



---

**POZNAN UNIVERSITY OF TECHNOLOGY**

---

**Filip Waligórski**

# Rozgłaszanie danych w grafach dużej skali

Praca magisterska

Promotor: dr Anna Kobusińska

Poznań, 2017



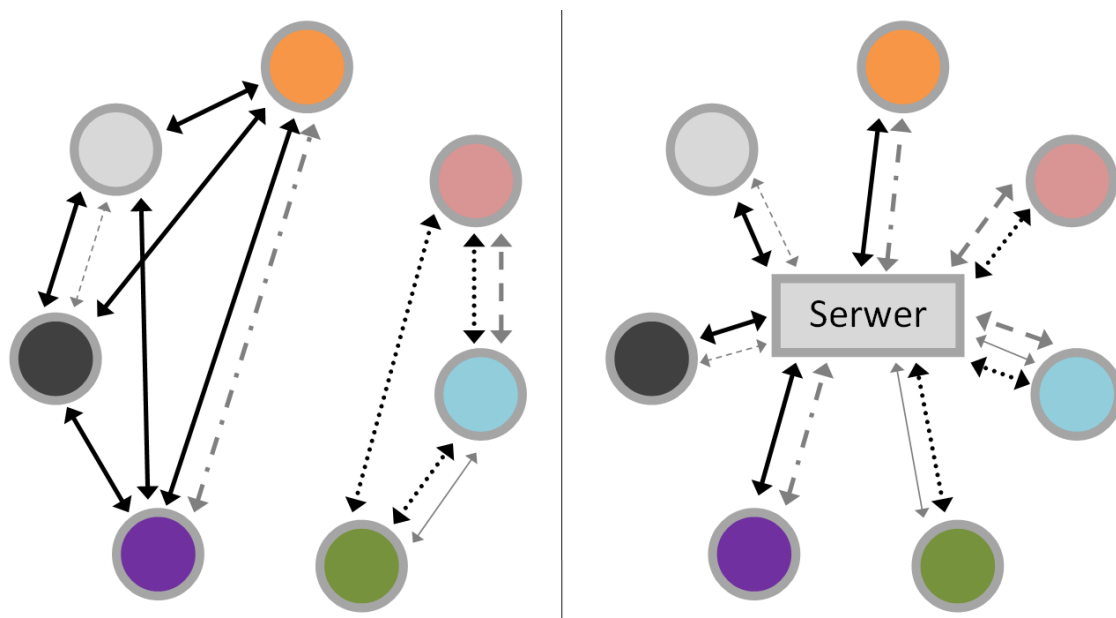
# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>3</b>
<b>2</b>	<b>Istniejące rozwiązania</b>	<b>5</b>
2.1	Facebook Messenger [1]	5
2.2	Bleep [3]	5
2.3	Signal [4]	6
2.4	WhatsApp [5]	6
2.5	Darkwire [6]	6
2.6	Friends [7]	7
2.7	Tox [9]	7
2.8	ZeroChat, ZeroMail, BitMessage [10][11]	8
<b>3</b>	<b>Koncepcja</b>	<b>9</b>
3.1	BitTorrent [13]	9
3.1.1	Podstawowe pojęcia	9
3.1.2	Scenariusz pobrania torrenta	10
3.1.3	Scenariusz utworzenia torrenta	10
3.1.4	Jak komunikują się peery	11
3.1.5	Choking algorithm	11
3.1.6	Serwery potrzebne do działania protokołu	11
3.1.7	Zalety i wady protokołu	12
3.2	WebTorrent [14]	12
<b>4</b>	<b>Architektura</b>	<b>14</b>
4.1	Serwer REST i Baza Danych	15
4.1.1	Użytkownicy i logowanie	15
4.1.2	Konwersacje	16
4.2	Klient i biblioteka WebTorrent	16
4.2.1	Logowanie	17
4.2.2	Inicjalizacja modułów	17
4.2.3	Utworzenie lub dołączenie do konwersacji	18
4.2.4	Struktura komunikatu	19
4.2.5	Wysyłanie wiadomości i algorytm ZLT	20

4.2.6	Odebranie wiadomości . . . . .	21
4.2.7	Udostępnianie starych torrentów . . . . .	22
4.2.8	Zbiór sąsiadów . . . . .	22
4.3	DHT . . . . .	23
4.4	Bezpieczeństwo i kryptografia w języku JavaScript . . . . .	24
4.4.1	Darkwire . . . . .	24
4.4.2	Bleep i Signal . . . . .	26
4.4.3	Inne pomysły i wnioski . . . . .	26
4.5	Dalszy rozwój . . . . .	27
<b>5</b>	<b>Wyniki testów</b>	<b>29</b>
5.1	Kryteria . . . . .	29
5.2	Scenariusze testowe . . . . .	29
5.3	Specyfikacja komputerów i środowiska testowe . . . . .	30
5.4	Wysyłanie pojedynczej wiadomości . . . . .	31
5.5	Wpływ rozmiaru wiadomości . . . . .	32
5.6	Wysyłanie i oczekiwanie na odpowiedź . . . . .	32
5.7	Wpływ algorytmu ZLT na wydajność . . . . .	34
5.8	Wpływ liczby wiadomości w algorytmie ZLT . . . . .	36
5.9	Pobieranie listy wiadomości . . . . .	39
5.10	Wpływ jednoczesnych konwersacji na wydajność . . . . .	39
<b>6</b>	<b>Wnioski</b>	<b>41</b>
	<b>Bibliografia</b>	<b>42</b>

# Wstęp

Komunikatory zrewolucjonizowały sposób, w jaki ludzie wymieniają się informacjami. Pozwalają na niemalże natychmiastowe wysłanie dowolnej treści (tekstu, zdjęcia, filmu) w formie wiadomości do grona odbiorców — najczęściej znajomych. Zbiór wszystkich użytkowników korzystających z danego komunikatora tworzy graf — za wierzchołki można uznać włączone aplikacje, a krawędziami są wysyłane i odbierane komunikaty. Przyjęcie takiej perspektywy pozwala na rozpatrzenie problemu dystrybucji wiadomości jako tytułowe rozgłaszanie danych w grafach dużej, wręcz globalnej, skali. W rozdziale 2 przedstawiono kilka istniejących komunikatorów. Wśród nich można wyróżnić dwie główne architektury — rozproszoną (komunikacja bezpośrednia P2P<sup>1</sup>) oraz scentralizowaną (z serwerem pośredniczącym). Architektury te, w formie wspomnianego grafu, zostały zaprezentowane na rysunku 1.1. Strzałki tego samego typu symbolizują rozmowy pomiędzy konkretnymi użytkownikami lub konwersacje grupowe.



**Rys. 1.1:** Architektura rozproszona (po lewej) i scentralizowana (po prawej)

Pod pojęciem „serwer” na rysunku kryje się zwykle jedno lub wiele centrów przetwarzania

<sup>1</sup>peer to peer

nia danych, których utrzymanie jest niezwykle kosztowne ze względu na duże obciążenie systemu. Często praktyką firm jest gromadzenie i analizowanie danych o użytkownikach, aby na tej podstawie generować spersonalizowane reklamy i osiągać zyski. Dzięki temu komunikator jest darmowy dla użytkowników, choć „płacą” oni za korzystanie swoją prywatnością. Wyeliminowanie kosztów (serwerów pośredniczących) pozwoliłoby na zrezygnowanie z tej praktyki. W tym celu możliwe jest zastosowanie architektury rozproszonej.

BitTorrent jest jednym z najpopularniejszych protokołów służących do dystrybucji plików w sieci P2P. Automatycznie dzieli udostępniane pliki na części i umożliwia ich pobieranie z wielu źródeł, na przykład od innych użytkowników. Pozwala to na równomierne rozłożenie obciążenia pomiędzy komputery biorące udział w udostępnianiu plików oraz umożliwia dalsze pobieranie nawet gdy pierwotny nadawca pliku jest niedostępny. W niniejszej pracy protokół został użyty do przekazywania wiadomości.

WebTorrent to biblioteka implementująca protokół BitTorrent. Została napisana w języku JavaScript, dzięki czemu możliwe jest użycie jej w aplikacji webowej — pozwala na wymianę plików pomiędzy przeglądarkami.

Celem pracy jest zaprojektowanie i implementacja systemu komunikatora grupowego w formie aplikacji webowej. Jednocześnie, założeniem jest jak największe wykorzystanie modelu komunikacji P2P w projekcie, aby minimalizować rolę serwerów w systemie. Dzięki temu komunikator powinien generować jak najmniejsze koszty. Do obsługi bezpośredniej komunikacji pomiędzy klientami, a konkretnie do przesyłania wiadomości, wykorzystano wspomnianą bibliotekę WebTorrent. W dalszej kolejności zbadano możliwości poprawy bezpieczeństwa przetwarzanych danych oraz zwiększenia prywatności użytkowników poprzez wprowadzenie szyfrowania wiadomości.

W kolejnych rozdziałach podsumowano wyniki pracy wykonanej w celu wdrożenia opisanego systemu. Rozdział 2 opisuje istniejące obecnie na rynku komunikatory. W rozdziale 3 zawarto teoretyczne podstawy działania protokołu BitTorrent, niezbędne do opracowania koncepcji jego wykorzystania w kontekście komunikatora. Rozdział 4 zawiera informacje dotyczące projektu systemu — opis architektury i sposób działania poszczególnych modułów. Znalazły się w nim również wnioski z badań dotyczących szyfrowania wiadomości. Na koniec przeprowadzono i udokumentowano testy wydajnościowe aplikacji — wyniki znajdują się w rozdziale 5.

# Istniejące rozwiązania

W tym rozdziale zaprezentowano istniejące komunikatory dla dwóch osób oraz komunikatory grupowe. Ich cele i funkcjonalność są zbliżone choć realizują je z różnymi założeniami oraz bazując na różnych architekturach i koncepcjach. Poniżej opisane zostały wybrane rozwiązania z naciskiem na cechy wyróżniające je spośród konkurencyjnych aplikacji.

## 2.1 Facebook Messenger [1]

Jest to jeden z najpopularniejszych obecnie komunikatorów. Oferuje zarówno rozmowy dla 2 osób jak i grupowe. Wspiera wysyłanie wszelkich multimediów i plików oraz dostarczanie wiadomości pod nieobecność nadawcy. Dostępny jest na najszerszej gamie platform — jako aplikacja webowa, mobilna oraz desktopowa, czym wyróżnia się na tle konkurencji. Architektonicznie Messenger polega na „centralnym serwerze” przekazującym wiadomości. Cudysłów wynika z faktu, że pod pojęciem „serwer” kryje się ogromna infrastruktura złożona z wielu maszyn, którą firma musi utrzymywać. Wadą tego komunikatora jest brak domyślnego wsparcia szyfrowania wiadomości — opcję można włączyć tylko w aplikacji mobilnej dla poszczególnych konwersacji, jednak nie każdy użytkownik jest tej opcji świadomy. Szyfrowanie w aplikacji webowej nie jest dostępne [2]. Kod źródłowy aplikacji nie został udostępniony (nie jest to open-source), co oznacza, że za wprowadzanie zmian i dodawanie nowych funkcji odpowiedzialny jest tylko wydawca — społeczność użytkowników nie ma możliwości ingerowania w funkcjonalność. Jest to wada w sytuacji, gdy nowa wersja oprogramowania zawiera niechciane przez użytkowników dodatki (zbędne lub na przykład szpiegujące), a nie zawiera takich, które są przez nich wyczekiwane.

## 2.2 Bleep [3]

Bleep jest komunikatorem zaprojektowanym przez firmę rozwijającą protokół BitTorrent. Do dyspozycji użytkowników oddano aplikację mobilną oraz aplikację desktopową (brak aplikacji webowej). Podobnie jak w przypadku Messengera z poprzedniego punktu, nie jest to oprogramowanie open-source. Bleep oferuje rozmowy dla dwóch osób, a w planach twórców jest zaimplementowanie komunikacji grupowej. Wiadomości są szyfrowane przed wysłaniem na urządzeniu nadawcy i odszyfrowywane na urządzeniu odbiorcy — szyfrowanie end to end.

Jednak najważniejszą cechą wyróżniającą ten komunikator jest jego architektura - brak centralnego serwera pośredniczącego w przekazywaniu wiadomości. Komunikaty przesyłane są bezpośrednio między urządzeniami, jeśli oba są dostępne w momencie wysyłania, a w przeciwnym przypadku wiadomość umieszczana jest w DHT (Distributed Hash Table) i przechowywana do czasu odebrania jej. Specjalny mechanizm dba o to, by wiadomość nie zniknęła z DHT wcześniej. Dane o koncie użytkownika oraz klucze szyfrujące pozostają lokalnie na urządzeniu.

## 2.3 Signal [4]

Twórcy aplikacji Signal skupili się przede wszystkim na bezpieczeństwie i prywatności użytkowników. Wiadomości są szyfrowane na urządzeniach, więc pomimo faktu, że architektura zakłada obecność centralnego serwera, wiadomości przechowywane na nim nie mogą zostać odczytane przez osoby trzecie. Jednakże, ten typ architektury posiada znaczącą wadę — jeśli serwer stanie się niedostępny (na przykład z powodu jego awarii lub braku łączności z nim — awarii łącza komunikacyjnego) to niemożliwe jest prowadzenie rozmów (dotyczy to każdej aplikacji z centralnym serwerem). Kod źródłowy jest dostępny publicznie co oznacza, że każdy może sprawdzić zgodność implementacji z oferowanymi założeniami oraz uczestniczyć w rozwoju aplikacji. Podobnie jak w przypadku aplikacji Bleep, dostępne są natywne aplikacje mobilna i desktopowa. Możliwe jest prowadzenie rozmowy grupowej pomimo zastosowania szyfrowania wiadomości — treść zostaje zaszyfrowana symetrycznie (jedna wersja dla wszystkich odbiorców niezależnie od ich liczby), a następnie sam klucz jest szyfrowany zgodnie z oczekiwaniami każdego z odbiorców z osobna. Dzięki temu mechanizmowi uniknięto sytuacji, w której nadawca musiałby przygotować  $n$  wersji całej, potencjalnie dużej wiadomości dla  $n$  odbiorców.

Podobne rozwiązania: Wire, Telegram, Allo

## 2.4 WhatsApp [5]

WhatsApp jest aplikacją podobną do wspomnianego w poprzednim punkcie programu Signal, ale jej kod źródłowy nie jest dostępny publicznie. Inną, ważną cechą wyróżniającą WhatsApp jest oferowanie użytkownikom aplikacji webowej. Nie jest to jednak typowy, webowy klient sieci a jedynie webowy interfejs do aplikacji zainstalowanej na telefonie. Wysłana w kliencie webowym wiadomość najpierw zostaje przesłana bezpośrednio do aplikacji w telefonie i dopiero wówczas przekazana dalej do właściwych odbiorców. Oznacza to, że telefon staje się pośrednikiem w wysyłaniu i odbieraniu wiadomości pomiędzy klientem webowym i resztą systemu.

## 2.5 Darkwire [6]

Darkwire to aplikacja open-source oferująca komunikator grupowy z dostępem poprzez stronę internetową (aplikacja webowa). W przeciwieństwie do większości rozwiązań użytkownik nie musi tworzyć konta by skorzystać z programu. W celu skomunikowania się



z użytkownikami należy wymienić między nimi identyfikator konwersacji (link do konkretnego pokoju) korzystając z innego sposobu komunikacji (np. poprzez e-mail, SMS czy osobiście). Takie rozwiązanie zakłada, że identyfikator nie zostanie odgadnięty przez osoby trzecie — w przeciwnym przypadku będą one mogły odczytać wysyłane wiadomości. Architektura zakłada istnienie centralnego serwera uczestniczącego w przekazywaniu wiadomości. Z racji faktu, że aplikacja ma otwarte źródła, każdy może uruchomić swój własny serwer. Komunikaty są szyfrowane na urządzeniu (w przeglądarce) przed wysłaniem, zatem serwer nie zna treści wiadomości. Centralny serwer przesyła wiadomości tylko do tych uczestników, którzy są dostępni w momencie nadania wiadomości (brak wsparcia dla odbierania starszych wiadomości czy wysyłania wiadomości do użytkowników niedostępnych w danej chwili).

