([7]).

Nie zawsze jednak motywacje stojące za rozpraszaniem danych poza centralne serwery były szczytne. Duży wpływ na rozwój technik stosowanych obecnie – w tym wykorzystywanych przez dHTTP – mają rozwiązania rozpowszechnione głównie przez piractwo komputerowe. Systemy peer-to-peer – polegające na współpracy równorzędnych węzłów – zostały spopularyzowane dzięki aplikacji Napster, która w roku 2001 zgromadziła około 80 milionów użytkowników. Zaprojektowana jako system współdzielenia plików, wyspecjalizowała się w łatwym udostępnianiu plików MP3. Napster wzburzył wiele kontrowersji – w systemie pojawiały się niewydane jeszcze utwory znanych artystów, prowadząc do milionowych strat i procesu, który

pogrążył działanie systemu. Problemem Napstera była architektura oparta o centralny serwer indeksujący – każdy podpięty węzeł informował o posiadanych przez siebie plikach, a punkt centralny był niezbędny do przeszukiwania bazy plików i przekazywania poleceń pobrania innym węzłom. To rozwiązanie stanowiło słaby punkt sieci, i pozwoliło obciążyć twórców programu odpowiedzialnością za szerzone w nim treści.

Błędy Napstera zostały zauważone przy kolejnych projektach tego typu. Gnutella, chcąc uniknąć istnienia *single point of failure*, działała na zasadzie rozgłaszania poleceń użytkowników do wszystkich maszyn w sieci, co owocowało jednak drastycznym spadkiem wydajności ([8]).

Alternatywne podejście podjęte zostało przez sieć Freenet, gdzie zastosowano heurystyczny routing oparty o klucze; każdy plik otrzymuje klucz, a podobne klucze lokowane były na zbliżonych do siebie węzłach, dzięki czemu przeszukiwanie sieci nie wymagało punktu centralnego, a polecenia kierowane były w sposób niewymagający wielu przeskoków. Niestety, ta metoda nie gwarantowała znalezienia danych ([9]).

Chcąc pogodzić kwestie bezpieczeństwa, wydajności i spójności systemu, rozpoczęte zostały intensywne prace nad rozproszonymi tablicami haszującymi (distributed hash table – **DHT**). To rozwiązanie pozwala na trzymanie dużych ilości informacji w sposób kompletnie rozproszony – wszystkie węzły sieci znają zestaw kluczy, stanowiących *de facto* adresy danych, pozwalające użytkownikom na przeszukania; znalezienie węzłów trzymających właściwe dane jest umożliwione dzięki ich opartemu o klucze sposobie identyfikacji. Choć przechowywanie tablicy DHT jest dodatkowym kosztem obciążającym system, i wymagane jest ustalenie wspólnych funkcji hashujących, przy właściwej konfiguracji to rozwiązanie pozwala na obsługę całej sieci danych bez serwerów pośredniczących.

2.3. Podstawa projektu

Wspomniane rozwiązania stanowią kanwę dla projektu IPFS ([4]), który stara się tworzyć efektywny, realnie rozproszony system plików, obsługujący gigantyczne zasoby danych z niskimi latencjami, redundancją danych i bez wymogu zaufania uczestnikom sieci. Dane przechowywane są w drzewie Merkla, dzięki czemu przeszukiwanie sieci ma niską złożoność obliczeniową. Projekt skupia się na warstwie infrastruktury – po stronie użytkowej przypomina nieco system kontroli wersji `git`, a dane indeksowane są dzięki sumom kontrolnym, i ich adresy to wieloznakowe hashe. Takie podejście doskonale sprawdza się w skryptach i przy pracach prowadzonych przez ludzi obytych z tego typu narzędziami, nie jest to jednak system przyjazny użytkownikom domowym.

Celem tej pracy jest adaptacja dokonań projektu IPFS jako „przezroczysty” system wsparcia HTTP. W swoim założeniu, system nie może próbować „zastąpić” istniejących rozwiązań. Internet w obecnym kształcie nie jest gotowy na rewolucje – przykładem może być poziom przyjęcia się IPv6 ([10]), następnej generacji protokołu IP, która ma między innymi rozwiązać problem rozmiaru puli adresów. Brak realnej kompatybilności wstecznej sprawia, że konieczne

są rozwiązania przejściowe. dHTTP jest bardziej nakładką, rozwinięciem obecnych rozwiązań – współżyje z nimi – co daje perspektywy spokojnego przyjęcia takiego rozwiązania przez rynek.

IPFS posiada wzorcową implementację w języku Go, i jest podstawą stosu sieciowego `libp2p`. Zbiór bibliotek udostępniony w projekcie `js-libp2p` stanowi fundament projektu dHTTP, który, dzięki natywnej implementacji w JavaScripcie, może być w pełni uruchamialny z poziomu przeglądarki, zapewniając użytkową przezroczystość odbiorcy.

