POLITECHNIKA POZNAŃSKA
Wydział Informatyki
Systemy Rozproszone

Mateusz Bartkowiak

Praca magisterska

Mechanizm grupowania węzłów w systemach P2P rozszerzający protokół Chord

Promotor: dr inż. Anna Kobusińska

Poznań, Październik 2018

Spis treści

Rozdział 1

Wstęp

Niniejsza praca magisterska przedstawia protokół Statycznych Grup. Jest on rozszerzeniem protokołu Chord o mechanizm grupowania węzłów w grupy, aby zapewnić infrastrukturę dla replikacji. Wstęp ten przedstawia motywację i cel niniejszej pracy magisterskiej oraz omawia jej poszczególne rozdziały.

1.1 Motywacja i cel pracy

Systemy rozproszone to systemy które składają się z wielu niezależnych komputerów, a sprawiają na użytkownikach wrażenie jednego, logicznie zwartego systemu [21]. Wraz z rozwojem Internetu, systemy te nabierają coraz większego znaczenia, stając jedną z najważniejszych technologii, z których korzystają codziennie miliardy użytkowników [3, 4, 28, 29]. Jedną z form systemów rozproszonych są systemy Peer-to-Peer [1, 6, 21], które znajdują coraz więcej zastosowań. Powstało wiele aplikacji, korzystających z systemów Peer-to-Peer [3, 4, 33, 34, 35, 36, 38, 39, 40]. Mimo tego, autorzy w trakcie konstrukcji tego typu systemów wciąż napotykają na liczne problemy. Przykładowymi problemami [1] są: występowanie pojedynczych punktów awarii, obciążenie komunikacyjne, różnice w obciążeniu różnych części systemu, czy brak skalowalności w przypadku dołączania dużej liczby węzłów. Jednym z rozwiązań na wymienione problemy jest replikacja, gdyż dzięki utrzymywaniu kopii elementów systemu można wyeliminować pojedyncze punkty awarii, zredukować i zrównoważyć obciążenie komunikacyjne oraz poprawić skalowalność systemu poprzez zachowanie akceptowalnych czasów odpowiedzi [1]. Implementacja protokołu replikacji zapewniającym różne modele spójności jest szerokim i trudnym zagadnieniem, któremu zostało poświęcone wiele badań [1, 2, 5, 6, 9, 12, 13, 15, 18, 31, 32].

Jednym z bardziej znanych protokołów w systemach Peer-to-Peer jest protokół Chord [6]. Nie jest on jednak dostosowany, aby zaimplementować w nim replikację. Pojawiły się prace [31, 32], w których rozszerzono protokół Chord o mechanizm grupowania węzłów w grupy, dzięki któremu możliwa jest implementacja replikacji. Jednak zaproponowane mechanizmy grupowania są kosztowne, gdyż wymagają wykonywania częstych operacji międzygrupowych, które zmieniają topologię sieci. W tej pracy magisterskiej zaproponowano protokół, który rozszerza protokół Chord

o infrastrukturę grup umożliwiającą implementację mechanizmu replikacji. Jednakże różni się on od przytoczonych rozwiązań tym, iż nie stosuje operacji międzygrupowych. Protokół ten nazwano protokołem Statycznych Grup. Analiza protokołu Chord, jego istniejących rozszerzeń oraz projekt, implementacja i przetestowanie protokołu Statycznych Grup jest celem niniejszej pracy.

1.2 Omówienie rozdziałów

Niniejszą pracę rozpoczęto rozdziałem opisującym systemy typu Peer-to-Peer. Rozdział ten wskazuje na problemy i wyzwania, które pojawiają się przy konstrukcji tego typu systemów. Dalej, uzasadniono istotność replikacji w systemach rozproszonych oraz wymieniono koszty wynikające z zastosowania replikacji. Na końcu opisano czym jest sieć nakładkowa w systemach Peer-to-Peer i przedstawiono jej typy.

Następny rozdział traktuje o protokole Chord, który służy do lokalizacji obiektów w sieci. W rozdziale tym przedstawione są własności, które Chord zapewnia. Dalej opisane są założenia protokołu i sposób jego działania. Szczegóły implementacji protokołu przybliżone są przy pomocy pseudokodów. Na końcu przedstawione są możliwe rozszerzenia protokołu Chord oraz zaprezentowane są istniejące już rozwiązania.