## 2.6 Friends [7]

Ten niszowy projekt open-source oferuje aplikację desktopową i umożliwia prowadzenie rozmów grupowych. Szyfrowanie wiadomości nie zostało do tej pory zrealizowane, ale jest jednym z punktów przyszłego rozwoju. Głównym celem twórców było stworzenie programu niezależnego od centralnego serwera oraz umożliwiającego rozmowę przy użyciu alternatywnych kanałów komunikacyjnych (np. poprzez Bluetooth) w sytuacji gdy połączenie internetowe jest niedostępne. Aplikacja wykorzystuje algorytm plotkowania (gossiping) oraz replikuje wiadomości przy użyciu drzewa skrótnów (hash tree, Merkle DAG, DAG - Directed Acyclic Graph [8]). Dzięki temu wiadomości w konwersacji mogą zostać połączone nawet w przypadku, gdy ktoś nadał komunikaty będąc odłączonym od sieci — mechanizm podobny do łączenia zmian w repozytorium kodu. Wykorzystanie opisanych powyżej mechanizmów gwarantuje ostateczną spójność otrzymanych wiadomości — każdy uczestnik rozmowy ostatecznie będzie widział taki sam zbiór wiadomości — przykładowy scenariusz dla 3 użytkowników:

1. Wiadomość wysłana przez użytkownika A została odebrana przez użytkownika B, który dołączył ją do swojego drzewa wiadomości.
2. Użytkownik A stał się niedostępny.
3. Użytkownik C stał się dostępny i odebrał od użytkownika B zmienioną wersję drzewa i w ten sposób dowiedział się o wiadomości wysłanej przez użytkownika A pomimo faktu, że ten jest w tej chwili niedostępny.

## 2.7 Tox [9]

Tox jest z założenia rozproszonym i szyfrowanym protokołem do wymiany wiadomości. Powstało kilkanaście implementacji klientów obsługujących go, co pozwala na komunikowanie się z użytkownikami różnych aplikacji. Wśród zaimplementowanych aplikacji są programy na komputery stacjonarne oraz smartfony. Wiadomości są przesyłane bezpośrednio między nadawcą i odbiorcą dlatego obie strony muszą być dostępne jednocześnie. Brak wsparcia dostarczania wiadomości gdy jedna strona jest niedostępna to duża wada wszystkich aplikacji implementujących wskazany powyżej rodzaj transmisji P2P. Jednym

z rozwiązań tego problemu zaproponowanym przez twórców protokołu jest skorzystanie z serwerów, którym użytkownik ufa i których zadaniem jest przekazywanie wiadomości do odbiorcy pod nieobecność nadawcy. Narusza to jednak założenie o rozproszeniu systemu (braku centralnych węzłów). Wsparcie dla komunikacji grupowej jest jednym z celów rozwoju protokołu.

## 2.8 ZeroChat, ZeroMail, BitMessage [10][11]

Przytoczone aplikacje realizują pomysły na komunikatory bazujące na mechanizmie podobnym do transakcji kryptowalutowych. Wysłanie wiadomości wymaga umieszczenia wiadomości w bloku, obliczenia funkcji skrótu z zadany prefiksem (proof of work) i umieszczenia bloku w łańcuchu (blockchain). Samo tylko wyliczenie funkcji skrótu powinno z definicji zająć około 4 minut [12], podczas gdy pozostałe komunikatory dążą do uzyskania czasu dostarczenia wiadomości bliskiego zeru (rozmowa w czasie rzeczywistym). Mimo tej znaczącej wady należy potraktować te projekty jako próbę stworzenia rozwiązania o innej architekturze niż dotychczas zaprezentowane (centralny serwer lub P2P). Przykładowymi zaletami architektury blockchain są: brak centralnego serwera lub centrów danych, nad którymi kontrolę ma jedna firma decydująca o sposobie ich działania lub ich wyłączeniu, skalowalność oraz brak możliwości ingerencji w zapisane dane — wraz z upływem czasu maleje prawdopodobieństwo, że treść wiadomości zapisana w blockchainie ulegnie zmianie (wynika to wprost z własności tej technologii). Być może w przyszłości wady uda się zminimalizować, a zalety architektury blockchain okażą się kluczowe.

# Koncepcja

W niniejszym rozdziale opisano technologie i protokoły wykorzystane do opracowania koncepcji i implementacji rozproszonego komunikatora grupowego. Konkretnie, są to: protokół BitTorrent przedstawiony w punkcie 3.1 oraz jego implementacja w języku JavaScript — WebTorrent — punkt 3.2.

## 3.1 BitTorrent [13]

BitTorrent to protokół komunikacyjny pozwalający na wymianę i dystrybucję plików przez Internet. Jego główną zaletą jest podział plików na części i możliwość pobierania tych części od użytkowników, którzy w danym momencie również uczestniczą w procesie udostępniania. Pozwala to na znaczne odciążenie serwera. W szczególności możliwe jest nawet wyłączenie serwera, a plik pozostanie dostępny do pobrania, jeśli tylko wszystkie jego fragmenty zostały przed wyłączeniem rozesłane do zainteresowanych komputerów — wystarczy, że jedna maszyna posiada daną część i podzieli się nią z pozostałymi.

### 3.1.1 Podstawowe pojęcia

Poniżej znajduje się lista najważniejszych pojęć związanych z protokołem wraz z krótkim wyjaśnieniem ich znaczenia:

- torrent — plik lub zbiór plików udostępnionych do pobrania.
- metaplik .torrent (.torrent metafile) — dodatkowy plik z metadanymi dotyczącymi udostępnionych plików. Zawiera między innymi:
  - nazwy i rozmiary plików,
  - liczbę i rozmiar fragmentów, na jakie zostały podzielone pliki,
  - listę skrótów SHA-1 fragmentów w celu weryfikacji poprawności,
  - adresy URL trackerów.
- piece, block — podczas przygotowywania torrenta pliki dzielone są na fragmenty (piece), a każdy taki fragment składa się z bloków (block). Blok jest najmniejszą jednostką, którą można przesłać przez sieć między użytkownikami. Aby udostępnić fragment użytkownik musi posiadać wszystkie jego bloki.

- info hash — 160-bitowa wartość będąca wynikiem funkcji skrótu SHA-1, której podawana jest część metapliku .torrent (nazwy plików i lista skrótów fragmentów). Info hash pozwala jednoznacznie zidentyfikować dany torrent.
- tracker — serwer, którego zadaniem jest przechowywanie adresów IP użytkowników pobierających dany torrent. Pozwala użytkownikom na znalezienie siebie nawzajem.
- klient (client) — program uruchomiony na komputerze użytkownika, który pozwala na pobieranie plików z wykorzystaniem protokołu BitTorrent.
- peer — węzeł (komputer, klient) pobierający i wysyłający fragmenty torrenta. Zazwyczaj nie posiada jeszcze wszystkich fragmentów.
- seed — peer posiadający wszystkie fragmenty.
- swarm — grupa peerów pobierających dany torrent.
- peer-to-peer (P2P) — sieć złożona z komputerów, które komunikują się ze sobą np. w celu wymiany plików. Peery tworzą sieć P2P.
- Distributed Hash Table (DHT) — rozproszona tablica mieszająca — sieć składająca się z węzłów, które umożliwiają zapisywanie i odczytywanie rekordów w formie klucz-wartość. Węzły dzielą między sobą zbiór wszystkich kluczy. W kontekście protokołu BitTorrent, DHT może zastąpić rolę trackera.
- magnet link — link pozwalający na uzyskanie metadanych torrenta bez konieczności pobierania metapliku .torrent. Link powinien zawierać przynajmniej info hash torrenta oraz listę trackerów.

### 3.1.2 Scenariusz pobrania torrenta

W celu pobrania torrenta niezbędne są następujące czynności:

1. Pobranie metapliku .torrent lub poznanie (kliknięcie) magnet linku identyfikującego dany torrent. Zazwyczaj informacje te można uzyskać na stronach internetowych katalogujących istniejące torrenty (wyszukiwarkach torrentów).
2. Uzyskanie listy trackerów z metapliku lub magnet linka.
3. Pobranie z trackera listy peerów uczestniczących w udostępnianiu torrenta.
4. Pobranie fragmentów torrenta od peerów.

### 3.1.3 Scenariusz utworzenia torrenta

W celu udostępnienia torrenta należy:

1. Stworzyć metaplik .torrent dla udostępnianych treści oraz utworzyć listę trackerów, które będą nadzorowały pobieranie treści.
2. Rozpocząć udostępnianie torrenta.
3. Umieścić metaplik .torrent w miejscu, z którego zainteresowani będą mogli go pobrać (np. na zewnętrznym serwerze). Alternatywnie można rozpowszechnić magnet link lub sam info hash — w ostatniej sytuacji osoba zainteresowana pobraniem torrenta musi znać listę trackerów, które koordynują pobieranie.

### 3.1.4 Jak komunikują się peery

BitTorrent używa 12 typów wiadomości do prowadzenia komunikacji pomiędzy peerami:

- hand-shake — wiadomość rozpoczynająca połączenie,
- bitfield — wskazuje, jakie fragmenty posiada peer,
- keep-alive — wiadomość podtrzymująca otwarte połączenie,
- port — informuje o zmianie portu,
- choke, unchoke, interested, not interested — 4 wiadomości informujące o zmianie stanu peera (związane z algorytmem z punktu 3.1.5),
- have — wiadomość informująca o tym, że peer otrzymał nowy fragment,
- request — żądanie fragmentu,
- piece — wiadomość zawierająca fragment torrenta,
- cancel — wiadomość anulująca żądanie fragmentu.

### 3.1.5 Choking algorithm

W idealnej sytuacji wymiana plików za pośrednictwem protokołu BitTorrent jest sprawiedliwa, to znaczy każdy peer może pobierać pliki, ale jednocześnie powinien udostępniać posiadane fragmenty innym. By zapobiec sytuacji, w której peer blokuje wysyłanie posiadanych fragmentów, wprowadzono algorytm „choking algorithm”. Wykorzystuje on 4 typy wiadomości spośród wspomnianych w punkcie 3.1.4. i polega na tym, że dany klient pozwala (unchoke) na pobieranie fragmentów od siebie tylko tym peerom, które posiadają i udostępnią klientowi swoje fragmenty. Pobieranie wymaga zatem kooperacji i wymiany interesujących, brakujących fragmentów. Bez tej wymiany połączenie zostaje przerwane (choke). Peer okresowo próbuje nawiązać współpracę z nowymi peerami.

### 3.1.6 Serwery potrzebne do działania protokołu

Protokół z założenia powinien być w pełni rozproszony i nie polegać na żadnych publicznych serwerach — jedynie na bezpośredniej komunikacji użytkowników końcowych (P2P). Niestety kilka usług serwerowych jest wciąż aktywnie wykorzystywanych do prawidłowego działania sieci:

- Serwer hostujący metapliki .torrent lub magnet linki pozwalający na wyszukiwanie interesujących plików po nazwie, tagach, innych właściwościach oraz oferujący na przykład statystyki torrenta czy komentarze. Może zostać zastąpiony wyszukiwarką torrentów wbudowaną w klienta, który wysyła zapytanie do podłączonych peerów, a one przekazują je dalej, aż do momentu otrzymania odpowiedzi. Przykładowym protokołem umożliwiającym wyszukiwanie zawartości w sieciach P2P jest Kademia. Protokół ten wykorzystuje rozproszoną tablicę mieszającą (DHT) — nie wymaga centralnego serwera.
- Tracker. Istnieją jednak rozwiązania umożliwiające śledzenie swarmu bez użycia zewnętrznego serwera np. DHT, PEX (Peer Exchange).

- Serwery pośredniczące w nawiązaniu połączenia dwóch klientów ukrytych w prywatnych sieciach IP (wykorzystujących translację adresów sieciowych).

### 3.1.7 Zalety i wady protokołu

Wśród zalet protokołu znajduje się przede wszystkim wspomniane na początku zmniejszone obciążenie serwera. Prędkość pobierania może osiągnąć wyższą wartość niż limit transferu wychodzącego z serwera — ograniczeniem jest jedynie dostępność pliku wśród peerów oraz limit transferu przychodzącego do danego klienta. Rozesłanie pliku o rozmiarze  $m$  do  $n$  odbiorców bez użycia protokołu wymagałoby transferu danych o rozmiarze  $m \cdot n$  z węzła udostępniającego, natomiast z użyciem protokołu ilość danych wysłanych przez nadawcę mieści się w przedziale  $\langle m; m \cdot n \rangle$ . Kolejną zaletą jest możliwość pobierania pliku nawet, jeśli oryginalny nadawca (twórca torrenta) jest niedostępny (zakładając oczywiście, że w swarmie rozesłane zostały najpierw wszystkie fragmenty pliku) — cecha ta okaże się przydatna podczas implementacji komunikatora grupowego.

Do wad protokołu należy zaliczyć zwiększone obciążenie oraz narzut komunikacyjny po stronie klienta — konieczność koordynacji pobierania i udostępniania, wysyłanie i odbieranie wiadomości kontrolnych. Wadą może być sam fakt, że klient zobowiązany jest do udostępniania pobieranego pliku. Jedną z ważniejszych kwestii, o jakie należy zadbać jest również dostępność pliku — klient nie pobierze całego torrenta jeśli nie znajdzie w sieci wszystkich jego fragmentów (tak zwany „problem ostatniego fragmentu”). Dodatkowo, problematyczne może być wyszukanie metapliku .torrent lub magnet linku odpowiadającego danemu torrentowi.

## 3.2 WebTorrent [14]

WebTorrent to biblioteka napisana w języku JavaScript, implementująca protokół BitTorrent. Dzięki zastosowaniu tego języka możliwe jest użycie protokołu w skrypcie wykonywanym w przeglądarce po wejściu na stronę internetową. Bibliotekę można również wykorzystać jako moduł w programie dla platformy Node.js. Funkcjonalność oferowana w obu przypadkach jest niemal identyczna (poza kilkoma wyjątkami wynikającymi z ograniczeń danej platformy).

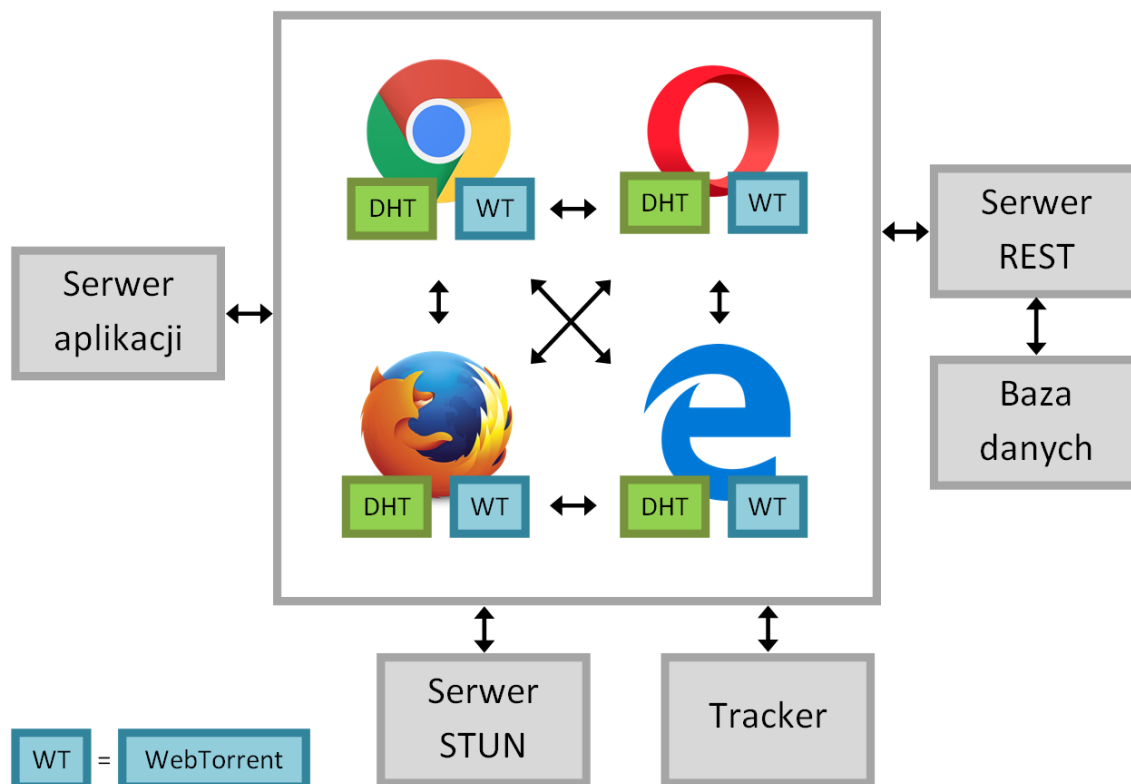
Główną różnicą pomiędzy wersjami biblioteki WebTorrent i protokołem BitTorrent jest wykorzystany protokół komunikacyjny. BitTorrent do komunikacji pomiędzy klientami wykorzystuje połączeniowy protokół TCP lub bezpołączeniowy  $\mu$ TP (Micro Transport Protocol), który bazuje na UDP. W przypadku biblioteki WebTorrent, klient serwerowy (Node.js) również wykorzystuje wspomniane protokoły, ale klient webowy już nie — w tym przypadku do komunikacji pomiędzy przeglądarkami użyto protokołu WebRTC.

WebRTC (Web Real-Time Communication) [15] jest zbiorem protokołów komunikacyjnych i interfejsów programistycznych (API), pozwalającym na komunikację w czasie rzeczywistym pomiędzy przeglądarkami dzięki wykorzystaniu połączeń P2P. WebRTC wymaga serwera pośredniczącego (signalling server) do nawiązania połączenia między przeglądarkami. Po poprawnym rozpoczęciu połączenia dane przesyłane są już bezpośrednio między użytkownikami. W przypadku biblioteki WebTorrent rolę signalling server pełnią trackery.