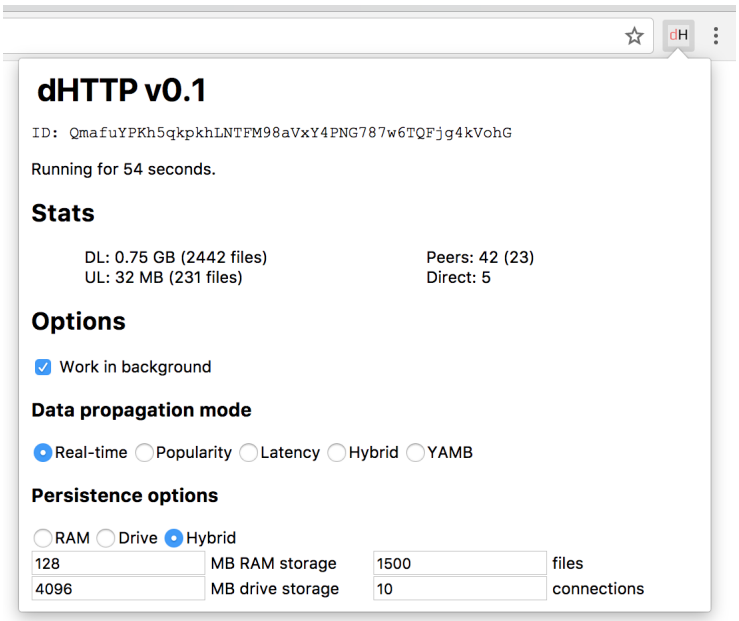
3. Projektowanie i implementacja

Nakreślony problem – rozproszonego systemu, który dzięki propagacji danych między użytkownikami obniży obciążenie serwerów sieci Web – stawia wiele wyzwań projektowych i architektonicznych.

3.1. Wymagania funkcjonalne

System dHTTP, z punktu widzenia użytkownika, ma spełniać jedną funkcjonalność: utrzymać lub poprawić płynność dostępu do interesujących go witryn internetowych, nie wpływając na ich treść i nie naruszając prywatności.

Projekt udostępnia interfejs zapewniający dostęp do statystyk, a także preferencji użytkownika. Udostępnione preferencje dotyczą: stopnia działania aplikacji w tle, trybów propagacji i przechowywania danych.



Rys. 3.1. Wstępna implementacja interfejsu operacji dla systemu dHTTP: pop-up dostarczany przez wtyczkę do przeglądarki Google Chrome pozwala na obserwację statystyk i zmianę preferencji.

Wymogiem dla projektu, koniecznym z racji potrzeby prostej automatyzacji rozrostu sieci, jest tryb niezależny aplikacji – *headless mode* – pozwalający na wystartowanie niezależnego

węzła jednym poleceniem. Istotnym jest, aby węzeł tego typu udostępniał statystyki użycia i wstępną konfigurację przy użyciu poleceń interfejsu konsolowego, pozwalając jednak na funkcjonalne uruchomienie z domyślną konfiguracją.

3.2. Definicje, architektura i technologie

3.2.1. Słownik pojęć

W celu uniknięcia niejednoznaczności w dalszym toku pracy, zdefiniowane zostaną następujące pojęcia:

- **węzeł** – pojedynczy, autonomiczny element systemu, który wykorzystuje komunikację sieciową w celu rozgłaszania i pobierania danych;
- **klient** – węzeł, który pobiera dane w celach użytkowych, zaimplementowany w wersji dedykowanej użytkownikowi końcowemu;
- **klaster** – zbiór węzłów, który posiada informacje i ścieżki komunikacyjne pozwalające efektywnie na wymianę informacji pomiędzy każdym z nich;
- **metadane klastra** – ustrukturyzowane informacje wymieniane pomiędzy węzłami w celu ustalenia stanu i optymalizacji działania klastra;
- **metadane pliku** – ustrukturyzowane informacje wymieniane między węzłami w celu propagacji i umożliwienia wymiany faktycznych danych plikowych;

3.2.2. Koncept architektury

Z punktu widzenia klastra, koncepty węzła i klienta są jednoznaczne i równorzędne – wszystkie wyposażone są we wspólne mechanizmy komunikacji, i protokół wymiany danych między nimi działa na niezmienionej zasadzie. Wspólnie budować będą rozproszoną tablicę haszującą (DHT), która pozwoli na dostęp do metadanych klastra informujących o cechach poszczególnych węzłów oraz metadanych plików, przyspieszając proces ich identyfikacji i zapewniając bezpieczeństwo oraz spójność danych.

W celu zapewnienia płynności działania systemu, metadane klastra powinny zawierać w sobie informacje o *reputacji* poszczególnych węzłów. Podczas gdy węzły serwerowe z reguły zostają uruchomione i będą działać, mając szczyty obciążenia czy wymagań zależne głównie od obciążenia całej sieci (lub ewentualnego alternatywnego oprogramowania serwerowego na nich uruchamianego), węzły klienckie podlegać będą prawdopodobnie o wiele większej dynamice, z racji licznych odczytów dużych plików i podejścia użytkownika, które może zakładać częste otwieranie i zamykanie przeglądarki.

Węzły serwerowe i klienci powinny mieć zatem różne wartości bazowe przy ocenie ich sprawności w sieci – reputacja oprogramowania klienckiego powinna być domyślnie niższa, i szybciej reagować na zmiany jej wartości.

3.2.3. Wykorzystane technologie i narzędzia

Budowa kompletnego stosu technologicznego dla projektu o takich wymaganiach przez lata pozostawała problemem nietrywialnym. Paradygmaty programowania specjalizowane w podejściu obiektowym wspierały budowę monolitów, a komunikacja klient-serwer często polegała na tworzeniu dużej ilości poleceń, bez skupienia na wydajności takich rozwiązań.