Rozdział 4 opisuje mechanizm grupowania węzłów w grupy na przykładzie dwóch istniejących rozwiązań — Scatter oraz Rollerchain. Opisuje ich działanie, przedstawiając niektóre mechanizmy z których korzystają. Na końcu rozdział ten przedstawia przykładowe podejścia zarządzania grupami, wskazując ich wady i zalety.

Kolejny rozdział przedstawia zaproponowane rozwiązanie — protokół Statycznych Grup, którego celem jest zapewnienie infrastruktury do implementacji replikacji. Przedstawione są zalety i wady zaproponowanego protokołu. Opisane są również zagadnienia, od których protokół ten abstrahuje. Dalej następuje szczegółowy opis protokołu Statycznych Grup – przy pomocy pseudokodu przedstawione są struktury, procedury oraz funkcje wykorzystywane w protokole. Opisany jest schemat działania węzłów w przypadku szukania lokalizacji obiektu, dołączania do sieci oraz stabilizacji.

W przedostatnim rozdziale przedstawione są testy symulacyjne protokołu Statycznych Grup. Rozdział ten rozpoczyna się od przedstawienia środowiska symulacyjnego PeerSim przy pomocy przykładowego pliku konfiguracyjnego. Dalej wyszczególnione są istotne warunki (wartości parametrów oraz zachowanie się symulatora) w których przeprowadzono testy protokołu Statycznych Grup. Następnie przedstawiono wyniki testów, które badały średni rozmiar grup oraz średni czas życia grup. Każdy przypadek testowy rozpoczyna się wskazaniem wartości parametrów, po czym następują — opis wyników oraz wnioski.

Niniejszą pracę magisterską kończy podsumowanie, które opisuje wnioski i wskazuje na dalsze możliwe rozwiązania mogące poprawić działanie protokołu Statycznych Grup.

Rozdział 2

Systemy Peer-to-Peer

System Peer-to-Peer (P2P) [1, 2, 23] jest z definicji systemem w którym wszystkie węzły tworzące sieć są równoważne w sensie funkcjonalnym i pełnią rolę zarówno klienta jak i serwera. Każdy węzeł może bezpośrednio nawiązywać połączenie z innym. Brak jest centralnego serwera który np. byłby głównym magazynem danych, bądź pośredniczyłby w komunikacji. Założenia te rodzą wiele problemów i ograniczeń, dlatego też budując komercyjne systemy P2P często odchodzi się od tej definicji wyróżniając specjalne węzły, które pełnią dodatkowe funkcje.

Rozdział ten opisuje model systemu, dla którego powstał zaproponowany w niniejszej pracy magisterskiej protokół Statycznych Grup, wskazuje na pojawiające się problemy i wyzwania które należy podjąć, wskazuje istotność replikacji oraz opisuje różne budowy systemów P2P.

2.1 Problemy i wyzwania

Każdy system dużej skali stoi przed wieloma problemami. Jednym z bardziej istotnych jest problem awarii różnych części systemu. W odróżnieniu od wysoce niezawodnych systemów rozproszonych, systemy P2P często składają się z niepewnych węzłów, które nieraz pracują w niestabilnych warunkach (np. łączą się z siecią poprzez niestabilne łącze), czy dołączają do sieci tylko na pewien okres czasu [6]. W owej pracy zjawisko częstego odłączania węzłów od sieci będzie nazywane *odpływem*, natomiast zjawisko częstego dołączania węzłów do sieci *przypływem*.

Systemy Peer-to-Peer różnią się od systemów rozproszonych tym, że przypływy i odpływy są w nich częstym zjawiskiem, a więc topologia sieci, która składa się z wielu węzłów, nieustannie się zmienia. Skonstruowanie poprawnego i efektywnego systemu P2P jest zadaniem trudnym. System taki powinien spełniać szereg własności. Wybrane własności ważne z perspektywy zarządzania danymi [1] są następujące:

- Dostępność: węzły powinny w każdym momencie mieć dostęp do potrzebnych danych.
- Autonomia: węzły powinny móc dołączać do sieci i odłączać od sieci w każdej

chwili, nie tworząc przy tym dodatkowych problemów, takich jak ograniczanie dostępności do danych.