Komputery użytkowników rzadko posiadają publiczne adresy IP — zazwyczaj router stosujący translację adresów (NAT — Network Address Translation) przydziela im prywatny adres IP w danej podsieci. Często też użytkownicy korzystają z zapory sieciowej (firewall), co dodatkowo utrudnia nawiązanie bezpośredniego połączenia. W tych sytuacjach konieczne jest użycie techniki ICE (Interactive Connectivity Establishment). Polega ona na znalezieniu optymalnego sposobu nawiązania połączenia — w pierwszej kolejności używając wprost adresów IP danych maszyn. W przypadku niepowodzenia wykorzystywane są serwery STUN (Session Traversal Utilities for NAT) do przekazania faktycznego (publicznego) adresu IP danego komputera. Gdy i ta metoda zawiedzie, możliwe jest użycie serwera TURN (Traversal Using Relays around NAT), który funkcjonuje jako pośrednik pomiędzy komputerami. Ostatniej metody nie można jednak nazwać komunikacją P2P.

# Architektura

System komunikatora grupowego będący przedmiotem niniejszej pracy magisterskiej wykorzystuje częściowo architekturę klient-serwer, jednakże rola serwera została maksymalnie ograniczona, a częściowo architekturę rozproszoną peer-to-peer (przy użyciu protokołu BitTorrent). W pierwszym typie architektury można wyróżnić 3 warstwy logiczne (3-layer architecture) — klienci, serwer oraz baza danych. Rysunek 4.1 przedstawia opisywaną architekturę.



**Rys. 4.1:** Architektura komunikatora

Klient jest aplikacją webową typu Single Page Application. Oznacza to, że cały kod źródłowy oraz niezbędne widoki pobierane są z serwera („Serwer aplikacji” na rysunku 4.1) przy pierwszym połączeniu.

Oprócz wspomnianego serwera aplikacji umożliwiającego pobranie plików HTML, CSS



oraz JS do działania programu niezbędny jest drugi serwer o architekturze REST. Umożliwia on rejestrację i autoryzację użytkowników oraz jest odpowiedzialny za tworzenie konwersacji i przydzielanie do nich odpowiednich użytkowników.

Serwer REST komunikuje się z trzecią warstwą — bazą danych. Baza danych typu dokumentowego (NoSQL) przechowuje niezbędne minimum informacji o użytkowniku, dzięki którym możliwe jest prowadzenie rozmów.

Kolejnym modulem jest wbudowana w klienta biblioteka WebTorrent do obsługi protokołu BitTorrent. Pozwala ona na bezpośrednią komunikację pomiędzy klientami. Wykorzystywana jest do przekazywania wiadomości.

Z specyfikacji biblioteki i protokołu wynika między innymi konieczność utrzymywania dodatkowych serwerów — trackerów. Ze względu na fakt, że biblioteka jest oprogramowaniem open-source, istnieje możliwość samodzielnego uruchomienia tego typu serwera, jednak na potrzeby testów nie było to konieczne — istnieją w sieci publiczne trackery, z których można skorzystać.

Przedostatnim modulem koniecznym do prawidłowego działania komunikatora jest rozproszona tablica mieszająca (DHT) utrzymywana przez klienty, a używana do zapisywania i odczytywania info hasha najnowszej wysyłanej wiadomości. Klient w momencie rozpoczęcia udostępniania wiadomości zapisuje jej info hash w DHT, a odbiorcy mogą go odczytać i rozpocząć pobieranie komunikatu.

Ostatnim elementem są publiczne serwery STUN (Session Traversal Utilities for NAT). Dzięki nim możliwe jest skomunikowanie ze sobą dwóch klientów znajdujących się w sieciach, których router stosuje translację adresów sieciowych. Takie serwery również istnieją w sieci więc ich implementacja i uruchomienie nie było konieczne. Za łączenie się z nimi odpowiedzialna jest biblioteka WebTorrent.

## 4.1 Serwer REST i Baza Danych

Serwer został napisany w języku Python z użyciem biblioteki Eve [16], która w łatwy sposób pozwala uruchomić serwer REST oferujący podstawowe operacje CRUD (Create, Read, Update, Delete) na obiektach. Biblioteka automatycznie generuje dla obiektów ścieżki dostępu REST z odpowiednimi metodami HTTP oraz zapisuje zmiany dokonywane na obiektach w podłączonej bazie danych. Wybór tej biblioteki wynika z faktu, że serwer REST w projekcie pełni niewielką rolę — wraz z bazą danych przechowuje jedynie te dane, które nie powinny zostać utracone, a mogłyby, gdyby były zapisane jedynie w przeglądarce użytkownika — wtedy użytkownik straciłby dostęp do swojego konta i informacji o przynależności do konwersacji.

Serwer przechowuje dane w nierelacyjnej (dokumentowej) bazie danych MongoDB. Struktura dokumentów w bazie odzwierciedla bezpośrednio schemat zaprojektowany w serwerze.

W kolejnych punktach przybliżono szczegóły obiektów „użytkownik” i „konwersacja” — ich atrybuty i metody REST.

### 4.1.1 Użytkownicy i logowanie

Encja user posiada następujące atrybuty:

- id (nadawany automatycznie),
- username,
- email,
- password,
- salt.

W trakcie rejestracji serwer automatycznie uzupełnia pole salt, które wraz z przesłanym hasłem zostaje zapisane jako wynik funkcji skrótu kryptograficznego BCrypt. Zapewnia to wyższy poziom bezpieczeństwa niż przechowywanie hasła w takiej postaci, w jakiej zostało ono przesłane przez użytkownika. Niezalogowany użytkownik ma dostęp jedynie do metody POST na kolekcji users (rejestracja nowego użytkownika) oraz metody login (POST) — zalogowanie się do systemu. Dopiero po zalogowaniu możliwe jest odczytanie szczegółów swojego profilu lub profilu innej osoby (metodą GET z podaniem odpowiedniego id).

W przypadku prawidłowego zalogowania serwer odsyła użytkownikowi szczegóły jego konta oraz dane poświadczające prawidłowe zalogowanie. Kontrola dostępu do ścieżek REST udostępnionych tylko dla zalogowanych użytkowników wykorzystuje najprostszy mechanizm autoryzacji — nagłówek HTTP Basic Authentication, który klient wypełnia danymi otrzymanymi od serwera podczas logowania.

### 4.1.2 Konwersacje

Atrybuty encji conversations zostały wymienione poniżej:

- id (nadawany automatycznie),
- conversation\_id (nazwa konwersacji; może być zdefiniowana przez użytkownika lub nadana losowo przez serwer),
- user\_id (id użytkownika),
- user\_dht\_id (klucz, pod którym użytkownik umieszcza info hash najnowszej wiadomości dla danej konwersacji w DHT).

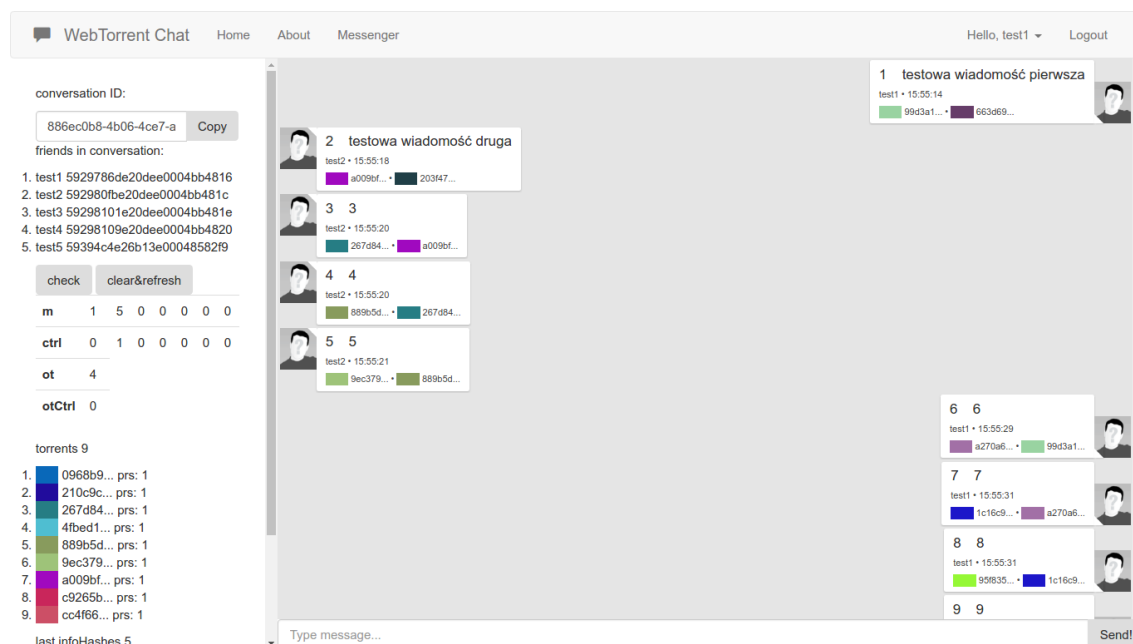
Ścieżki REST dla obiektu konwersacje są dostępne dla zalogowanego użytkownika. Taka osoba ma kilka możliwości:

- stworzenie i automatyczne dołączenie do nowej konwersacji lub dołączenie do już istniejącej konwersacji (metoda POST),
- pobranie informacji o użytkownikach, którzy uczestniczą w tej samej konwersacji co użytkownik, np. ich user\_dht\_id (metoda GET),
- wypisanie się z konwersacji (metoda DELETE).

## 4.2 Klient i biblioteka WebTorrent

Zaimplementowany w ramach projektu klient jest aplikacją webową typu Single Page Application. Do jej przygotowania wykorzystano framework AngularJS. Integralną częścią

programu jest moduł do obsługi protokołu BitTorrent — biblioteka WebTorrent. Wygląd komunikatora przedstawiony został na rysunku 4.2.



Rys. 4.2: Wygląd aplikacji klienta, ekran komunikatora

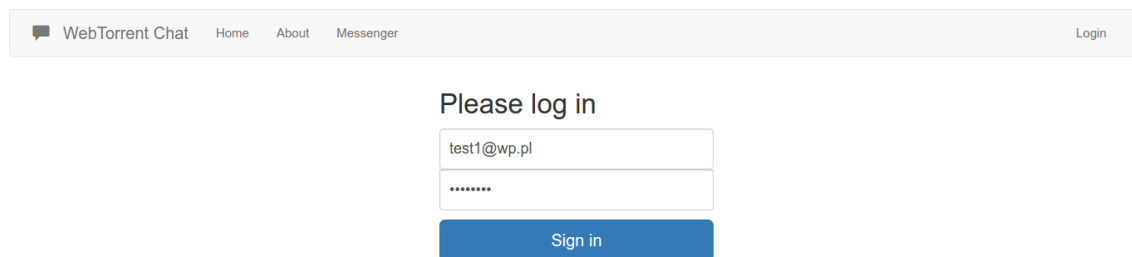
Program klienta pozwala użytkownikowi na zalogowanie się do systemu komunikatora. Następnie użytkownik może dołączyć do konwersacji podając jej nazwę (`conversation_id` z punktu 4.1.2) lub utworzyć nową konwersację. Głównym celem programu jest prowadzenie rozmowy z innymi użytkownikami, którzy są zapisani do tej samej konwersacji — użytkownicy wysyłają i odbierają wiadomości. Kluczową cechą komunikatora jest sposób, w jaki instancje programu wymieniają się wiadomościami — nowo utworzona wiadomość wraz z pewnymi metadanymi jest traktowana jako plik i upubliczniana w sieci P2P w sposób opisany w punkcie 3.1.3. Odbiór wiadomości przebiega według scenariusza opisanego w punkcie 3.1.2. W kolejnych punktach przybliżone zostaną szczegóły działania programu.

### 4.2.1 Logowanie

Na rysunku 4.3 zaprezentowano wygląd ekranu logowania. Formularz pozwala na podanie adresu email oraz hasła niezbędnych do zalogowania się do systemu komunikatora. Zatwierdzenie danych przyciskiem „Sign in” powoduje wysłanie ich do serwera REST, na którym przechodzą one weryfikację, czy dany użytkownik istnieje oraz czy podał prawidłowe hasło. W przypadku pomyślnego uwierzytelnienia, serwer odsyła szczegóły konta użytkownika i dane niezbędne do autoryzacji, a klient zapisuje je w pamięci przeglądarki (local storage).

### 4.2.2 Inicjalizacja modułów

Poprawne zalogowanie do serwisu pozwala na przejście do głównego ekranu komunikatora (przedstawiony na rysunku 4.2). Po wyświetleniu tego widoku następuje pobranie z ser-



**Rys. 4.3:** Ekran logowania

wera REST niezbędnych informacji o konwersacji (jeśli użytkownik już w jakiejś uczestniczy), uruchomienie modułu torrent (wykorzystującego bibliotekę WebTorrent, odpowiedzialnego za wymianę wiadomości) oraz modułu, którego zadaniem jest przechowywanie posiadanych wiadomości i kontakt z bazą danych wbudowaną w przeglądarkę. Szczegółowo ten proces został omówiony poniżej.

Jeśli użytkownik był zapisany do konwersacji to klient pobiera z serwera REST szczegółowe informacje o tej konwersacji (atrybuty przedstawione w punkcie 4.1.2) oraz dane o innych uczestnikach tej konwersacji — między innymi ich identyfikator `user_dht_id`, który jest niezbędny w procesie odbierania wiadomości. Jeżeli użytkownik nie uczestniczył jeszcze w żadnej konwersacji to wyświetlone zostaje pole umożliwiające podanie jej nazwy (`conversation_id`) — rysunek 4.4. Szczegóły dołączania do konwersacji opisano w punkcie 4.2.3.

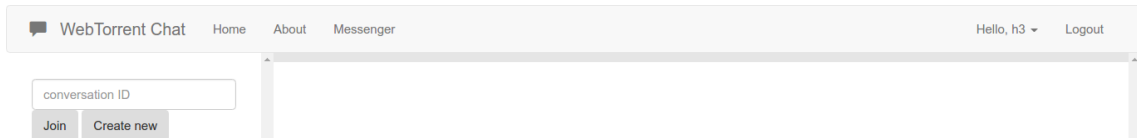
Kolejnym krokiem jest uruchomienie modułu odpowiedzialnego za przechowywanie wiadomości (nazywany dalej modułem wiadomości). Moduł korzysta z bazy danych typu klucz-wartość znajdującej się w przeglądarce (IndexedDB) do „trwałego” zapisywania wysłanych i odebranych wiadomości. Słowo „trwałego” jest ujęte w cudzysłów, ponieważ użytkownik może w każdej chwili usunąć dane. Jedynym sposobem, by je w takiej sytuacji odzyskać jest pobranie ich od innego użytkownika. Podczas inicjalizacji moduł wczytuje do pamięci podręcznej wszystkie wiadomości znajdujące się w bazie danych — dzięki temu będą one dostępne dla modułu torrent.

Ostatni etap przygotowania aplikacji do pełnego działania wymaga uruchomienia modułu torrent obsługującego komunikację poprzez protokół BitTorrent. W ramach inicjalizacji moduł włącza klienta sieci torrent i rozpoczyna ponowne udostępnianie posiadanych wiadomości.

### 4.2.3 Utworzenie lub dołączenie do konwersacji

W celu prowadzenia rozmowy z użytkownikami konieczne jest dołączenie do wspólnej konwersacji. Rysunek 4.4 pokazuje formularz umożliwiający utworzenie nowej rozmowy lub podanie nazwy istniejącej. Użytkownik może rozpocząć nową konwersację i poczekać, aż inni do niej dołączą lub samemu dołączyć do już istniejącej.

W pierwszym przypadku użytkownik wybiera opcję „Create new”. Aplikacja wysyła do serwera REST żądanie stworzenia konwersacji o losowej nazwie. Serwer automatycznie dołącza użytkownika do nowej konwersacji i odpowiada odsyłając nazwę konwersacji (`conversation_id`). Użytkownik zobaczy tę nazwę w polu „conversation ID”. Zadaniem użytkownika jest teraz rozesłanie nazwy konwersacji do osób, z którymi chce prowadzić



Rys. 4.4: Dołączanie do konwersacji

rozmowę. Może to zrobić, podobnie jak w przypadku komunikatora Darkwire (2.5), na przykład za pośrednictwem email, SMS lub przekazać osobiście.

Drugi przypadek zakłada, że użytkownik otrzymał od kogoś nazwę konwersacji — wpisuje ją w pole „conversation ID” i zatwierdza przyciskiem „Join”. Klient wysyła do serwera REST żądanie, a serwer zapisuje w bazie danych fakt dołączenia użytkownika do konwersacji. Serwer w odpowiedzi informuje klienta o pomyślnym zakończeniu procedury.