Dużą zmianą w tej kwestii jest rozwój `libp2p`, stanowiący efekt długotrwałej pracy nad zrozumieniem stosu sieciowego Internetu, zbiór protokołów i wyprowadzonych zeń narzędzi, mechanik i interfejsów, pozwalających na ich podstawie budować własne, kompleksowe rozwiązania ([11]).

`libp2p` stanowi względnie wysokopoziomową kanwę dla projektu dHTTP. To właśnie na efektach pracy tego projektu opiera się warstwa tworzenia i komunikacji węzłów, budująca klaster dHTTP. Implementacja `js-libp2p` udostępnia moduły niezbędne w komunikacji sieciowej, łączeniu strumieni różnych protokołów, wykrywaniu nowych węzłów, propagowaniu informacji o stanie sieci czy wreszcie budowaniu rozproszonej tablicy haszującej, stanowiącej bazę metadanych klastra i plików.

Nie bez wpływu pozostaje rozwój przeglądarek internetowych. Współczesne browsery udostępniają kompleksowe API, pozwalające rozwijać wtyczki zmieniające zawartość stron internetowych, z uwzględnieniem kwestii wydajności i bezpieczeństwa. Istotne jest również wsparcie dla nowych technik komunikacji takich jak *WebSockets*, pozwalającej na wymianę informacji w czasie rzeczywistym, dzięki utrzymywaniu dwustronnie interaktywnej sesji TCP pomiędzy przeglądarką i serwerem. W ramach tej pracy rozwinięta została wtyczka dla przeglądarki Google Chrome. Wybór ten podyktowany został znaczną przewagą Chrome – w chwili pisania tej pracy, udział Chrome w rynku przeglądarek na komputerach osobistych wynosił ponad 64% ([12]). Prace i testy przeprowadzono przy użyciu 64-bitowego Google Chrome w wersji 63.0, na platformie macOS w wersji 10.13.2.

Za powstanie *headless* dHTTP w znacznej części odpowiada Node.js – środowisko uruchomieniowe pozwalające na uruchamianie kodu JavaScript po stronie serwerowej. Narzędzia, takie jak `npm` i Browserify, pozwalają z kolei na wykorzystywanie serwerowego kodu JavaScriptu w kodzie klienckim. Dzięki możliwości rozwoju obu aplikacji przy użyciu tego samego języka programowania i powyższym rozwiązaniom, znaczna część kodu aplikacji może być współdzielona. *Headless* dHTTP testowane było przy użyciu `node` w wersji v8.9.1, na platformie macOS oraz Amazon Linux AMI.

3.2.4. Paradygmaty i koncepcje

Istotą projektu dHTTP jest reagowanie na zapytania i komunikacja pomiędzy węzłami. Ponadto, projekt działać będzie w środowisku JavaScriptowym – język ten (poza web workers – rozwiązaniem polegającymi na delegacji obliczeń do wydzielonego środowiska, patrz: [13]) uruchamiany jest jednowątkowo. W podejściu proceduralnym może to powodować problemy związane z blokowaniem się wydarzeń; jest to dotkliwe zwłaszcza w przypadku interfejsów

użytkownika, które w takiej sytuacji tracą responsywność. Poniższe sekcje zawierają omówienie związanym z tych praktyk, stosowanych w projekcie dHTTP.

3.2.4.1. Asynchroniczność

Podstawowym rozwiązaniem stosowanym w języku JavaScript, pozwalającym na wykonanie w środowisku jednowątkowym, jest model współbieżności oparty o tzw *event loop*. Wywołania funkcji odkładane są na stosie, podczas gdy kolejne wiadomości są wkładane do kolejki FIFO. Podstawowym założeniem jest nigdy nie blokować – jeśli jakieś wywołanie wymaga konkretnej reakcji, definiowanej przez użytkownika, należy nasłuchiwać wydarzeń lub użyć wywołań zwrotnych (`callback`).

```
1  function foo(callback) {
2      var a = someWaitingOperation()
3      callback(a)
4  }
5
6  foo((a) => print(a))
7  console.log('Hello, world!')
```

W powyższym przykładzie warto zauważyć, że wywołanie `console.log(...)` będzie miało miejsce natychmiast po zawołaniu funkcji foo – jest więc bardzo prawdopodobnym, że efekt jego pracy widoczny będzie przed zawołaniem funkcji `callback`.

Asynchroniczne odwołania są kluczowe w przypadku wydarzeń takich jak oczekiwanie na odpowiedź serwera; czekając, program może wykonywać następne polecenia, a reakcja na odpowiedź zostanie wpleciona w tok pracy aplikacji.

3.2.4.2. Event-driven architecture

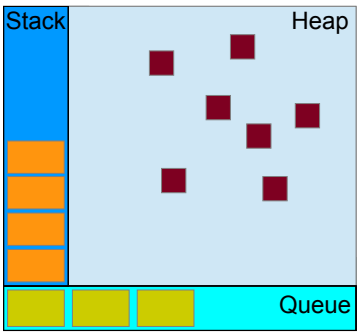
Pozioma oś modelu współbieżności JavaScriptu zapewnia natywne wsparcie dla reakcji na wydarzenia. W celu nasłuchiwania wydarzenia, należy zarejestrować jego obserwatora (*listener*) stosując z reguły składnię zbliżoną do:

```
1  a.addEventListener('click', ()=> {
2      console.log('Button clicked!')
3  })
```

W powyższym przykładzie, widać rejestrację reakcji na kliknięcie w przycisk. Jeżeli dana sytuacja nastąpi, wiadomość wysłana zostanie na kolejkę, co w momencie jej napotkania doprowadzi do wrzucenia na stos zarejestrowanej funkcji.