- Wydajność: system powinien efektywnie wykorzystywać dostępne zasoby sieci (np. przepustowość, czy moc obliczeniową).
- Jakość usług: z perspektywy użytkownika system powinien być wysokiej jakości, tzn. wyniki winne być kompletne, dane spójne i zawsze dostępne, a czas odpowiedzi powinien być krótki.
- Odporność na awarie: ewentualne awarie, które się zdarzają, nie powinny mieć wpływu na wydajność, czy jakość usługi.
- Bezpieczeństwo: naturalna otwartość systemów P2P sprawia, że bezpieczeństwo, głównie w kontekście nieuprawnionego dostępu do danych, jest dużym wyzwaniem.

2.2 Replikacja

Jednym z rozwiązań na problemy z dostępnością, jakością usług, czy odpornością na awarie powstałe w wyniku nieuniknionych odpływów jest replikacja [1]. Po pierwsze, replikacja zwiększa dostępność poprzez eliminację pojedynczych punktów awarii (obiekty są dostępne z różnych węzłów). Po drugie, poprawia wydajność systemu poprzez redukcję obciążenia komunikacyjnego w sieci (obiekty mogą być umiejscowione bliżej węzłów pytających oraz są udostępniane jednocześnie z wielu węzłów). Ponadto replikacja poprawia skalowalność systemu, ponieważ przy wzroście liczby węzłów, możliwe jest zachowanie akceptowalnych czasów odpowiedzi.

Minusem replikacji jest konieczność zużywania większej ilości pamięci, aby przechowywać kopie obiektów. Innym minusem replikacji jest konieczność uspójniania kopii obiektów, czyli ich aktualizowanie, aby były takie same. Rodzi to dodatkowy narzut komunikacyjny. Replikacja jest trudnym zagadnieniem, gdyż koszty replikowania danych rosną wraz z liczbą utrzymywanych kopii [32]. Próby zminimalizowania kosztów w jednym obszarze, najczęściej kończą się podwyższeniem kosztów w innym. Poniżej przedstawione są trzy różne metryki kosztów związane z replikacją [6]:

- Koszty monitoringu: potrzebne są mechanizmy monitoringu istniejących replik, aby mieć informację, czy są dostępne, oraz wychwytywanie nowych replik, aby móc zmienić lokalizację danych w przypadku awarii.
- Koszty przesyłu danych: związane z tworzeniem nowych replik w systemie.
- Koszty nierównomiernego obciążenia: niektóre mechanizmy replikacji mogą skutkować w nierównomiernej dystrybucji danych pomiędzy węzłami. Ma to znaczący wpływ na działanie systemu, ponieważ niektóre węzły mogą być przeciążone, podczas gdy inne będą nieużywane.

Jak zostało już nadmienione, poprawienie wszystkich powyższych metryk może okazać się niemożliwe. Dla przykładu, istnieją takie mechanizmy replikacji, które preferują replikowanie danych na węzłach bardziej stabilnych [6]. Pozwala to zaoszczędzić czas przesyłu danych (mniejsza częstotliwość tworzenia nowych replik), ale odbywa się to kosztem nierównomiernego obciążenia (bardziej niezawodne węzły będą bardziej obciążone). Ponadto twórcy konkretnych rozwiązań często badają swoje algorytmy pod kątem wybranych metryk, pomijając inne [6].

W protokole Statycznych Grup intensywnie wykorzystywany jest mechanizm replikacji, aczkolwiek protokół ten abstrahuje od konkretnego sposobu replikowania, pozostawiając swobodę w tej kwestii.

2.3 Sieć nakładkowa

Systemy P2P wykorzystują do działania sieć nakładkową (ang. *overlay network*), która jest dodatkową warstwą abstrakcji nadbudowaną nad istniejącą siecią (np. Internet) [1, 21]. Parametry niefunkcjonalne systemu P2P takie jak odporność na awarie, autonomia węzłów, wydajność, skalowalność czy bezpieczeństwo zależą w dużej mierze od budowy sieci nakładkowej, którą można podzielić na trzy główne typy [1]: nieustrukturyzowane, ustrukturyzowane oraz sieci z tzw. super-węzłami (ang. *super-peer network*).