Dodatkowym założeniem, które należy spełnić w obu przypadkach przed dołączeniem do konwersacji jest posiadanie przez użytkownika unikalnego `user_dht_id` (atrybut z punktu 4.1.2). Jest to klucz w DHT, pod którym użytkownik umieszcza informację o najnowszej wysłanej przez siebie wiadomości (jej info hash). Wartość `user_dht_id` powinna być unikalna. Należy ją przesłać razem z żądaniem opisanym w powyższych przypadkach.

#### 4.2.4 Struktura komunikatu

Wiadomość przesyłana między uczestnikami rozmowy ma następującą strukturę:

- `content` (treść wiadomości),
- `type` (typ wiadomości, tekst, plik lub kontrolna),
- `timestamp` (znacznik czasu nadania wiadomości),
- `sender` (nadawca wiadomości),
- `previousInfoHash` (wartość info hash poprzedniej wiadomości).

Z własności protokołu BitTorrent wynika, że torrenty są niezmiennie (immutable) — zawartość plików lub ich liczba nie może ulec zmianie bez zmiany wartości info hash. Konieczne zatem było przyjęcie założenia, że każda nowa wiadomość wysłana przez użytkownika staje się nowym torrentem. Scenariusz, w którym użytkownik dodaje wiadomość jako kolejny plik do istniejącego torrenta jest na chwilę obecną niemożliwy do zrealizowania, chociaż istnieją próby wdrożenia takiej możliwości (w protokole BitTorrent [17, 18] oraz w bibliotece WebTorrent [19]). Niemniej jednak, zdecydowano o zachowaniu przyjętego założenia.

Zastosowanie znaczników czasowych pozwala na zdefiniowanie kolejności wiadomości — ich odbiorcy będą w stanie posortować odbierane komunikaty zgodnie z porządkiem FIFO. Znaczniki dodatkowo gwarantują globalne uporządkowanie wiadomości (total order) — jeśli wszyscy użytkownicy uczestniczący w jednej konwersacji odbiorą wszystkie wiadomości (każdy będzie posiadał identyczny zbiór), to zobaczą je w takiej samej kolejności. W trakcie implementowania projektu zdecydowano, że każda odebrana wiadomość będzie natychmiast wyświetlana — aplikacja nie czeka na pobranie starszych wiadomości wysłanych przez danego nadawcę. Narusza to co prawda oba wspomniane założenia, ale

jedynie tymczasowo, do momentu pobrania wszystkich wcześniejszych wiadomości. Od tej chwili założenia pozostają spełnione.

Użycie znaczników nie gwarantuje jednak przyczynowego uporządkowania (causal order), ponieważ zegary na maszynach użytkowników nie są zsynchronizowane. W większości przypadków *niewielka* różnica czasu pozwoli na zachowanie uporządkowania przyczynowego. *Niewielka* różnica czasu  $\Delta t$  pomiędzy zegarami na maszynach dwóch użytkowników  $U_N$  (użytkownik-nadawca) i  $U_O$  (użytkownik-odbiorca) oznacza, że  $\Delta t < \min(t) + t_0$ , gdzie:

- $\min(t)$  to minimalny czas, który upłynie od momentu nadania wiadomości  $m$  przez  $U_N$  (ustawienia czasu nadania w polu timestamp) do momentu odebrania wiadomości  $m$  przez  $U_O$  (wyświetlenia wiadomości  $m$  u odbiorcy),
- $t_0$  to czas potrzebny na napisanie i nadanie odpowiedzi zależnej przyczynowo od wiadomości  $m$  przez  $U_O$ , przy czym w ogólności może on wynosić 0 (zero).

Możliwy jest jednak następujący scenariusz:

1.  $\Delta t > \min(t)$ ,  $t_0 = 0$ .
2.  $T_N > T_O$ , gdzie  $T_N$  i  $T_O$  to stan zegara maszyny nadawcy/odbiorcy w danym momencie, a nierówność oznacza, że gdyby porównać w tym samym momencie zegary obu maszyn, to zegar maszyny nadawcy będzie wskazywał wyższą wartość.
3.  $U_N$  wysyła wiadomość  $m$ .
4.  $U_O$  odbiera wiadomość  $m$  i od razu na nią odpowiada ( $t_0 = 0$ ), wysyłając wiadomość  $n$ . Timestamp ustawiony w odpowiedzi  $n$  będzie miał niższą wartość niż timestamp otrzymany w wiadomości  $m$ . Zarówno  $U_N$  jak i  $U_O$  zaobserwują ten fakt — wiadomość  $n$  zostanie wyświetlona powyżej wiadomości  $m$  (co oznacza, że została wysłana wcześniej).

## 4.2.5 Wysłanie wiadomości i algorytm ZLT

Aby wysłać wiadomość użytkownik podaje jej treść w polu „Type message” i zatwierdza przyciskiem „Send”. Dalsze czynności wykonuje moduł torrent. Najpierw tworzy strukturę wiadomości (szerzej opisaną w punkcie 4.2.4). Następnie rozpoczyna udostępnianie komunikatu w sieci P2P zgodnie z opisem z punktu 3.1.3 (jest to zadanie biblioteki WebTorrent). Kolejnym krokiem jest zapamiętanie info hasha udostępnionej wiadomości. Klient zapisuje komunikat za pośrednictwem modułu wiadomości w bazie danych IndexedDB — kluczem jest info hash, a wartością struktura wiadomości.

W trakcie implementowania programu zaobserwowano, że wraz ze wzrostem liczby wiadomości umieszczanie każdej z nich w osobnym torrencie bardzo szybko prowadzi do nadmiernego zużywania zasobów, a nawet do błędnego zakończenia programu. Z tego powodu zdecydowano o zaprojektowaniu i wdrożeniu algorytmu zmniejszającego liczbę jednocześnie udostępnianych torrentów (nazywany dalej **algorytmem ZLT** — Zmniejszającym Liczbę Torrentów). Polega on na okresowym zastępowaniu torrentów zawierających pojedynczą wiadomość (poziom 0) jednym torrentem (nazywanym dalej **torrentem kontrolnym**), który składa się z wielu wiadomości (poziom 1). Algorytm wykonuje się rekurencyjnie — jeśli powstanie zbyt dużo torrentów kontrolnych na poziomie 1 to zostaną

one połączone w jeden torrent kontrolny na poziomie 2, itd. Algorytm uruchamiany jest jako kolejny krok po zleceniu zapisu do bazy danych. Został opisany szczegółowo poniżej:

1. Jeżeli na danym poziomie znajduje się zbyt dużo wiadomości (więcej niż zadany próg) to rozpocznij procedurę zamieniania wielu torrentów w jeden torrent kontrolny. W przeciwnym przypadku zakończ algorytm.
2. Stwórz listę wiadomości, które znajdują się w nowym torrencie kontrolnym.
3. Stwórz listę info hashy, które posiadają wiadomości znajdujące się na liście w punkcie 2. Dodatkowo, jeśli torrent kontrolny będzie zastępował inne torrenty kontrolne, stwórz listę info hashy zastępowanych torrentów kontrolnych.
4. Stwórz strukturę wiadomości (punkt 4.2.4), w polu „treść” umieść listy z punktu 3.
5. Usuń z klienta sieci P2P torrenty o info hashach znajdujących się na listach z punktu 3. Usunięcie wykonywane jest z opóźnieniem (domyślnie po 5 sekundach).
6. Rozpocznij udostępnianie pliku składającego się z wszystkich wiadomości z punktu 2 oraz wiadomości z punktu 4 — procedura identyczna jak wysyłanie zwykłej wiadomości. W bazie danych zapisywana jest wiadomość z punktu 4 oraz usuwane z niej są te zastąpione.
7. Jeśli stworzenie nowego torrenta kontrolnego spowodowało, że na jego poziomie znalazło się zbyt wiele torrentów, wykonaj algorytm dla tego poziomu (od punktu 1).

Po zakończeniu algorytmu należy wykonać ostatni krok, którym jest umieszczenie wartości najnowszego info hasha w DHT pod kluczem `user_dht_id`. Działanie DHT zostało szczegółowo opisane w punkcie 4.3.

Dla poprawnego działania programu kluczowe jest, by procedura wysyłania kolejnej wiadomości  $m_j$  rozpoczęła się dopiero po zakończeniu procedury dla poprzedniej wiadomości  $m_i$ . Wynika to z konieczności podania wartości info hash wiadomości  $m_i$  w strukturze wiadomości  $m_j$ .

Domyślnie algorytm umieszcza 5 wiadomości w torrencie kontrolnym. W teście przedstawionym w punkcie 5.8 zbadano działanie aplikacji, gdy ta liczba jest inna.

Zaletą algorytmu ZLT jest znaczne zmniejszenie liczby jednocześnie udostępnianych torrentów, a to obniża zużycie zasobów. Wadą natomiast jest powtórne wysyłanie wiadomości, które uczestnicy rozmowy mogli już wcześniej odebrać — przykładowo, klient odebrał 4 kolejne wiadomości będące pojedynczymi torrentami, a następnie pobrał torrent kontrolny zawierający 5 wiadomości, spośród których 4 już posiada i jedną nową. Oznacza to, że 80% przesłanych danych było zbędne.

## 4.2.6 Odebranie wiadomości

Procedura odebrania wiadomości rozpoczyna się okresowym sprawdzeniem najnowszej wartości info hash w DHT dla kluczy `user_dht_id` poszczególnych uczestników rozmowy. Moduł torrent sprawdza, czy torrenty o tych info hashach znajdują się na liście pobieranych torrentów lub czy zostały już pobrane i znajdują się na liście wiadomości w module wiadomości. Jeśli żaden z powyższych warunków nie jest spełniony, wtedy na podstawie wartości info hash tworzony jest magnet link i rozpoczyna się procedura pobrania torrenta



(punkt 3.1.2). Biblioteka WebTorrent pozwala na zarządzanie plikami w trakcie ich pobierania — możliwe jest odczytanie zawartości wiadomości pobranych w całości, podczas gdy pozostałe pliki są dalej przesyłane. Odczytana wiadomość zostaje zapisana za pośrednictwem modułu wiadomości w bazie danych IndexedDB oraz na liście wiadomości do wyświetlenia użytkownikowi. Oprócz tego moduł torrent sprawdza, czy poprzednia wiadomość wysłana przez nadawcę znajduje się na liście wiadomości lub liście pobieranych torrentów. Jeśli nie — moduł rozpoczyna procedurę pobierania tej wiadomości.

W sytuacji, gdy pobierany torrent był torrentem kontrolnym, moduł torrent usuwa z klienta sieci P2P torrenty, które dany torrent kontrolny zastępuje. Moduł zapisuje również w bazie danych brakujące wiadomości, o ile istnieje taka konieczność. Podobnie jak w przypadku zwykłego torrenta, jeśli brakuje poprzedniej wiadomości, moduł zacznie jej pobieranie.

Koncepcja wysyłania i odbierania wiadomości wykorzystująca protokół BitTorrent umożliwia odebranie wiadomości nawet w sytuacji, gdy jej nadawca nie jest dostępny. Warunkiem koniecznym jest oczywiście dostępność innego uczestnika rozmowy, który zdążył tę wiadomość pobrać, lub obecność wielu użytkowników, z których każdy posiada fragmenty wiadomości (pieces). W drugim przypadku niezbędne jest założenie, że każdy fragment (piece) torrenta został pobrany przez co najmniej jednego użytkownika.

## 4.2.7 Udostępnianie starych torrentów

W celu zmniejszenia liczby torrentów, które klient w danej chwili pobiera i wysyła, zdecydowano o wprowadzeniu mechanizmu ograniczającego udostępnianie starych wiadomości. Przyjęto założenie, że każdy uczestnik rozmowy będzie pobierał wszystkie nowe wiadomości minimum raz w tygodniu. Podczas inicjalizacji modułu torrent każdy klient rozpoczyna ponowne udostępnianie jedynie tych wiadomości i torrentów kontrolnych, których data wysłania jest młodsza niż 1 tydzień. Możliwy jest przypadek, w którym udostępniany torrent kontrolny zawiera wiadomości starsze niż 1 tydzień — przyjęto, iż jest to sytuacja zgodna z oczekiwaniami. Wiadomości te i tak przestaną być udostępniane maksymalnie w ciągu tygodnia ze względu na przedawnienie torrenta kontrolnego, który je zawiera.

Wadą tego rozwiązania jest brak możliwości przeczytania wiadomości starszych niż 1 tydzień przez osoby, które dopiero dołączyły do rozmowy. Zdecydowano, że nie jest to poważny błąd, a wręcz zaleta w sytuacji, gdy konwersacja zawiera dużą liczbę wiadomości. Dołączający klient zmuszony byłby do pobrania całej historii konwersacji, co może być operacją kosztowną komunikacyjnie.

## 4.2.8 Zbiór sąsiadów

W ramach pojedynczej konwersacji klienty wysyłają i odbierają wiadomości. Każda wiadomość zostaje umieszczona w osobnym torrencie, a te mogą zostać połączone w nowy torrent kontrolny przez algorytm ZLT. Wszelka wymiana danych pomiędzy klientami odbywa się zatem w swarmach, dla każdego torrenta z osobna. Struktura (np. łańcuch, pierścień czy graf pełny) jaką utworzą peery w swarmie zależy od postępu algorytmu wymiany danych. Możliwe jest ręczne zarządzanie listą peerów, z którymi dany klient posiada nawiązane połączenia, jednak w ramach systemu komunikatora nie zdecydowano się na



ten krok — rolę zarządcy wystarczająco dobrze spełniają algorytmy wbudowane w protokół BitTorrent. W trakcie działania aplikacji i tworzenia przez użytkowników nowych wiadomości, klienci dynamicznie dołączają do powstających swarmów oraz odłączają się od tych, które przestały funkcjonować, gdyż dany torrent został zastąpiony przez nowy (kontrolny).

Konwersacje są odseparowane od siebie, co oznacza, że klienci nie nawiązują połączeń z peerami znajdującymi się w swarmach należących do innej rozmowy.

## 4.3 DHT

Moduł DHT czyli rozproszona tablica mieszająca jest systemem składającym się z wielu węzłów pozwalającym na przechowywanie informacji. Węzłem staje się każdy klient systemu komunikatora. Węzły dzielą między sobą pewien zbiór kluczy i posiadają możliwość komunikowania się z innymi węzłami i przesyłania im posiadanych informacji. Zadaniem węzła jest przechowywanie informacji dotyczących przypisanego mu podzbioru kluczy. Każda informacja zapisywana w DHT posiada etykietę (w przypadku systemu komunikatora jest nią `user_dht_id`), która następnie zamieniana jest na klucz przez funkcję haszującą. Informacją w systemie komunikatora jest info hash najnowszej wysłanej przez klienta wiadomości.

Przed przystąpieniem do pracy nad projektem zakładano, że biblioteka WebTorrent jest wyposażona w mechanizm opisany powyżej. Jest to częściowo prawda — wersja biblioteki dla programów na platformę Node.js posiada taką możliwość. Niestety brakuje jej w wersji dla przeglądarek. Powody dla braku wsparcia DHT w bibliotece WebTorrent w przeglądarce są następujące:

- Przeglądarki mogą komunikować się bezpośrednio ze sobą jedynie dzięki protokołowi WebRTC. Funkcjonuje on poprawnie dla niewielkiej liczby jednoczesnych połączeń, jednak na potrzeby zaimplementowania DHT konieczne byłoby utrzymywanie większej liczby równoległych połączeń, co może prowadzić do szybkiego wyczerpania zasobów przeglądarki.
- Jednym ze sposobów zaimplementowania DHT dla biblioteki WebTorrent jest zaadaptowanie algorytmu Kademlia działającego już dla protokołu BitTorrent. Wymaga on jednak zapamiętywania listy węzłów, z którymi dany węzeł nawiązał dotychczas połączenie, a to oznacza, że połączenia WebRTC muszą pozostać otwarte (nie można ich zakończyć). Jest to o wiele bardziej kosztowne w przypadku WebRTC niż w oryginalnej implementacji, która korzysta z beipołączeniowego protokołu komunikacji UDP.
- Kolejnym, bardzo istotnym problemem jest brak wsparcia protokołu WebRTC dla dedykowanych wątków roboczych (WebWorker, ServiceWorker, itp.). Bez tego wsparcia kod obsługujący DHT musi być wykonywany w głównym wątku danej strony internetowej, a to może doprowadzić nawet do całkowitego zablokowania interfejsu. W chwili obecnej trwają próby wprowadzenia wsparcia protokołu dla wątków roboczych — być może w przyszłości ten problem zostanie rozwiązany.

W zaistniałej sytuacji przeprowadzono poszukiwania alternatywnej biblioteki udostępnia-

jącej mechanizm DHT dla przeglądarek. Znalezione biblioteki pozwalały programiście na zdefiniowanie kanału wymiany informacji pomiędzy węzłami [20] lub posiadały działający mechanizm [21]. Jednakże w trakcie wstępnych testów okazało się, że żadna biblioteka nie zapewnia satysfakcjonujących rezultatów — informacje były przesyłane zbyt długo lub w ogóle nie docierały do niektórych węzłów.