3.2.4.3. Strumienie

Node.js przeniósł środowisko JavaScript, którego dotychczasowym targetem były lekkie rozwiązania klienckie, na poziom serwerowy, w którym niektóre operacje są długotrwałe, mają przerwy w wywołaniach i nierzadko wymagają informowania o postępach (przykładem może



Rys. 3.2. Wizualizacja modelu współbieżności języka JavaScript – widać stos funkcji (oś pionowa) i kolejkę wydarzeń (oś pozioma). Pomiędzy nimi znajduje się współdzielona sterta, zapewniająca dostęp do danych. Źródło: [14]

być utrzymanie połączenia sieciowego, czy odczyt standardowego wyjścia). Choć jest możliwym implementacja tego typu rozwiązań wywołaniami zwrotnymi, czytelność i utrzymywalność takiego kodu drastycznie spada – przepływ logiki programu staje się trudny do wyśledzenia.

Z pomocą przychodzą strumienie ([15]). Strumienie nakładają abstrakcję, która pozwala intuicyjnie czytać (dla strumieni `Readable`) wrzucać (w przypadku `Writable`), a także transformować (`Transform`) przekazywane wartości.

```
1  // Przykład nawiązywania połączenia w metaprotokole dHTTP
2  dhttpClient.node.dial(peerInfo, '/dhttp/meta/0.1', (err, connStream) => {
3      // connStream to przykład strumienia Duplex - pozwala zarówno na zapis,
           jak i odczyt.
4      // zapis do connStream
5      connStream.write("Let's talk")
6      // jeśli w strumieniu pojawią się dane, nastąpi wywołanie funkcji
           callback. Warto zauważyć, że to wywołanie może nastąpić zarówno
           przed, jak i po zawołaniu write -- zależy od stanu strumienia, który
           może otrzymywać dane w innych miejscach programu
7      connStream.on('data', (data) => {
8          callback(JSON.parse(data))
9      })
10 }
11
12
13 // Przykład uruchamiania serwera HTTP w node.js dzięki 'http-server'
14 http.createServer((req, res) => {
15     // Przekazanie strumienia req strumieniowi request, który następnie
           zostanie przekazany do res. W efekcie zadziałamy jako najprostsze
           proxy - otrzymamy dane o oryginalnym zapytaniu, wywołamy je z punktu
           widzenia serwera, i przekażemy wynik w ramach odpowiedzi.
16     req.pipe(request(req.url)).pipe(res)
17 }).listen(34887)
```

Powyższy przykład, stanowiący część kodu dHTTP, pokazuje istotny koncept *potoków* (ang. *pipe*). Potoki pozwalają łączyć strumienie w sposób analogiczny do strumieni znanych z systemów Uniksowych: odczytane wartości można przekazać kolejnemu strumieniowi, który może zamienić je na inne wartości, aż w końcu wynik pracy zostanie zapisany w strumieniu odpowiedzi. Tego typu operacje stanowią kanwę przemian stosowanych w dHTTP.

Podkreślić należy również pozytywny wpływ strumieni na wydajność – ta abstrakcja pozwala operować na plikach przy użyciu buforów w sposób przezroczysty: przykładowo, aby wysłać duży obrazek nie przechowując go w całości w pamięci, wystarczy utworzyć strumień pliku i przepotokować go do strumienia wyjściowego.

```
1  var fileStream =  fs.createReadStream('files/caviar.jpg');
2  fileStream.pipe(connStream)
3  //kiedy strumień pliku się zakończy, nastąpi zamknięcie połączenia aby
       poinformować odbiorcę, że nic już na niego nie czeka
4  fileStream.on('finish', ()=> connStream.close())
```

3.3. Interfejs `dhttp.js`

Podstawę logiki systemu stanowi plik `dhttp.js`. Jest on interfejsem łączącym backend i stronę użytkową, pozwalającym na dostęp do informacji na temat sposobu działania węzła, a także wykonywanie operacji w sieci dHTTP.

`dhttp.js` operuje na dwóch protokołach:

- `/dhttp/meta/0.1` – *metaprotokół*, stanowi podstawę szkieletu systemu dHTTP. Informuje o dostępnych węzłach i plikach, obciążeniu sieci, a także następujących na bieżąco zmianach. Konwencją podjętą dla *metaprotokołu* jest powszechna propagacja danych – przykładowo, jeśli następuje połączenie z nowym węzłem, należy wysłać mu listę posiadanych plików, a jeśli planowane jest opuszczenie sieci, trzeba poinformować o tym fakcie podłączonych sąsiadów. Dzięki temu podejściu każdy węzeł posiada zbiór informacji na temat otoczenia, pozwalający na szybsze podejmowanie decyzji.

- `/dhttp/data/0.1` – *protokół danych*, stanowi system przesyłu docelowych plików w dHTTP. Zorientowany strumieniowo, przesyła zgromadzone pliki w ich oryginalnym formacie. Może zostać rozszerzony o obsługę zapytań cząstkowych – tak, aby duże pliki pobierać częściami od swoich sąsiadów.

Ponadto, `dhttp.js` wykorzystuje wbudowane *mechanizmy odkryć* `libp2p` w celu wykrywania sąsiadów i zmian sieci.