2.3.1 Sieci nieustrukturyzowane

W sieciach nieustrukturyzowanych, sieć nakładkowa jest budowana ad hoc w sposób niedeterministyczny. Rozmieszczenie danych jest zupełnie niezależne od topologii sieci nakładkowej, a każdy węzeł wie jedynie o swoich sąsiadach, aczkolwiek nie jest poinformowany o zasobach jakie posiadają. Mechanizmy wyszukiwania są z reguły proste, acz kosztowne. Przykładem może być mechanizm zalewania sieci zapytaniami o dany zasób, które krążą po sieci dopóki żądany zasób nie zostanie odnaleziony. Innym, nieco bardziej wyrafinowanym i wydajnym sposobem jest przesyłanie kilku równoległych zapytań, które każde jest przesyłane między węzłami w taki sposób, że jeden węzeł przesyła dalej zapytanie tylko do swojego jednego sąsiada. Z tego typu sieci nakładkowej korzystają m.in. takie systemy P2P jak Gnutella [33], Kazaa [34], czy FreeHaven [35]. Jednym z największych wad sieci nieustrukturalizowanych jest niska skalowalności, gdyż przy coraz to większej ilości węzłów w sieci, znalezienie danego zasobu jest coraz bardziej kosztowne. Odpowiedzią na ten problem jest sieć ustrukturalizowana.

2.3.2 Sieci ustrukturyzowane

W sieciach ustrukturalizowanych, lokalizacja zasobów w sieci jest deterministyczna. Każdy zasób posiada swój identyfikator, który jednocześnie wskazuje miejsce w sieci, w którym się znajduje. W sieciach tych często korzysta się z rozproszonych tablic mieszających (ang. *distributed hash tables*, w skrócie DHT), które udo-

stępniają interfejs do przechowywania i pobierania danych. Każdy obiekt danych posiada swój klucz, który jest jego identyfikatorem i jest zależny od wartości tego obiektu. Każdy węzeł odpowiedzialny jest za przechowywanie tych danych, których wartość kluczy mieści się w odpowiednim dla niego, ściśle określonym zakresie. Ponadto każdy węzeł ma informacje o pewnej liczbie innych węzłów w sieci (sąsiadów): przechowuje specjalne tablice trasowania w której przypisane są identyfikatory sąsiadów do odpowiednich adresów. Większość operacji dostępu do danych to operacje *lookup*, czyli próby znalezienia lokalizacji jakiegoś obiektu. Operacja lookup najczęściej wykorzystywana jest do znalezienia adresu węzła odpowiedzialnego za dany zasób, dzięki czemu węzeł pytający może bezpośrednio nawiązać z nim komunikację. Aby skutecznie odnaleźć węzeł odpowiedzialny, pojedyncze wywołanie tej operacji może skutkować kilkoma przeskokami żądania pomiędzy sąsiadami. Ponieważ każdy węzeł jest odpowiedzialny za pewien zakres kluczy oraz współtworzy system trasowania, autonomia pojedynczego węzła jest mocno ograniczona. Jest to największą wadą sieci ustrukturalizowanych, gdyż każde dołączenie, bądź odłączenie węzła zaburza strukturę sieci, co skutkuje potrzebą przeprowadzenia jej rekonstrukcji. Przykładami tego typu sieci są takie systemy jak Chord (więcej o protokole Chord w rozdziale 3), Pastry [36], czy Pier [38]. Zaproponowany w tej pracy protokół Statycznych Grup opiera się na rozszerzeniu protokołu Chord, a więc jest przykładem sieci ustrukturyzowanej.