Ze względu na brak zaimplementowanego mechanizmu DHT w bibliotece WebTorrent oraz niesatysfakcjonujące alternatywy zdecydowano o zastąpieniu tego mechanizmu atrapą (mock). Funkcjonalność tego rozwiązania pozostaje niezmienna — każdy klient ma możliwość zapisania najnowszej wartości info hash pod określonym kluczem (`user_dht_id`) oraz odczytania tej informacji. Zmianie uległ sposób przechowywania informacji — zadanie to nie należy już do sieci powiązanych ze sobą klientów, a stało się jednym z zadań serwera REST. Serwer udostępnia ścieżkę REST `/dht`, która umożliwia następujące akcje:

- metoda HTTP GET z parametrem `user_dht_id` pozwala na pobranie wartości przechowywanej dla klucza `user_dht_id`,
- metoda POST pozwala na zapisanie wartości dla klucza `user_dht_id`.

## 4.4 Bezpieczeństwo i kryptografia w języku JavaScript

Mobilne i dektopowe klienty aplikacji Bleep (punkt 2.2) oraz aplikacji Signal (punkt 2.3) oferują szyfrowanie wiadomości. Webowy klient aplikacji Darkwire (punkt 2.5) również. W ramach niniejszej pracy magisterskiej zbadano sposób implementacji tego mechanizmu w przytoczonych aplikacjach oraz podjęto próbę zaprojektowania go dla systemu komunikatora.

Poufność wiadomości w systemie komunikatora jest konieczna, ponieważ każdy użytkownik systemu (ale też osoba spoza niego) ma możliwość pobrania i odczytania treści wiadomości przesyłanych jako torrenty. Gdyby wiadomość była zaszyfrowana, posiadanie jej w takiej formie nie powoduje negatywnych skutków. Głównym problemem do rozwiązania jest przechowywanie kluczy pozwalających na odczytanie zaszyfrowanych wiadomości i podpisywanie wysyłanych treści. Problem przechowywania dotyczy zarówno zarządzania kluczami w trakcie pracy aplikacji jak i po jej zamknięciu — zapisanie kluczy w pamięci trwałej. Klucz prywatny nie powinien być wysyłany poza urządzenie, na którym został wygenerowany.

Ostatecznie zrezygnowano z wdrożenia szyfrowania wiadomości, choć technicznie jest to osiągalne, ponieważ mechanizm ten posiadałby istotne wady mogące powodować brak gwarancji poufności przesyłanych danych. Podsumowanie badań opisano w poniższych punktach.

### 4.4.1 Darkwire

Aplikacja webowa Darkwire automatycznie szyfruje wiadomości w przeglądarce użytkownika przed wysłaniem ich do serwera. Serwer przesyła wiadomości do odbiorców, a po

dotarciu do nich, treść jest odszyfrowywana. W programie zastosowano kryptografię klucza publicznego do stworzenia klucza symetrycznego danej wiadomości. Aplikacja korzysta z web cryptography API [22] do zarządzania kluczami kryptograficznymi.

Scenariusz uruchomienia programu i przesłania zaszyfrowanej wiadomości jest następujący [23]:

1. Podczas uruchomienia, program tworzy nową rozmowę (chat room) oraz parę kluczy (publiczny i prywatny).
2. W momencie dołączenia nowej osoby do rozmowy, aplikacje użytkowników wymieniają się kluczami publicznymi uczestników rozmowy.
3. W celu wysłania wiadomości program najpierw generuje 3 nowe klucze: symetryczny klucz sesji, klucz podpisujący (umożliwiający weryfikację sygnatury wiadomości) oraz wektor inicjujący (initialization vector).
4. Treść wiadomości zostaje zaszyfrowana przy pomocy klucza sesji i wektora inicjującego. Oprócz tego tworzona jest sygnatura wiadomości przy użyciu klucza podpisującego.
5. Klucz sesji i klucz podpisujący zostają zaszyfrowane kluczami publicznymi odbiorców, dla każdego z osobna — ta technika pozwala na uniknięcie szyfrowania całej treści wiadomości osobno dla wszystkich odbiorców.
6. Ostatnim krokiem przed wysłaniem wiadomości jest przygotowanie paczki danych do wysłania — składa się ona z zaszyfrowanej wiadomości, wektora inicjującego, sygnatury wiadomości, zaszyfrowanego klucza sesji oraz zaszyfrowanego klucza podpisującego.
7. Odbiorca, po otrzymaniu takiej paczki danych, odszyfrowuje swoim kluczem prywatnym klucze sesji i podpisujący. Używając odszyfrowanego klucza sesji i wektora inicjującego odbiorca odszyfrowuje treść wiadomości i może zweryfikować sygnaturę wiadomości odszyfrowanym kluczem podpisującym.

Przedstawiony scenariusz gwarantuje poufność wiadomości i autentyczność odbiorcy podczas normalnej pracy aplikacji. Istnieją jednak wektory ataku pozwalające na zdobycie kluczy prywatnych i w konsekwencji na odczytanie przesłanych wiadomości czy przejęcie kontroli nad kontem użytkownika i podszywanie się pod nadawcę. Głównym problemem z punktu widzenia bezpieczeństwa jest przechowywanie klucza prywatnego jako wartość zmiennej w skrypcie JavaScript (lub w niezabezpieczony sposób w local storage czy IndexedDB). Wykorzystując choćby atak XSS (Cross-site scripting), czyli uruchomienie własnego skryptu osadzonego w treści atakowanej strony, można pozyskać wrażliwe dane. Zmiany w kodzie strony może dokonać nie tylko cracker, ale również twórca oprogramowania z własnej woli lub zmuszony przez trzecią stronę (np. agencję rządową).

Użycie powyższej metody do przechowywania wrażliwych danych może prowadzić do ich kradzieży i ujawnienia treści wiadomości. Jako usprawiedliwienie można podać przykład Messengera (punkt 2.1), który nie stosuje w ogóle szyfrowania wiadomości — jedynym zabezpieczeniem jest przesyłanie ich do serwera szyfrowanymi kanałami. W bazie danych są one zapisywane w takiej samej formie, w jakiej zostały wprowadzone przez użytkownika. Zdobycie dostępu do bazy danych przez osoby niepowołane spowoduje ujawnienie treści

wiadomości. Niemniej jednak, taki wektor ataku wydaje się być mniej prawdopodobny — bazy danych są zazwyczaj lepiej zabezpieczone niż komputery użytkowników.

W aplikacjach webowych istnieje jeszcze inna możliwość przesyłania i przechowywania poufnych informacji (np. tokenów uwierzytelniających), a mianowicie ciasteczka (cookies). Te małe fragmenty tekstu są wysyłane przez serwer do klienta w nagłówku HTTP, a przeglądarka odsyła je wraz z kolejnymi żadaniami. W celu zabezpieczenia tych informacji stosuje się szyfrowany protokół (np. HTTPS) oraz ustawia niezbędne flagi (jak choćby `httpOnly`, `secure`, `domain`, `path` itd.). Ustawienie pierwszej flagi skutkuje brakiem dostępu do zawartości ciasteczka z poziomu kodu JavaScript. Niestety w systemie komunikatora ten sposób przechowywania poufnych informacji nie może być wykorzystany, gdyż większość operacji wykonywanych jest z poziomu skryptu, a komunikaty przesyłane są innymi protokołami niż HTTP.

#### 4.4.2 Bleep i Signal

Natywne aplikacje mobilne mają nad aplikacjami webowymi istotną przewagę — umożliwiają programistom dostęp do pamięci trwałej urządzenia. Dodatkowo, systemy mobilne zwiększają bezpieczeństwo przechowywanych informacji poprzez izolację danych zapisywanych przez konkretną aplikację — inne programy nie mają dostępu do tych obszarów. Ta możliwość pozwala na znacznie bezpieczniejsze przechowywanie poufnych danych. Wszelkie operacje związane z szyfrowaniem odbywają się na urządzeniach użytkowników.

Oprócz tego, twórcy przytoczonych aplikacji zastosowali znacznie bardziej rozbudowany mechanizm przesyłania wiadomości niż w przypadku aplikacji Darkwire. W aplikacji Signal wprowadzono choćby „The Double Ratchet Algorithm” [24] — algorytm gwarantujący własność utajnienia przekazywania (forward secrecy). Każda wiadomość jest szyfrowana innym kluczem. Ponadto, nawet jeśli cracker uzyskałby dostęp do pojedynczej wiadomości i jej metadanych, nie może na ich podstawie obliczyć wartości poprzednich ani przyszłych kluczy. Podobny mechanizm oferuje aplikacja Bleep [25].

Twórcy aplikacji Bleep umożliwiają użytkownikom wysyłanie wiadomości pod nieobecność odbiorcy — zapisywana jest ona wtedy w DHT. Węzłami w DHT są aplikacje użytkowników, więc dane przechowywane w DHT również muszą być zaszyfrowane [26]. Algorytm jest podobny do przedstawionego powyżej z niewielką zmianą — nowy klucz generowany jest nie dla każdej wiadomości, a dla paczki wiadomości nadanych podczas nieobecności odbiorcy. Gdy odbiorca staje się dostępny, odczytuje wiadomości i generuje nowy klucz.

#### 4.4.3 Inne pomysły i wnioski

Badanie rozwiązań wdrożonych w powyższych aplikacjach doprowadziło do powstania innych koncepcji rozwoju systemu komunikatora prowadzących do zabezpieczenia konwersacji między użytkownikami. Przedstawione zostały poniżej.

- Przechowywanie kluczy publicznych i prywatnych w bazie danych z dostępem poprzez serwer REST. Klucze mogą być generowane przez serwer lub przez klienta. Rozwiązanie nie jest poprawne jeśli zakłada się, że nikt poza nadawcą i odbiorcami

nie ma prawa odczytać treści wiadomości. Jeśli jednak założenie to zostanie złagodzzone i oprócz wspomnianych osób dostęp do treści wiadomości będzie miał zaufany serwer, rozwiązanie mogłoby zostać wprowadzone. W tej sytuacji uzyskuje się poziom poufności podobny do aplikacji Messenger (punkt 2.1), w przypadku której wiadomości również mogą być odczytane z bazy danych.

- Rozwinięciem poprzedniego rozwiązania byłoby wygenerowanie pary kluczy po stronie klienta oraz dodatkowe zabezpieczenie ich w taki sposób, by jedynie użytkownik miał możliwość ich odczytania po podaniu hasła dostępu. Następnie, bezpieczne dane są wysyłane do serwera i tam przechowywane. Na żądanie klienta, na przykład podczas uruchamiania aplikacji, dane są pobierane, użytkownik podaje hasło i uzyskuje dostęp do poufnych informacji. Niestety, oba rozwiązania są w dalszym ciągu podatne na kradzież danych — odszyfrowane klucze są przetwarzane przez skrypt i mogą zostać przechwycone.
- Ostatnim, póki co koncepcyjnym, pomysłem jest zastosowanie Web Cryptography API [27, 28]. Konieczne jest jednak założenie, że nie będzie istniała możliwość odczytania czy importu kluczy prywatnych z poziomu kodu JavaScript. Wszelkie operacje na poufnych danych (generowanie kluczy, szyfrowanie i odszyfrowywanie danych) powinny być wykonywane przez algorytmy zaimplementowane bezpośrednio w przeglądarce — dzięki temu unika się przechowywania kluczy prywatnych w zmiennych i ich przetwarzania przez kod JavaScript, a to z kolei uniemożliwia ich kradzież. Algorytmy dostępne byłyby przez API.

## 4.5 Dalszy rozwój

Poniżej opisano pokrótce kilka idei rozwijających system komunikatora.

- Zaimplementowanie relacji znajomości pomiędzy użytkownikami. Dzięki temu osoba tworząca konwersację będzie mogła od razu dopisać do niej uczestników wybierając ich z listy swoich znajomych. Kwestią do rozstrzygnięcia pozostaje w tej sytuacji wybranie, kto ma przypisać uczestnikom wartość `user_dht_id` (osoba tworząca konwersację czy każdy jej członek z osobna).
- Zagwarantowanie spójności przyczynowej wiadomości i wyświetlanie komunikatów dopiero, gdy pobrane zostały wszystkie poprzednie. Obie własności spowodują bardziej realistyczne wrażenie prowadzenia rozmowy. Jeśli użytkownik wysłał wiadomość *b* w odpowiedzi na otrzymaną wiadomość *a* to pojawią się one w oknie rozmowy w odpowiedniej kolejności — najpierw *a*, potem *b*. Jest to możliwe do osiągnięcia, jeśli wiadomości będą zawierały zegar wektorowy z wartościami ostatnio odebranych wiadomości.
- Szyfrowanie wiadomości. Sposoby implementacji oraz ich znane wady opisano w punkcie 4.4. Niemniej jednak ta własność jest niezbędna, by treść wiadomości nie mogła zostać odczytana przez osoby niepowołane (użytkowników nie należących do danej konwersacji).
- Zagwarantowanie dostarczenia wiadomości w sytuacji, gdy użytkownik uruchamia

aplikację i żaden z uczestników konwersacji nie jest dostępny. Problem ten może zostać rozwiązany na przykład poprzez dodanie serwera do konwersacji jako uczestnika. Serwer ten wykonywałby taki sam kod programu co inni klienci, z jednym wyjątkiem — nie miałby możliwości odszyfrowywania wiadomości ani nadawania własnych komunikatów. Alternatywnym sposobem rozwiązania problemu byłoby powierzenie przekazywania wiadomości programom należącym do znajomych użytkowników. Oni również nie mieliby możliwości odczytania i modyfikacji treści wiadomości.

- Przechowywanie listy ostatnich wartości info hash w DHT (zamiast pojedynczej wartości) oraz przesyłanie jej w strukturze wiadomości (również zamiast pojedynczej wartości previousInfoHash). Dzięki temu możliwe byłoby równoległe pobieranie wiadomości przez dołączających klientów. Ponadto, to rozwiązanie pozwala na zmniejszenie skali problemu nadpisywania wartości w DHT — jeśli użytkownik wysłał wiadomości w szybkim tempie, to możliwe jest nadpisanie wartości info hash w DHT, zanim ktokolwiek zdąży ją odczytać.
- Modyfikacja protokołu BitTorrent pozwalająca na ponowne wykorzystanie istniejących swarmów, ponieważ konieczność dołączania do nowego swarmu dla każdej wysłanej wiadomości powoduje duży narzut komunikacyjny wynikający z nawiązywania połączenia pomiędzy peerami. Z dużym prawdopodobieństwem w nowych swarmach znajdują się te same peery, które uczestniczyły w udostępnianiu poprzednich wiadomości. Alternatywnym sposobem rozwiązania tego problemu jest możliwość dodawania nowych plików do istniejącego torrenta bez zmiany wartości info hash — istnieją propozycje wprowadzenia takiej modyfikacji protokołu [17, 18]. Klient dołącza wtedy do tylu swarmów, ilu jest uczestników rozmowy i nie opuszcza ich aż do momentu, w którym użytkownik zdecyduje o wyłączeniu aplikacji. Każda wiadomość byłaby wtedy nowym plikiem w tym samym, istniejącym już torrentcie. Zaletą wynikającą z omówionej modyfikacji jest znaczne zmniejszenie liczby swarmów, w których klient musi się jednocześnie znajdować. Dodatkowo, w drugim rozwiązaniu możliwe byłoby zrezygnowanie z algorytmu ZLT oraz wyeliminowanie ponownego wysyłania tych samych wiadomości w torrentach kontrolnych.
- Możliwość wysyłania plików (zdjęć, filmów). Komunikatory wymienione w rozdziale 2 posiadają taką funkcjonalność.

# Wyniki testów

Po zakończeniu prac nad systemem komunikatora przeprowadzono serię testów mających na celu zbadanie wydajności aplikacji. W punkcie 5.1 przedstawiono parametry, w przypadku których zmiana wartości mogłaby wpłynąć na wydajność aplikacji. W punkcie 5.2 przybliżono scenariusze testowe, a w punkcie 5.3 opisano specyfikację maszyn testowych. W dalszej części znajdują się szczegółowe opisy poszczególnych testów oraz wnioski, które z nich wynikają.

## 5.1 Kryteria

Wymienione poniżej parametry umożliwiają symulację różnych sytuacji podczas testów. Zmiana ich wartości może pozytywnie lub negatywnie wpływać na wydajność aplikacji. Pod uwagę wzięto następujące kryteria:

1. rozmiar wiadomości,
2. liczba uruchomionych klientów (dostępnych użytkowników) oraz liczba klientów wysyłających wiadomości,
3. liczba wiadomości w konwersacji,
4. liczba wiadomości, która zostaje umieszczona w torrencie kontrolnym,
5. sposób podłączenia klientów do sieci — czy klienci znajdują się w jednej lub różnych podsieciach oraz czy jest to sieć domowa, czy laboratoryjna,
6. zbadanie, czy uruchomienie w jednej sieci wielu klientów obsługujących różne konwersacje wpływa na wydajność.