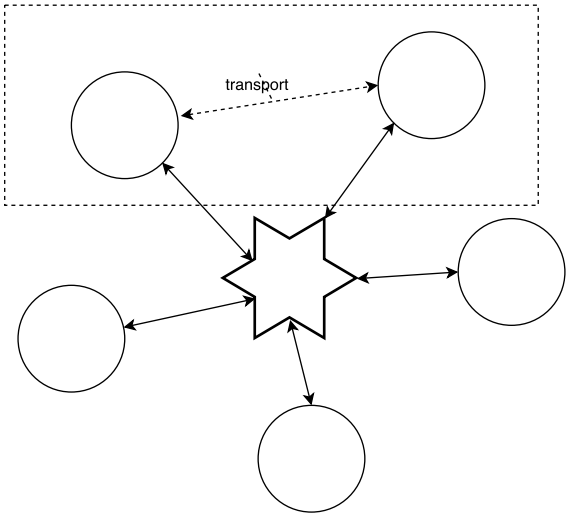
3.3.1. Warstwa sieciowa

Warto zaznaczyć, że rozwiązania opisywane w powyższej sekcji jako *protokół* są wysokopoziomową nakładką na warstwę sieciową. Jak zatem węzły nawiązują połączenie?

Połączenia w sieci peer-to-peer są znacznie utrudnione przez **Network Address Translation** (NAT). NAT jest techniką stosowaną przez routery, polegającą na zmianie adresów i portów IP w toku przesyłu danych w celu udostępnienia węzłom sieci prywatnych dostępu do internetu przy użyciu jednego adresu. Gdy użytkownik za „maskaradą" (jak bywa określany NAT) wykonuje zapytania, serwery widzą adres IP pierwszego routera, który posiada publiczny adres sieciowy. Odpowiedzi są możliwe dzięki informacjom przechowywanym przez routery prowadzące ten typ komunikacji.

Jest to konieczność w obliczu wyczerpanej puli adresów IPv4, prowadzi jednak do sytuacji, w której otwarcie bezpośredniego łącza pomiędzy dwoma użytkownikami końcowymi jest z reguły niemożliwe – nie ma możliwości odpytania adresu IP końcowego komputera.

W celu poradzenia sobie z tym problemem, dHTTP wykorzystuje bibliotekę `js-libp2p-webrtc-star`. To rozwiązanie działa na zasadzie serwera spotkań (*Rendezvous Server*), który stanowi swoisty przekaźnik i pozwala na rejestrację węzłów. Dzięki temu, jest jednocześnie odpowiedzialny za warstwę transportu jak i *mechanizm odkryć*.



Rys. 3.3. Wizualizacja sieci opartej o `js-libp2p-webrtc-star`, z przykładem wirtualnego połączenia transportowego pomiędzy dwoma węzłami.

Niestety, stawia to pod znakiem zapytania rozproszenie systemu – jak widać na rys. 3.3, obecnie system przypomina bardziej scentralizowaną topologię gwiazdy niż system faktycznie rozproszony. Obciąża też *Rendezvous Server* koniecznością przekazywania treści. Jest to jednak rozwiązanie przejściowe i na chwilę obecną konieczne – `js-libp2p` w czasie tworzenia tego projektu nie udostępnia innych metod trawersowania NAT-u.

Warto dodać, że w przypadku sieci opartej wyłącznie o węzły w sieci lokalnej możliwe jest stosowanie tzw. Multicast DNS, który nie potrzebuje żadnego punktu centralnego; w przypadku serwerów posiadających publiczne adresy IP, możliwe staje się użycie list adresów utrzymywanych przez serwery śledzące – trackery – oba rozwiązania pozwalają na bezpośredni transport pomiędzy węzłami i nie wymagają zmiany logiki systemu ([16]).

3.3.2. Struktura metadanych

Metadane przesyłane przy użyciu `/dhttp/meta/0.1` to podstawa przepływu danych w systemie. Wykorzystywane do budowania wspólnej bazy wiedzy na temat sieci, składać się muszą z lekkich struktur i jasnych komunikatów.

Podstawowe zapytanie `/dhttp/meta/0.1` wygląda następująco:

```
1  {
2      type: "echo",
3      peerId: dhttpClient.node.peerInfo.id.toB58String(),
4      text: "Hello, node."
5  }
```

Omówienie zapytań i odpowiedzi jest zawarte w rozdziale 3.4.

3.4. API

W tej sekcji szczegółowo opisane zostały udostępnione przez `dhttp.js` funkcje i mechanizmy związane ze strukturami i przepływami danych pomiędzy węzłami. Ogólny przepływ informacji systemu zawiera się w do podsekcji 3.5.2, opisującej przepływy danych, oraz podsekcji 3.5.3, opisującej cykl życia węzła.

Podstawowym założeniem jest istnienie singletona `dhttpClient`, który przy rozruchu aplikacji inicjalizuje węzeł sieci. Wszelkie odwołania do działania węzła powinny polegać na rejestrowaniu wydarzeń i wywoływaniu funkcji globalnego obiektu `dhttpClient`.

3.4.1. Struktury danych

W celu poprawy wydajności i uproszczenia przepływów w aplikacji, dHTTP oferuje kilka struktur informujących o dostępności danych.