## 5.2 Scenariusze testowe

Zaproponowano następujące scenariusze:

1. Jeden nadawca wysyła wszystkie wiadomości bez czekania na odpowiedzi — zmierzono czas otrzymania przez wszystkich odbiorców pełnej listy wiadomości.
2. Jeden nadawca  $U_N$  wysyła wiadomość  $m$  i czeka na odpowiedzi od wszystkich odbiorców. Odpowiedzi są wysyłane natychmiast po otrzymaniu wiadomości  $m$ . Zmie-

rzono czas otrzymania poszczególnych odpowiedzi przez nadawcę  $U_N$  od momentu nadania wiadomości  $m$ . Scenariusz ma symulować naturalną rozmowę, w której uczestnicy odpowiadają na otrzymaną wiadomość dopiero, gdy ją otrzymają. Pomiar czasu zaczyna się w momencie wysłania wiadomości przez nadawcę  $U_N$ , a kończy się w momencie otrzymania przez nadawcę  $U_N$  odpowiedzi od danego odbiorcy (dla każdego odbiorcy z osobna). Czas przesłania wiadomości w jedną stronę jest (uśredniając) połową uzyskanego wyniku. Nadawca  $U_N$  wyśle kolejną wiadomość dopiero po otrzymaniu wszystkich odpowiedzi.

3. Jeden klient dołącza do konwersacji. Zmierzono czas otrzymania pełnej listy wiadomości przez dołączającego klienta.

## 5.3 Specyfikacja komputerów i środowiska testowe

Część testów przeprowadzono w laboratorium wyposażonym w komputery stacjonarne posiadające adresy IP w tej samej podsieci — komputer tego typu będzie dalej oznaczony jako K1. Dodatkowo, w powyższych testach wykorzystano komputer podłączony do odrębnej podsieci (nazwany dalej K2). Druga część testów została przeprowadzona na komputerach podłączonych do jednej sieci domowej — w tych testach wykorzystano komputer K3 oraz komputer K4. Podział testów na części przeprowadzane w różnych środowiskach pozwolił na zwiększenie różnorodności konfiguracji oraz zaobserwowanie zjawisk, które występują w jednym z środowisk, a nie zachodzą w drugim — szczegóły zostały opisane w dalszej części rozdziału.

Specyfikacja komputerów K1 w laboratorium jest następująca:

- procesor 8 rdzeniowy,
- 32 GB pamięci RAM,
- system operacyjny Linux (OpenSUSE 42.1),
- przeglądarka Google Chrome (wersja 56),
- prędkość łącza w laboratorium: pobieranie 400 Mbps, wysyłanie 200 Mbps (komputery w laboratorium posiadają adresy IP w tej samej podsieci).

Specyfikacja komputera K2 (w laboratorium) oraz K3 (w sieci domowej):

- procesor 4 rdzeniowy,
- 8 GB pamięci RAM,
- system operacyjny Linux (Ubuntu 16.04),
- przeglądarka Google Chrome (wersja 59) + przeglądarka Opera (wersja 45) (symulowanie 2 klientów na jednej maszynie, jeśli testy odbywały się w sieci domowej)
- prędkość łącza w laboratorium (K2): pobieranie 400 Mbps, wysyłanie 200 Mbps,
- prędkość łącza w sieci domowej (K3): pobieranie 6 Mbps, wysyłanie 1 Mbps.

Specyfikacja komputera K4:

- procesor 4 rdzeniowy,

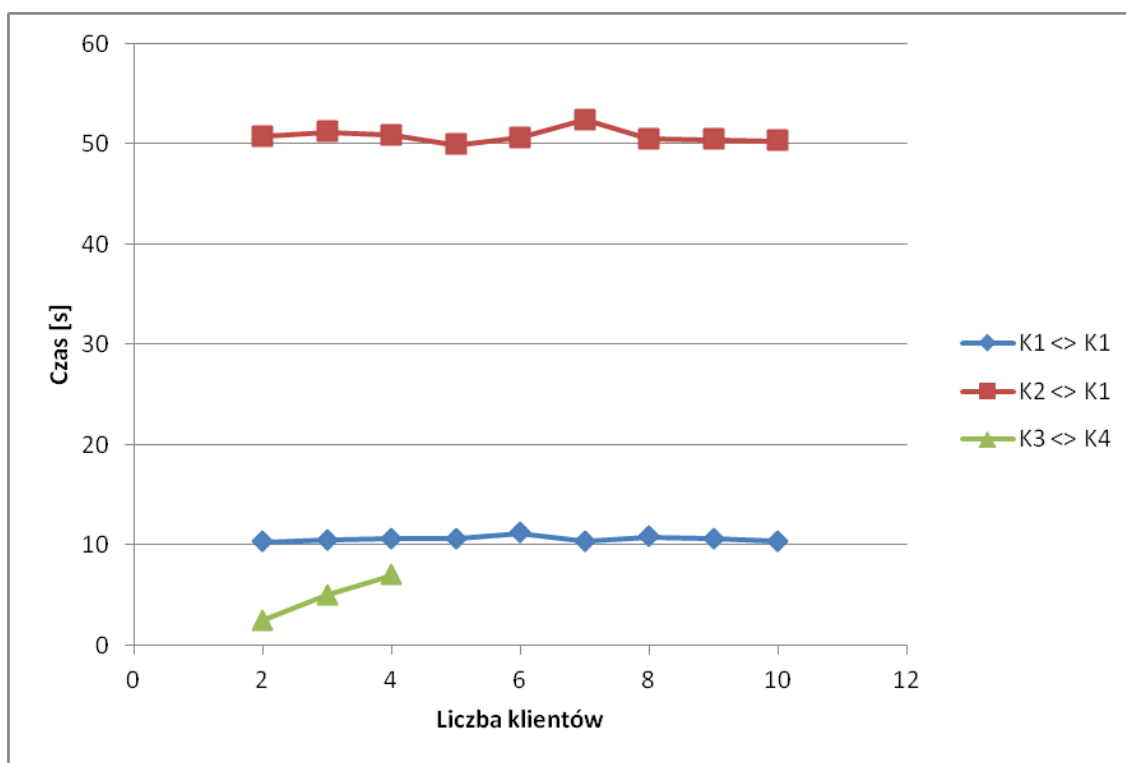


- 8 GB pamięci RAM,
- system operacyjny Windows 7,
- przeglądarka Google Chrome (wersja 59) + przeglądarka Opera (wersja 45) (symulowanie 2 klientów na jednej maszynie)
- prędkość łącza w sieci domowej: pobieranie 6 Mbps, wysyłanie 1 Mbps.

## 5.4 Wysyłanie pojedynczej wiadomości

Celem testu było sprawdzenie czasu przesłania pojedynczej wiadomości od nadawcy do odbiorcy (scenariusz 1). Wynik stanowi punkt odniesienia dla kolejnych testów, w których wysyłana jest większa liczba wiadomości w różnych konfiguracjach testowanych kryteriów.

Pomiar czasu rozpoczynał się w momencie nadania wiadomości, a kończył dla każdego odbiorcy z osobna w momencie odebrania przez niego wiadomości. W ramach pojedynczego testu pomiar został uśredniony. Otrzymane czasy zostały zaprezentowane na wykresie 5.1. Na osi poziomej oznaczono liczbę klientów uruchomionych w trakcie testu — jeden z nich jest nadawcą, a pozostali odbiorcami. Na osi pionowej znajduje się średni czas odebrania wiadomości przez poszczególnych odbiorców.



Rys. 5.1: Czas odebrania pojedynczej wiadomości w zależności o liczby klientów

Najniższy czas dostarczenia notowany jest w sieci domowej (K3 <> K4). Na komputerach K3 i K4 były uruchomione 1 lub 2 klienty (w osobnych oknach przeglądarki) — stąd możliwość przetestowania od 2 do 4 klientów. Średni czas wynosi od około 2 sekund dla 2 klientów do 7 sekund dla 4 klientów, jednakże szczegółowe pomiary wykazały

znaczne odchyły od średniej — minimalny czas wyniósł ok. 0,5 sekundy, podczas gdy maksymalny mógł wynieść nawet 30 sekund. Na czas przesyłu ma wpływ między innymi częstotliwość sprawdzania wartości info hash w DHT — domyślnie są to 2 sekundy. Jeśli odbiorca sprawdzi wartość tuż przed ustawieniem nowej wartości przez nadawcę to dowie się o nowej wiadomości dopiero przy kolejnym sprawdzeniu.

Średnio 10-11 sekund trwa przesyłanie wiadomości pomiędzy komputerami K1 ( $K1 \leftrightarrow K1$ ). Czas nie ulega zmianie, gdy wiadomość odbierana jest przez większą liczbę klientów.

Najdłużej wiadomość była przesyłana między komputerami K2 i K1 ( $K2 \leftrightarrow K1$ ). Minimalny czas odebrania wiadomości wyniósł około 50 sekund, a średni był niewiele wyższy — ok 51 sekund. Na bazie logów aplikacji zaobserwowano, że komunikaty charakterystyczne dla protokołu BitTorrent (punkt 3.1.4) pojawiają się dopiero po upływie minimalnego czasu (50 sekund) — następuje wtedy nawiązanie połączenia pomiędzy klientami.

Wyniki testu wskazują na jeszcze jedną zależność — przesłanie wiadomości w sieci lokalnej (czy to laboratoryjnej, czy domowej) trwa krócej niż przesłanie pomiędzy komputerami znajdującymi się w różnych podsięciach.

## 5.5 Wpływ rozmiaru wiadomości

Nie zaobserwowano istotnego wzrostu czasu przesyłania wiadomości w zależności od długości tekstu będącego treścią komunikatu.

## 5.6 Wysyłanie i oczekiwanie na odpowiedź

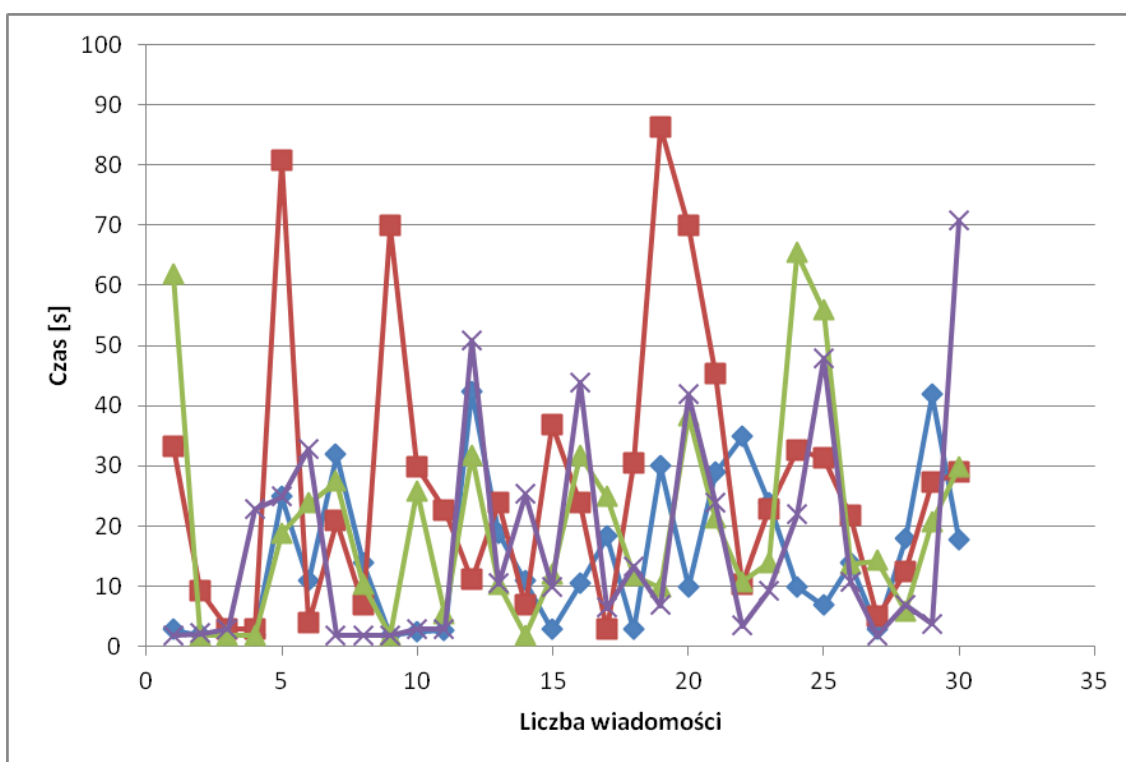
Bardziej rozbudowanym testem, w porównaniu do przedstawionego w punkcie 5.4, jest scenariusz 2. Polega on na wysłaniu wiadomości i czekaniu na odpowiedzi od wszystkich uczestników rozmowy lub wybranej podgrupy. W drugim przypadku pozostałe klienty mają za zadanie jedynie bierne odbieranie i przekazywanie komunikatów.

Wykres 5.2 przedstawia wyniki 4 powtórzeń testu w sieci domowej. Wykres 5.3 prezentuje uśrednione wyniki wspomnianych testów. Testy przeprowadzono na 2 komputerach K3 i K4 (w sieci domowej). W testach brały udział 2 klienty — nadawca (wysła wiadomość i czeka na odpowiedź) oraz odbiorca (czeka na wiadomość i natychmiast na nią odpowiada). Na osi poziomej zaznaczono liczbę wiadomości wysłanych przez nadawcę w ramach testu, a na osi pionowej czas przesyłania pojedynczego komunikatu pomiędzy komputerami. Pomiar czasu w scenariuszu 2 obejmuje przesłanie wszystkich wiadomości w danej iteracji (wiadomość od nadawcy i wszystkie odpowiedzi), ale czas przedstawiony na wykresie jest średnią czasów przesyłu — w tym wypadku średnią czasów przesyłania 2 wiadomości.

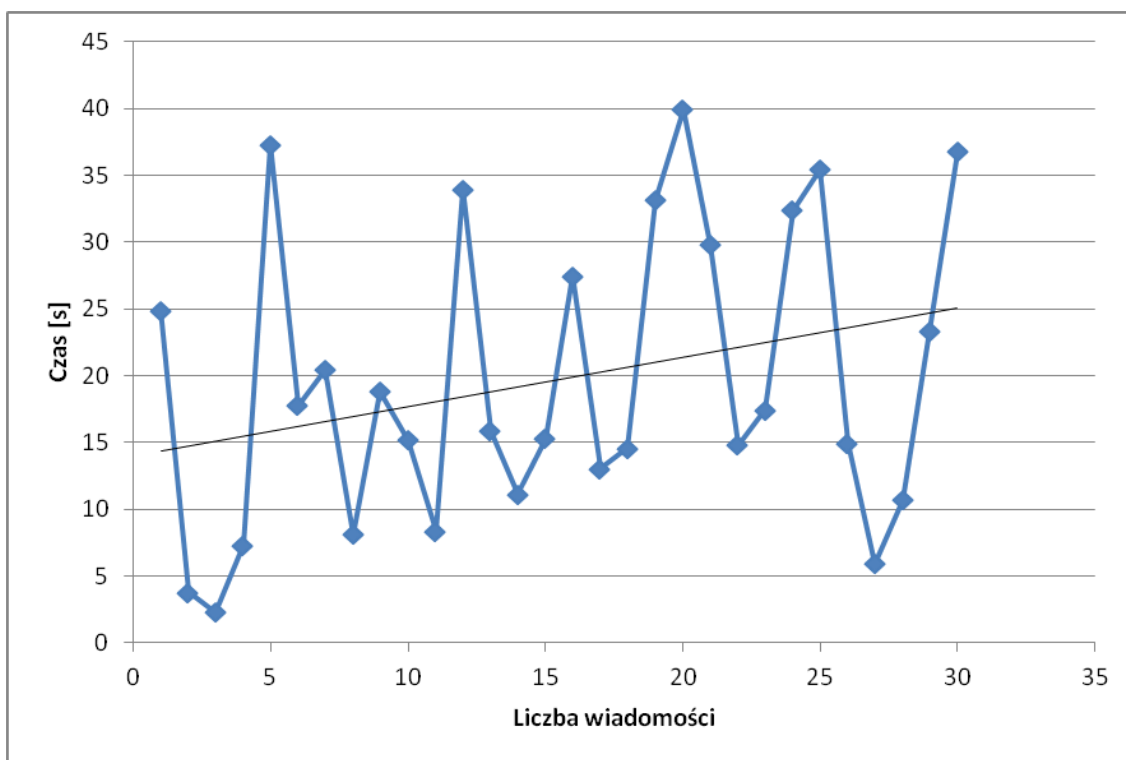
Zachowano domyślną liczbę wiadomości umieszczanych w torrencie kontrolnym — 5.

Czas przesyłania pojedynczej wiadomości wyniósł minimalnie około 2 sekund (czyli podobnie jak w przypadku punktu 5.4), a maksymalnie sięga nawet 90 sekund.

Przed wszystkim należy zwrócić uwagę na nieregularne występowanie pogorszeń wydajności (dłuższych czasów przesyłu wiadomości) — wykres 5.2. Zdarza się, że w danej iteracji w jednym teście czas jest znacznie gorszy niż w przypadku pozostałych testów. Co



**Rys. 5.2:** Czas przestania wiadomości w sieci domowej



**Rys. 5.3:** Uśredniony czas przestania wiadomości w sieci domowej

więcej, spadki wydajności wydają się być niezależne od zastosowanego algorytmu zmniejszania liczby wiadomości — nie występują jedynie w momentach, kiedy liczba torrentów jest najwyższa, czyli tuż przed uruchomieniem algorytmu zmniejszającego ich liczbę. Można również zaobserwować niespodziewane pogorszenie wydajności tuż po zmniejszeniu liczby torrentów (na przykład dla 16 wiadomości). Niemniej jednak, ten algorytm jest niezbędny, co zostanie pokazane w punkcie 5.7.

Nieregularność można po części wytłumaczyć relatywnie niską prędkością łącza sieciowego. W trakcie testów zaobserwowano błędy połączeń z trackerem lub DHT — przekroczenie maksymalnego czasu nawiązywania połączenia. Zbadano zatem liczbę pakietów wysyłanych i odbieranych przez komputery K3 i K4 w ciągu minuty w następujących sytuacjach:

- Klienci wyłączone — punkt odniesienia dla dalszych badań,
- Bezczynność — klienci są włączone, ale nie są przesyłane żadne torrenty,
- Klienci przesyłają zmienną liczbę torrentów (od 1 do 20).

Wyniki przedstawione zostały w tabeli 5.1 oraz na wykresie 5.4. Oprócz liczby połączeń w ciągu minuty („Łącznie”), w tabeli przedstawiono procentowy i ilościowy udział pakietów protokołu STUN.

**Tab. 5.1:** Statystyka połączeń

	Kl. wyłączone	Bezczynność	1	2	5	10	15	20
Łącznie	230	300	1050	1800	3840	7380	11200	17000
% STUN	0%	0%	57%	70%	76%	80%	83%	87%
STUN	0	0	598	1260	2918	5904	9296	14790

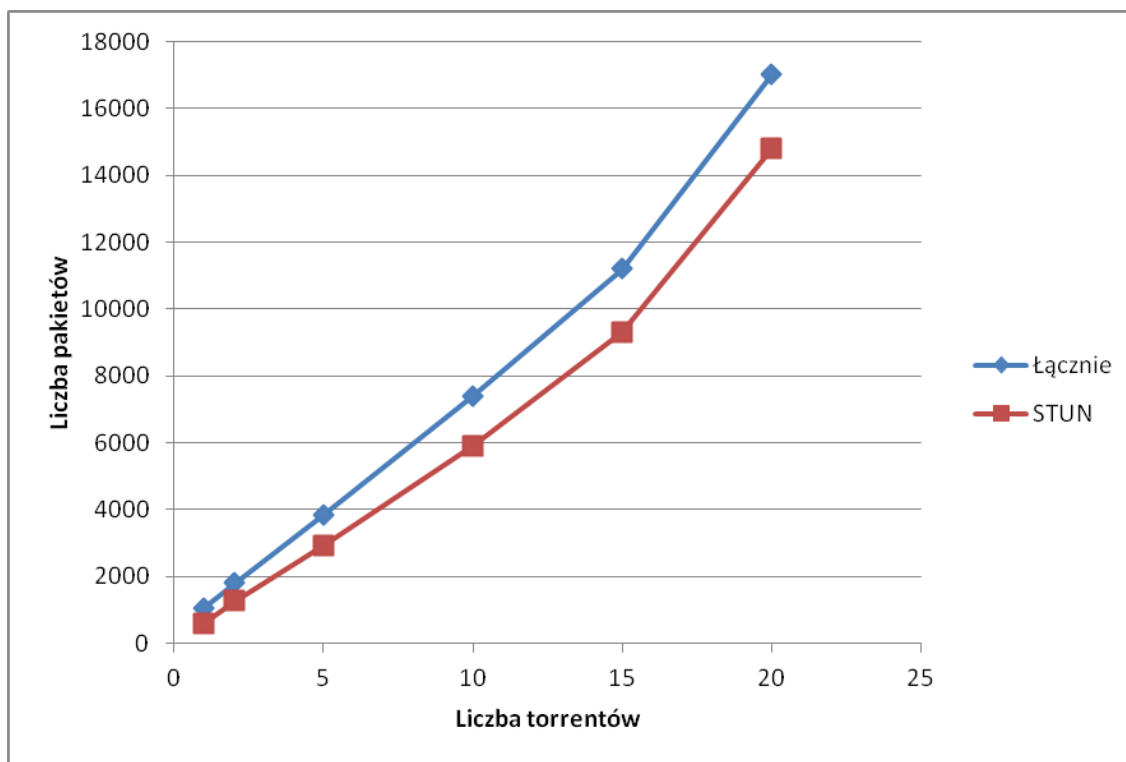
Jak wynika z przeprowadzonego doświadczenia, zdecydowaną większość wysłanych i odebranych pakietów stanowią pakiety protokołu STUN, który odpowiedzialny jest odzukiwanie adresów IP komputerów znajdujących się w sieciach stosujących translację adresów. Łączna liczba pakietów jest znacznie wyższa niż podczas próby kontrolnej z wyłączonymi klientami lub z klientami w stanie beczynności.

W sieci laboratoryjnej, pomiędzy komputerami K1, czas przesyłu wiadomości był bardziej stabilny. Dla 2-4 klientów czas przesłania wiadomości pozostawał stabilny (nieco ponad 50 sekund) niezależnie od liczby wysłanych wiadomości. Dla większej liczby klientów zaobserwowano spadki wydajności. Ta kwestia została poruszona w kolejnym punkcie (5.7).

## 5.7 Wpływ algorytmu ZLT na wydajność

W celu sprawdzenia, czy algorytm zmniejszający liczbę torrentów (algorytm ZLT, opisany w punkcie 4.2.5) jest niezbędny i spełnia swoją rolę, wykonano następujące testy:

1. Zbadanie maksymalnej liczby torrentów, które jednocześnie może obsłużyć klient. Dokonano tego przy pomocy testu polegającego na wysyłaniu wiadomości i odbieraniu odpowiedzi (scenariusz 2) z wyłączonym algorytmem ZLT — każda wiadomość



Rys. 5.4: Statystyka połączeń

stawiała się nowym torrentem. Zmieniano przy tym liczbę klientów uczestniczących w rozmowie (aktywnie i biernie).

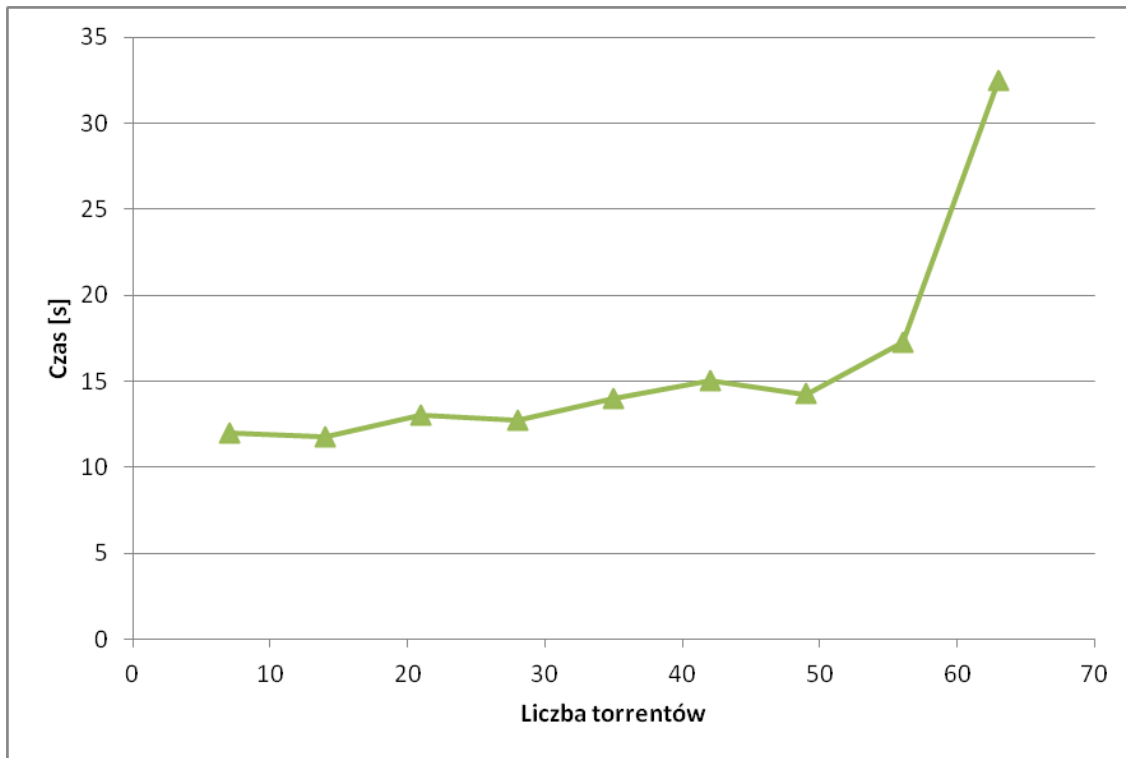
## 2. Zbadanie scenariusza 2 z włączonym algorytmem ZLT.

Testy przeprowadzono w laboratorium. W pierwszym teście uczestniczyły jedynie komputery K1 ze względu na potrzebę utrzymania spójnych parametrów maszyn. Wynik tego testu zaprezentowano na wykresie 5.5. Na osi poziomej zaznaczono liczbę torrentów, którą w danym momencie obsługiwał każdy klient, a na osi pionowej znajduje się średni czas przesyłu wiadomości pomiędzy klientami.

Dla przypomnienia: minimalny czas wysłania pojedynczej wiadomości pomiędzy komputerami K1 wynosi 10 s (punkt 5.4). Zaobserwowano niewielki wzrost czasu przesyłu wiadomości wraz ze wzrostem liczby torrentów aż do osiągnięcia liczby około 60-70 torrentów. W tym momencie na wszystkich maszynach następowało zatrzymanie pracy aplikacji. Oznacza to, że aplikacje mogą obsługiwać maksymalnie około 60-70 torrentów i nie należy przekraczać tej wartości.

W trakcie testu badano również wykorzystanie zasobów komputera. Podczas normalnej pracy aplikacji zużycie pamięci RAM oscyluje w granicach 200-400 MB, a tuż przed błędnym zatrzymaniem pracy aplikacji zużycie wynosiło ponad 600 MB.

Przeprowadzenie testu 1 potwierdziło słuszność decyzji o zaimplementowaniu algorytmu ZLT. Dodatkowym potwierdzeniem stały się wyniki drugiego testu. Zostały one przedstawione na wykresie 5.6. Na osi poziomej znajduje się liczba wiadomości wysłanych przez nadawcę, a na pionowej — czas przesyłu. Zachowano domyślną liczbę wiadomości umieszczanych w torrentcie kontrolnym — 5. W drugim teście uczestniczyły komputery



Rys. 5.5: Zależność czasu przesyłu od liczby torrentów

K1 oraz komputer K2, który był nadawcą  $U_N$  w scenariuszu 2.

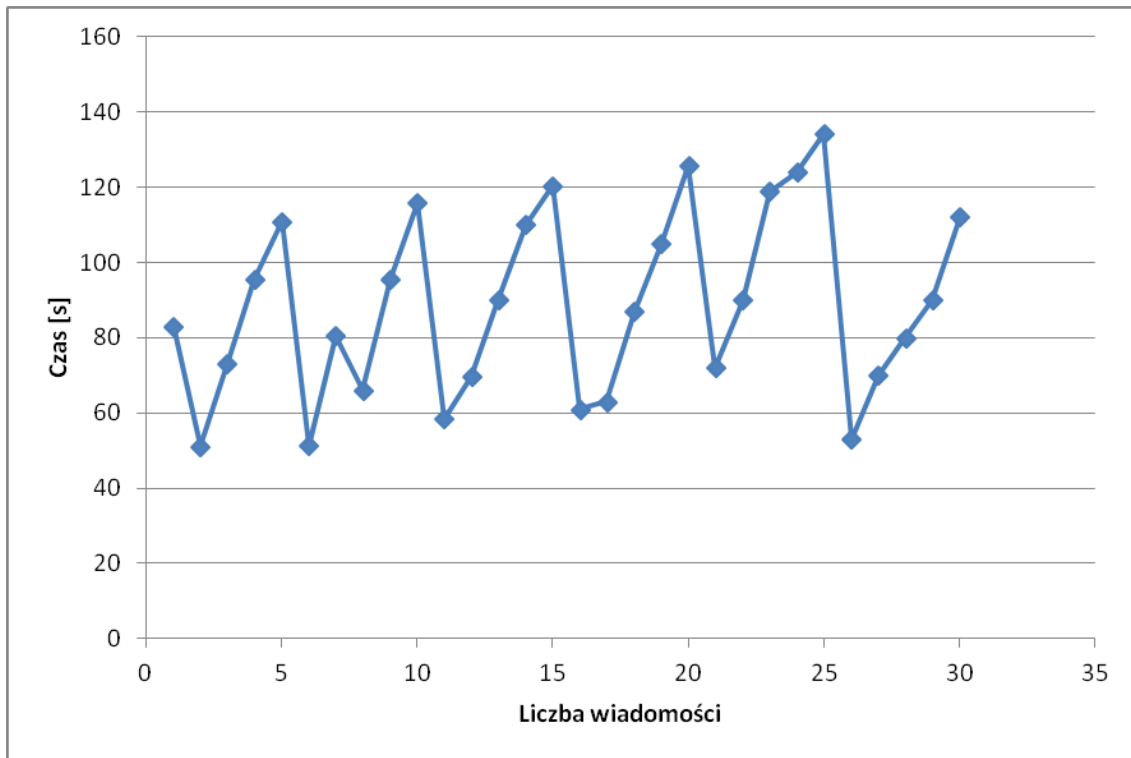
Minimalny czas przesyłu wiadomości zgadza się z wynikiem uzyskanym w teście z punktu 5.4 — 50 sekund. Wyniki na powyższym wykresie zostały uśrednione — w testach brała udział różna liczba odbiorców. Można zaobserwować spadek wydajności w miarę zbliżania do momentu, w którym algorytm zmniejsza liczbę torrentów, po czym następuje wzrost wydajności (zmniejszenie czasu przesyłu).

Należy dodać, że symulowana rozmowa nie odzwierciedla idealnie prawdziwej sytuacji, w której użytkownicy mogą nie odpowiedzieć na jakąś wiadomość, lub odpowiedzieć poprzez wysłanie większej liczby wiadomości. W tej sytuacji algorytm ZLT może zmniejszać liczbę torrentów różnym użytkownikom w różnych momentach i uzyskane czasy przesyłu wiadomości zostaną uśrednione.

## 5.8 Wpływ liczby wiadomości w algorytmie ZLT

Kolejnym kryterium, które wzięto pod uwagę podczas testów była liczba wiadomości, które algorytm ZLT umieszcza w torrencie kontrolnym (liczba ta będzie dalej nazywana **zmienną X**). Algorytm okazuje się niezbędny, ponieważ nadmierna liczba torrentów, które w danym momencie udostępnia klient, może spowodować jego błędne zatrzymanie.