Typy struktur jak i funkcje oferowane przez `js-libp2p`, do których odnosi się poniższe API, opisane są na stronie projektu: [17].

3.4.1.1. Struktury podstawowe

– **`PeerId`** – `String` zawierający ID węzła sieci, uzyskane dzięki zawołaniu `PeerInfo.id.toB58String()`

– **`RequestURL`** – `String` zawierający ścieżkę zapytania klienckiego. Jeśli klient odpyta nasz serwer na adresie `http://localhost:34887/http://example.com/script.js`, instancja `RequestURL` zawierać będzie `http://example.com/script.js`.

– **`popularityIndex`** – `Integer` określający popularność danego pliku z perspektywy posiadającego go węzła.

– **Data** – serializowalny obiekt zawierający dane przesyłane w ramach metaprotokołu, zgodne z formatami zdefiniowanymi w 3.4.4.

3.4.1.2. filesInNetwork

```
filesInNetwork: {
        "http://example.com/file.jpg": { // RequestURL
            nodes: {
                PeerId: {
                    popularityIndex: Int,
                }
            }
            popularityIndex: Int
        },
        "http://example.com/script.js" : {...}
    }
```

`filesInNetwork` jest strukturą odpytywaną przy połączeniu klienckim. Stanowi pierwszy punkt wejścia do sieci, i zoptymalizowany jest pod szybkie przeszukanie i weryfikację, czy plik jest dostępny przy użyciu dHTTP.

Poszczególne elementy indeksowane są poprzez adresy URL zasobów pobieranych przez klienta (`ReqestURL`).

Pola:

– **nodes** – mapa zawierająca informacje tych klientów, którzy posiadają dostęp do danego pliku.

– **popularityIndex** – liczba całkowita określająca popularność pliku z perspektywy tego węzła. Może być wymieniana z innymi węzłami sieci w celu optymalizacji obciążenia – ułatwi węzłowi odpytywanemu decyzję na temat przekazania połączenia do najmniej zajętego sąsiada.

3.4.1.3. swarm

```
swarm: {
    nodes: {
        QmVxxoRFLR8VjbqeVer9Z9DqCJoJT36c9Uomd9AtP8NBx6: // PeerId
            PeerInfo {...}
    }
}
```

`swarm` jest strukturą utrzymującą informacje na temat wszystkich węzłów, które są bezpośrednio podłączone do obecnego. Reprezentuje więc swoisty *lokalny klaster*, z całością którego można nawiązywać połączenia.

Pola:

– **nodes** – obiekt, w którym poszczególne elementy reprezentują `PeerInfo` – kompleksowy zbiór informacji z API `js-libp2p`, pozwalający na nawiązywanie połączeń.

3.4.1.4. `stats`

```
stats: {
    downloaded: {bytes: 0, files: 0},
    fetched: {bytes: 0, files: 0},
    uploaded: {bytes: 0, files: 0},
}
```

`stats` gromadzi statystyki zużycia sieci przez obecny węzeł.

Pola:

- **`downloaded`** – ile danych pobrano z perspektywy klienta.
- **`fetched`** – ile danych pobrano w tle – podczas operacji niezwiązanych bezpośrednio z działania użytkownika, lecz podjętych w celach optymalizacyjnych.
- **`uploaded`** – ile danych wysłano do innych węzłów podczas działań optymalizacyjnych i propagacji danych w systemie.

3.4.1.5. `node`

```
    node: Node {protocols: {...}, peerInfo: PeerInfo, ...}
```

`node` jest obiektem pozwalającym na bezpośredni dostęp do reprezentacji danego węzła w warstwie `js-libp2p`. Pozwala na wykonywanie połączeń i propagację danych z poziomu innych funkcji dHTTP. Szczegóły zawarte w dokumentacji projektu ([17]).

3.4.1.6. `storage`

```
storage: {
    quotas: {
        RAM: {
            files: 0, megabytes: 0
        },
        drive: {
            files: 0, megabytes: 0
        }
    },
    RAMFiles: {
        files: {
            "http://example.com/logo.png": { // RequestURL
                value: Blob
            },
            "http://example.com/logoLocalStorage.png": { ... },
            ...
        }
    },
    driveFiles: {
        files: {
            "http://example.com/onDrive.png": { // RequestURL
                path: "/files/http://example.com/onDrive.png"
```

```
            },
            "http://example.com/onDrive2.png": { ... },
            ...
        }
    }
}
```

`storage` nakłada warstwę abstrakcji na obsługę plików utrzymywanych przez węzeł, która automatyzuje proces wyboru rodzaju pamięci dla zapisu oraz odczytu danych.

Pola:

– **`quotas`** – obiekt który zawiera w sobie konfigurację ograniczeń pamięci. Dla wartości `0` nie istnieją żadne ograniczenia; wartość `-1` oznacza zakaz korzystania z danego rodzaju pamięci.

– **`RAMFiles`** – zbiór wszystkich plików przechowywanych w pamięci RAM.

  `value` – plik trzymany w formacie binarnym, gotowy do strumieniowego przesłania użytkownikowi końcowemu.

– **`RAMFiles`** – zbiór wszystkich plików przechowywanych w systemie plików użytkownika.

  `value` – względna ścieżka do pliku, który zawiera dany zasób.

3.4.2. Funkcje

Dzięki szeregowi udostępnionych funkcji, dHTTP umożliwia wysokopoziomowe, asynchroniczne podejście do informacji o systemie i zawartości plików. Poniżej zostały zawarte opisy najważniejszych z nich.