Im mniejsza wartość zmiennej X, tym częściej algorytm wykonuje zamianę wielu torrentów na pojedynczy kontrolny. Skrajnym przypadkiem jest umieszczanie każdej nowej wiadomości razem ze wszystkimi poprzednimi w nowym torrencie i usunięcie poprzedniego torrenta. W takiej sytuacji liczba torrentów utrzymywanych przez każdego klienta byłaby



Rys. 5.6: Działanie algorytmu ZLT

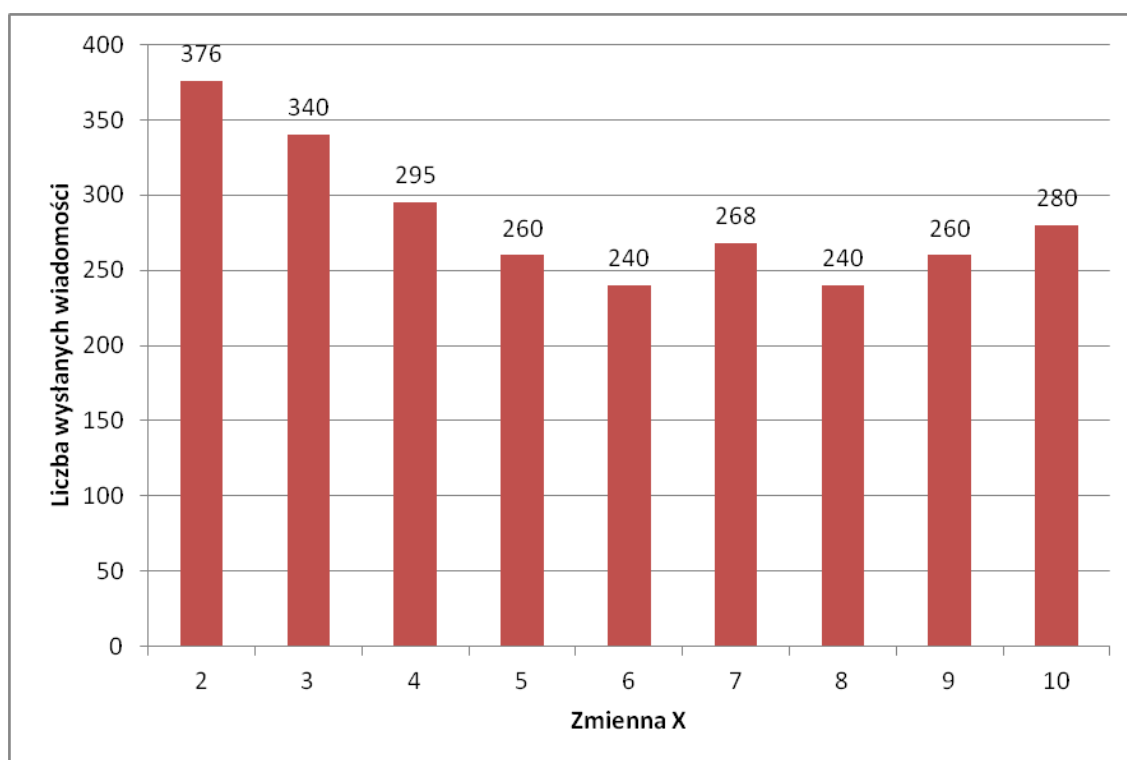
równa liczbie uczestników rozmowy. Jednakże, każda operacja zamiany wielu torrentów na pojedynczy wymaga pewnego nakładu pracy, a w opisanym przypadku koszt osiągnąłby wartość maksymalną. Dodatkowo, istnieje narzut komunikacyjny spowodowany przesyłaniem tych samych wiadomości ponownie w połączonym torrencie.

Im wyższa wartość zmiennej  $X$ , tym wyższa jest maksymalna liczba torrentów, które klient będzie musiał utrzymywać z powodu rzadziej następującej zamiany.

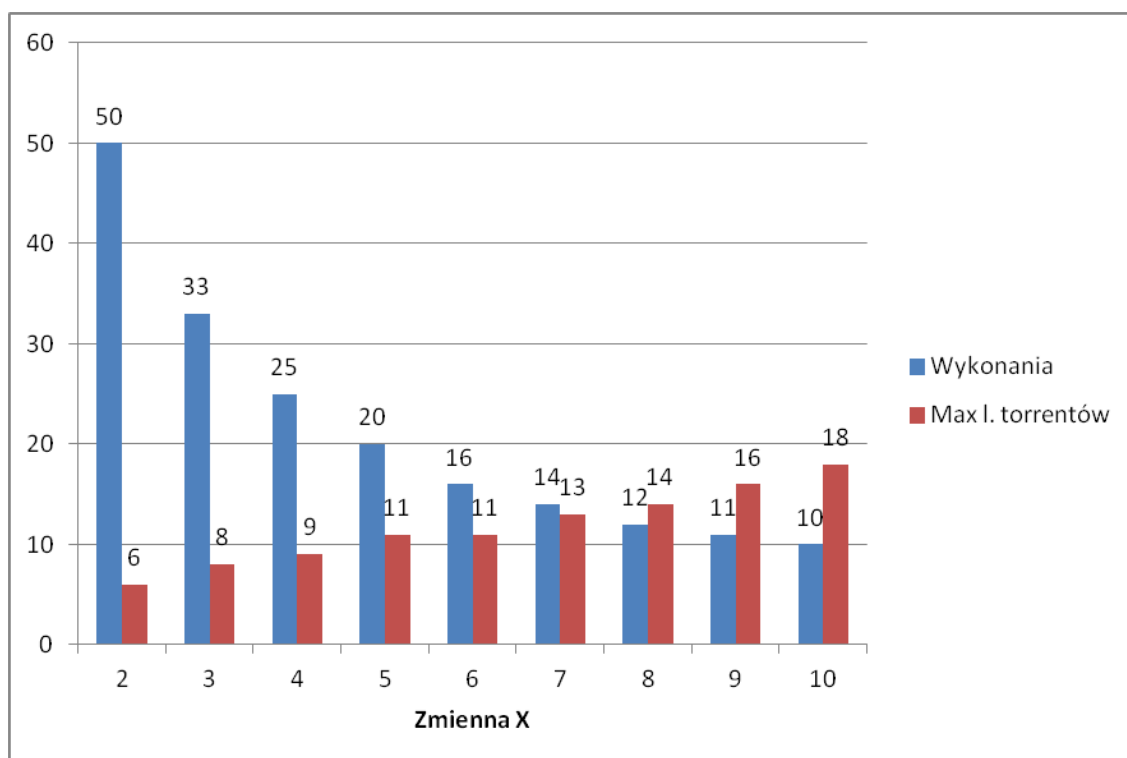
Wykresy 5.7 i 5.8 przedstawiają zależności pewnych parametrów względem różnych wartości zmiennej  $X$ . Oba wykresy bazują na 100 powtórzeniach wysłania wiadomości (100 iteracji). Pierwszy wykres przedstawia, ile faktycznie wiadomości wysłał klient — w wynikach uwzględniono też powtórzone wiadomości, które zostały wysłane wcześniej, a znalazły się ponownie w nowym torrencie kontrolnym. Drugi wykres pokazuje: ile razy algorytm ZLT stworzył nowy torrent („Wykonania”) oraz jaka jest maksymalna liczba torrentów zaobserwowana w trakcie 100 iteracji.

Z pierwszego wykresu wynika, że optymalna wartość zmiennej  $X$  wynosi 6 lub 8 — dla tych wartości zmiennej  $X$  liczba wysłanych wiadomości jest najmniejsza. Zbadano również, że wraz ze wzrostem liczby iteracji (więcej niż 100) optimum na pierwszym wykresie przesunęło się w stronę wyższej wartości  $X$ . Należy jednak pamiętać, że w teście 5.7 (a konkretnie próbie z wyłączonym algorytmem ZLT) wykazano, iż maksymalna liczba torrentów, jakie może obsługiwać aplikacja wynosi około 60-70. Konieczne jest zatem minimalizowanie liczby jednocześnie uruchomionych torrentów.

Dodatkowo, każdy użytkownik, który aktywnie udziela się w rozmowie, powoduje wzrost liczby uruchomionych torrentów, które wszystkie klienty w konwersacji muszą obsługiwać. Przedstawione na wykresie wartości dotyczą pojedynczego użytkownika. Gdyby,



**Rys. 5.7:** Liczba wysłanych wiadomości



**Rys. 5.8:** Liczba wykonań algorytmu i maksymalna liczba torrentów



przykładowo, uczestników było dwóch, maksymalna liczba torrentów podwaja się, w przypadku 3 uczestników — potraja, itd. Można więc zauważyć, że przyjmując wartość zmiennej  $X$  równą 10, możliwe jest przekroczenie 60 obsługiwanych torrentów przy 4 uczestnikach rozmowy.

## 5.9 Pobieranie listy wiadomości

Ostatni test korzysta ze scenariusza 3. Jego celem jest pomiar czasu potrzebnego na pobranie pełnej listy wiadomości przez użytkownika dołączającego do trwającej rozmowy. Istotnymi kryteriami w tym teście są: liczba wiadomości do pobrania oraz zastosowana wartość zmiennej  $X$  w algorytmie ZLT. Wyniki przeprowadzonych doświadczeń nie odbiegają od oczekiwań i rezultatów poprzednich testów.

Przed wszystkim, przy wyłączonym algorytmie ZLT konieczne byłoby pobieranie każdej wiadomości z osobna, jedna po drugiej, dlatego całkowity czas pobrania wynosiłby  $n \cdot V$ , gdzie  $n$  oznacza liczbę wiadomości do pobrania, a  $V$  — średni czas przesłania wiadomości w danym środowisku testowym. Włączenie algorytmu pozwala na znaczne skrócenie całkowitego czasu.

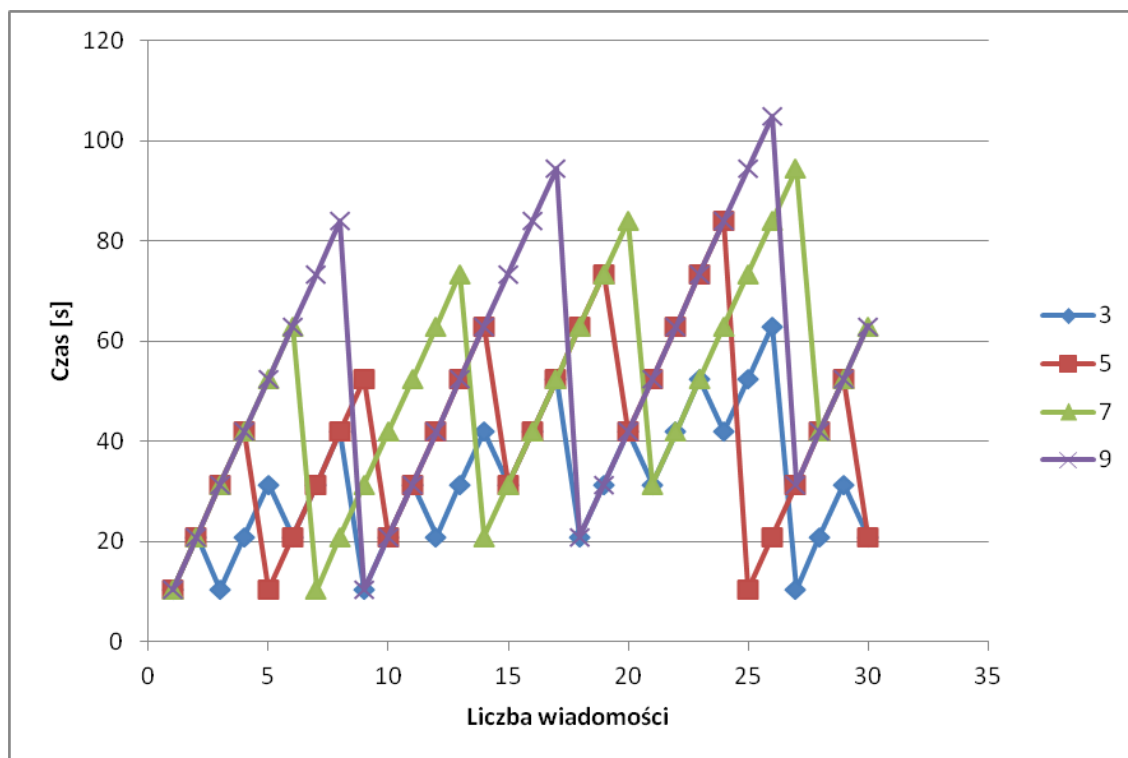
Czas ten można by jeszcze przyspieszyć, gdyby w aplikacji zaimplementowany został mechanizm przechowywania list ostatnich wartości info hash (zamiast pojedynczych wartości) w DHT oraz strukturze komunikatu (wspomniany w punkcie 4.5). W takiej sytuacji część wiadomości mogłaby być pobierana równolegle.

Bez wspomnianego mechanizmu, czasy pobierania wiadomości w sieci laboratoryjnej pomiędzy komputerami K1 są dłuższe i zostały przedstawione na wykresie 5.9. Liczba komputerów biorących udział w teście miała marginalne znaczenie. Każdy punkt oznacza czas pobrania danej liczby wiadomości w zależności od wartości zmiennej  $X$  ustawionej w algorytmie ZLT.

Można zaobserwować, że czas pobierania listy wiadomości jest tym dłuższy, im więcej komunikatów zostało wysłanych od momentu utworzenia ostatniego torrenta kontrolnego przez algorytm ZLT. Ponadto, dla wyższych wartości zmiennej  $X$  istnieje możliwość większego oddalenia się od wspomnianego torrenta. Przykładowo, dla zmiennej  $X$  równej 9 prawdopodobieństwo, że konieczne będzie pobranie więcej niż 3 wiadomości zanim pobrany zostanie torrent kontrolny wynosi 63%, natomiast dla zmiennej  $X$  równej 3 — 0%. Takie samo prawdopodobieństwo występuje na kolejnych poziomach algorytmu.

## 5.10 Wpływ jednoczesnych konwersacji na wydajność

W sieci domowej prędkości pobierania i wysyłania danych są znacznie mniejsze niż w sieci laboratoryjnej. W celu sprawdzenia, czy większa liczba konwersacji wpływa na pogorszenie wydajności, uruchomiono 4 klienty aplikacji na komputerach K3 i K4. Zaobserwowano porównywalne pogorszenie wydajności aplikacji zarówno gdy klienty prowadzą wspólną konwersację oraz gdy klienty prowadzą 2 konwersacje parami. Oznacza to, że w wolniejszych sieciach większy wpływ na wydajność ma liczba włączonych klientów niż to, w jaki



**Rys. 5.9:** Pobieranie listy wiadomości o różnej długości

sposób są one połączone (kto rozmawia z kim).

W sieci laboratoryjnej test przeprowadzono na komputerach K1. Nie zaobserwowano istotnej różnicy w czasie przesłania wiadomości w zależności od tego, czy klienci prowadzą wspólną rozmowę, czy kilka osobnych.

# Wnioski

# Bibliografia

- [1] <https://pl-pl.messenger.com/> (dostęp 09.06.2017)
- [2] [https://www.facebook.com/help/messenger-app/811527538946901?helpref=uf\\_permalink](https://www.facebook.com/help/messenger-app/811527538946901?helpref=uf_permalink) (dostęp 13.06.2017)
- [3] <http://www.bleep.pm/> (dostęp 09.06.2017)
- [4] <https://whispersystems.org/> (dostęp 09.06.2017)
- [5] <https://www.whatsapp.com/> (dostęp 27.06.2017)
- [6] <https://darkwire.io/> (dostęp 09.06.2017)
- [7] <http://moose-team.github.io/friends/> (dostęp 09.06.2017)
- [8] Becker, Georg. „Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis”. Ruhr-Universität Bochum, 2008, [http://www.emsec.rub.de/media/crypto/attachments/files/2011/04/becker\\_1.pdf](http://www.emsec.rub.de/media/crypto/attachments/files/2011/04/becker_1.pdf) (dostęp 09.06.2017)
- [9] <https://tox.chat/> (dostęp 09.06.2017)
- [10] [https://zeronet.readthedocs.io/en/latest/using\\_zeronet/sample\\_sites/](https://zeronet.readthedocs.io/en/latest/using_zeronet/sample_sites/) (dostęp 09.06.2017)
- [11] [https://bitmessage.org/wiki/Main\\_Page](https://bitmessage.org/wiki/Main_Page) (dostęp 09.06.2017)
- [12] <https://bitmessage.org/bitmessage.pdf> (dostęp 09.06.2017)
- [13] <http://www.bittorrent.org/> (dostęp 09.06.2017)
- [14] <https://webtorrent.io/> (dostęp 09.06.2017)
- [15] <https://webrtc.org/> (dostęp 27.06.2017)
- [16] <http://python-eve.org/> (dostęp 09.06.2017)
- [17] [http://www.bittorrent.org/beps/bep\\_0039.html](http://www.bittorrent.org/beps/bep_0039.html) (dostęp 28.06.2017)
- [18] [http://www.bittorrent.org/beps/bep\\_0046.html](http://www.bittorrent.org/beps/bep_0046.html) (dostęp 09.06.2017)

- [19] <https://github.com/webtorrent/webtorrent/issues/886> (dostęp 09.06.2017)
- [20] <https://github.com/jeanlauliac/kademlia-dht> (dostęp 09.06.2017)
- [21] <https://github.com/kadtools/kad> (dostęp 09.06.2017)
- [22] <https://developer.mozilla.org/en-US/docs/Web/API/Window/crypto> (dostęp 09.06.2017)
- [23] <https://github.com/seripap/darkwire.io#how-it-works> (dostęp 09.06.2017)
- [24] Trevor Perrin, Moxie Marlinspike, „The Double Ratchet Algorithm”, 2016, <https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf> (dostęp 13.06.2017)
- [25] <http://engineering.bittorrent.com/2014/12/11/authentication-and-forward-secrecy-in-bleep/> (dostęp 14.06.2017)
- [26] <http://engineering.bittorrent.com/2015/08/06/forward-secrecy-for-offline-messages-in-bleep/> (dostęp 14.06.2017)
- [27] <https://www.w3.org/TR/WebCryptoAPI/> (dostęp 09.06.2017)
- [28] [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Crypto\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API) (dostęp 09.06.2017)