3.4.2.1. `filesAvailable/0`

```
    filesAvailable: function() -> [RequestURL]
```

`filesAvailable` jest funkcją stanowiącą wykaz wszystkich plików udostępnionych przez dany węzeł. Nie stosuje rozróżnienia na dane dostępne w RAMie czy na dysku – ta optymalizacja pozostaje na poziomie poszczególnych węzłów, aby nie obciążać wszystkich klientów sieci.

Wartość zwracana: Lista `RequestURL` obsługiwanych przez dany węzeł.

3.4.2.2. `dialNode/3`

```
    dialNode: function(PeerInfo, Data, (ResponseData) -> ()) -> ()
```

`dialNode` pozwala na wykonanie połączenia do istniejącego węzła w celu przesłania mu metadanych projektu. W ramach danych, wysłane mogą zostać serializowane obiekty komunikatów, opisane szczegółowo w sekcji 3.4.4.

Parametry:

– **`PeerInfo`** – obiekt `PeerInfo` węzła do którego ma zostać wysłana wiadomość

– **Data** – serializowalny obiekt zawierający wysyłane dane; `ResponseData` jest tym samym typem, wydzielonym w celu większej czytelności – zaznaczenia, że reprezentuje dane otrzymane.

– **(ResponseData) -> ()** – callback, wołany dla odpowiedzi otrzymywanych od serwera. Otrzymuje jako parametr zdeserializowany obiekt odpowiedzi.

3.4.2.3. addFile/3

```
1    addFile: function(RequestURL, popularityIndex, (ResponseData) -> ()) ->
         ()
```

`addFile` pozwala poinformować sieć o pojawieniu się nowego pliku i o tym, że jest on obsługiwany przez nas, z danym indeksem popularności. Sieć może wykorzystać tę wiedzę aby zacząć przekierowywać użytkowników na dany plik, lub pobierać go w celach udostępniania.

Parametry:

– **RequestURL** – ścieżka pliku

– **popularityIndex** – patrz: `filesInNetwork`

– **(ResponseData) -> ()** – callback, wołany dla odpowiedzi otrzymywanych od serwera; pozwala reagować w przypadkach wystąpienia nieoczekiwanych błędów.

3.4.2.4. removeFile/2

```
1    removeFile: function(RequestURL, (ResponseData) -> ()) -> ()
```

`removeFile` pozwala poinformować sieć o decyzji zakończenia udostępnienia danego pliku. Może mieć związek z decyzjami optymalizacyjnymi – jeżeli danemu węzłowi zaczyna brakować miejsca, może podjąć decyzję o usuwaniu najmniej popularnych plików.

Parametry:

– **RequestURL** – ścieżka pliku

– **(ResponseData) -> ()** – callback, wołany dla odpowiedzi otrzymywanych od serwera; pozwala reagować w przypadkach wystąpienia nieoczekiwanych błędów.

3.4.2.5. fetchFromNode/3

```
1    fetchFromNode: function(RequestURL, Writable, PeerInfo) -> ()
```

`fetchFromNode` pozwala pobierać pliki. Jest dedykowane pobraniom wewnętrznym, optymalizacyjnym – powinno służyć węzłowi do aktualizacji własnej bazy plików. Funkcja wysyła `end` do strumienia pytającego w momencie zakończenia przesyłu.

Parametry:

– **RequestURL** – ścieżka pliku

– **Writable** – strumień, do którego ma zostać wysłany poszukiwany plik.

– **PeerInfo** – obiekt reprezentujący węzeł, z którego pobieramy plik.

3.4.2.6. `propagate/2`

```
1    propagate: function(Data, (Data) -> ()) -> ()
```

`propagate` wysyła dane przy użyciu metaprotokołu do wszystkich dostępnych węzłów. **??** zawiera szczegóły obsługiwanych komunikatów.

Parametry:

– **`Data`** – przesyłane, serializowalne dane.

– **`(ResponseData) -> ()`** – callback, wołany dla odpowiedzi otrzymywanych od serwera; pozwala reagować w przypadkach wystąpienia nieoczekiwanych błędów.

3.4.3. Wydarzenia

Ta sekcja również zawiera funkcje, spełniają one jednak specyficzną funkcję – są częścią *reaktywną* systemu, wołane są poprzez odwołania w czasie połączeń między węzłami i stanowią definicję reakcji na sytuacje i zapytania występujące w systemie.

3.4.3.1. `download/3`

```
1    download: function(RequestURL, Writable, Fallback () -> ()) -> ()
```

`download` jest funkcją pozwalającą na pobranie pliku z sieci, z uwzględnieniem źródeł danych i sąsiadów węzła. Szczegóły cyklu pobierania danych omówione są w sekcji 3.5.4.

Parametry:

– **`RequestURL`** – adres żądanego pliku.

– **`Writable`** – strumień do którego powinien zostać zapisany plik. Otrzymuje `end` w momencie zakończenia pliku.

– **`Fallback`** – funkcja która może zostać zawołana w przypadku nieznalezienia pliku w zasięgu węzła – przykładowo, możliwe jest dzięki niej wysłanie pliku pobranego bezpośrednio z oryginalnego serwera.

3.4.3.2. `acceptIntroduction/2`

```
1    acceptIntroduction: function(ResponseData) -> ()
```

`acceptIntroduction` służy obsłudze sytuacji, gdy otrzymamy informację o pojawieniu się węzła sieci – propaguje on `cluster` i `filesInNetwork` szczegółami na temat nowego sąsiada i oferowanych przez niego plików.

Parametry:

– **`ResponseData`** – przesyłane przez węzeł, serializowalne metadane typu `introduction`.

3.4.3.3. attachNode/2

```
1    attachNode: function (PeerInfo) -> ()
```

`attachNode` służy dodaniu połączenia do węzła, i wywołaniu `introduceSelf/1` – odesłaniu nowemu sąsiadowi szczegółów na temat samego siebie w postaci metadanych typu `introduction`.

Parametry:

– **`PeerInfo`** – obiekt pozwalający na nawiązywanie połączeń.

3.4.3.4. detachNode/2

```
1    detachNode: function (PeerInfo) -> ()
```

`detachNode` pozwala usunąć obiekt z mapy `swarm` oraz informacje o istniejących plikach, efektywnie wypinając dany węzeł z własnej sieci.

Parametry:

– **`PeerInfo`** – obiekt reprezentujący usuwany węzeł.

3.4.4. Komunikaty metaprotokołu

3.4.5. Połączenia

3.5. Węzeł dHTTP

3.5.1. Aplikacja

3.5.2. Przepływy

3.5.3. Cykl życia węzła

3.5.4. Algorytm transportu danych

3.5.5. Przechowywanie danych

3.5.6. Komunikacja z klientem

3.6. Klient dHTTP

3.6.1. Aplikacja

3.6.2. Połączenie z węzłem

3.6.3. Eksperymentalny pełny klient

4. Testy, rozszerzenia i optymalizacja

4.1. Zarys użytkowości i działania systemu

4.2. Wydajność systemu

4.2.1. Wydajność klienta

4.2.2. Wydajność serwera

4.2.2.1. Wydajność rozwiązań propagacji danych

4.2.2.2. Wydajność protokołów komunikacji między węzłami

4.3. Bezpieczeństwo

4.3.1. Proponowane rozwiązania

4.4. Podsumowanie rozwiązań optymalizacyjnych

5. Podsumowanie

5.1. Potencjalne kierunki rozwoju

Bibliografia

[1] Amazon Web Services Inc. *Auto Scaling*. 2017. URL: *http://web.archive.org/web/20171127225615/https://aws.amazon.com/autoscaling/* (term. wiz. 2017-12-31).

[2] Gordon E Moore. „Cramming more components onto integrated circuits". W: *Proceedings of the IEEE* 86.1 (1998).

[3] Miniwatts Marketing Group. *World Internet Users and 2017 Population Stats*. 2017. URL: *http://web.archive.org/web/20171226094500/http://www.internetworldstats.com/stats.htm* (term. wiz. 2017-12-31).

[4] Juan Benet. „IPFS - Content Addressed, Versioned, P2P File System". W: *CoRR* (2014).

[5] Rob Pike. „Concurrency is not Parallelism". W: 2012.

[6] Sebastian Wyngaard Simon Perkins James Gain Michelle Kuttel i Jason Brownbridge. „Parallel Algorithms". W: 2011.

[7] Karthik Ram. „Git can facilitate greater reproducibility and increased transparency in science". W: *Source code for biology and medicine* 8.1 (2013), s. 7.

[8] Stefan Saroiu, Krishna P Gummadi i Steven D Gribble. „Measuring and analyzing the characteristics of Napster and Gnutella hosts". W: *Multimedia systems* 9.2 (2003), s. 170–184.

[9] Oskar Sandberg. „Searching in a Small World". Prac. dokt. Göteborg University, 2005.

[10] Google Inc. *IPv6 Statistics*. 2017. URL: *http://web.archive.org/web/20171220144243/https://www.google.com/intl/en/ipv6/statistics.html* (term. wiz. 2017-12-31).

[11] Protocol Labs Inc. *The spec for libp2p and associated submodules.* URL: *https://github.com/libp2p/specs* (term. wiz. 2017-12-31).

[12] StatCounter. *Desktop Browser Market Share Worldwide*. URL: *http://web.archive.org/web/20171226043756/http://gs.statcounter.com/browser-market-share/desktop/worldwide* (term. wiz. 2017-12-31).

[13] Mozilla. *Web Workers API*. URL: *http://web.archive.org/web/20180105105400/https://developer.mozilla.org/pl/docs/Web/API/Web_Workers_API* (term. wiz. 2018-01-05).

[14] Mozilla. *Model współbieżności i Event Loop*. URL: *http://web.archive.org/web/20170913075430/https://developer.mozilla.org/pl/docs/Web/JavaScript/EventLoop* (term. wiz. 2018-01-05).

[15] Node.js. *Node.js v9.3.0 Documentation. Stream*. URL: *https://nodejs.org/api/stream.html* (term. wiz. 2018-01-05).

[16] Protocol Labs Inc. *Peer Discovery Mechanisms*. URL: *https://github.com/libp2p/js-libp2p/tree/master/examples/discovery-mechanisms* (term. wiz. 2017-12-31).

[17] Protocol Labs Inc. *The JavaScript implementation of the libp2p Networking Stack. README.md*. URL: *https://github.com/libp2p/js-libp2p/blob/f8b441fff5bb5f5403c6e0af32511ed5f0bf2e80/README.md* (term. wiz. 2018-01-05).