2.3.3 Sieci z super-węzłem

W sieciach ustrukturalizowanych oraz nieustrukturalizowanych wszystkie węzły są takie same pod względem funkcjonalności. Inaczej się ma sprawa z sieciami super-peer (sieci z super-węzłami), które stanowią klasę pośrednią, między modelem P2P a modelem klient-serwer. W sieciach tej klasy niektóre węzły pełnią specjalne role w sieci, spełniając takie dodatkowe funkcje jak indeksowanie, przetwarzanie zapytań, kontrola dostępu, czy zarządzanie metadanymi. W przypadku awarii, rolę super-węzłów mogą dynamicznie przejmować inne węzły. Działanie tych sieci przeważnie polega na wysyłaniu zapytań do super-węzła, który posiada informacje o położeniu żądanych obiektów. Dlatego też zaletami sieci super-peer są wydajność oraz jakość usługi. Czas potrzebny na znalezienie żądanego obiektu jest zazwyczaj krótszy, niż w przypadku zalewania sieci. Ponadto heterogeniczność węzłów w sensie ich zasobów i wydajności można wykorzystać do wyróżnionych zadań, obierając je za super-węzły. Jednakże autonomia węzłów w owych sieciach jest ograniczona, a odporność na awarie jest niska, gdyż super-węzły stają się pojedynczymi punktami awarii (skutki tego problemu można załagodzić dynamicznie wybierając nowe super-węzły). Z tego typu sieci korzystają m.in. takie systemy jak Edutella [39], czy JXTA [40].

Rozdział 3

Protokół Chord

Fundamentalnym problemem z którym należy się zmierzyć tworząc system P2P jest efektywne znalezienie węzła odpowiedzialnego za dany zasób. Jednym z bardziej znanych protokołów, które rozwiązują ten problem jest Chord, na którym opiera się zaproponowany w niniejszej pracy magisterskiej protokół Statycznych Grup. Chord dostarcza operację wyznaczenia węzła na podstawie klucza. Dzięki temu można odnaleźć żądany zasób (który jest reprezentowany przez odpowiedni klucz). Chord dostosowywuje się do dynamicznego dołączania oraz odłączania węzłów z systemu, i nawet podczas tych zmian systemu działa w prawidłowy sposób, zachowując przy tym dobrą skalowalność [23]. Tak jak wspomniano w paragrafie 2.3.1, Chord jest przykładem sieci ustrukturyzowanej korzystającej z DHT.

3.1 Opis

Chord dostarcza operację *lookup*, która mapuje klucz na węzeł. Upraszcza konstrukcję systemu P2P i aplikacji bazującej na nim zapewniając następujące własności [23]:

- Równowaga obciążenia: Chord przydziela klucze węzłom w sposób równomierny.
- Decentralizacja: Chord działa w rozproszeniu i żaden węzeł nie jest ważniejszy od innego.
- Skalowalność: koszt operacji lookup rośnie logarytmicznie wraz z wzrostem liczby węzłów, dlatego też działa sprawnie nawet w systemach z relatywnie dużą liczebnością węzłów.
- Dostępność: Chord automatycznie dostraja się do ciągłych zmian w sieci, zapewniając, że węzeł odpowiedzialny za dany klucz może w każdej chwili zostać odnaleziony.
- Elastyczne nazewnictwo: Chord nie wymusza struktury nazewniczej kluczy — przestrzeń kluczy jest płaska. Aplikacja korzystająca z tego protokołu ma dużą swobodę w sposobie mapowania własnych nazw na klucze.

W kontekście budowania aplikacji, o Chordzie można myśleć jak o bibliotece, która dostarcza tej aplikacji operację `lookup(key)`, oraz powiadamia ją na danym węźle o zmianie zbioru kluczy, za które ten węzeł jest odpowiedzialny. Dzięki temu, aplikacja może podjąć stosowne działania, jak np. przenieść odpowiednie wartości do nowo dołączonego węzła. Aplikacja używająca Chorda jest odpowiedzialna za ewentualną autoryzację, używanie pamięci podręcznej, replikację, czy przyjazne dla użytkownika nazewnictwo danych. Płaska przestrzeń adresowa ułatwia implementację tych funkcjonalności. Dla przykładu, aplikacja mogłaby autentykować dane, przechowując je pod kluczem pochodzącym z mechanizmów kryptograficznych. Tak samo, aplikacja mogłaby replikować dane poprzez przechowywanie ich na miejscach związanych z kluczami wywiedzionymi z identyfikatora tych danych z poziomu aplikacji.

3.2 Protokół

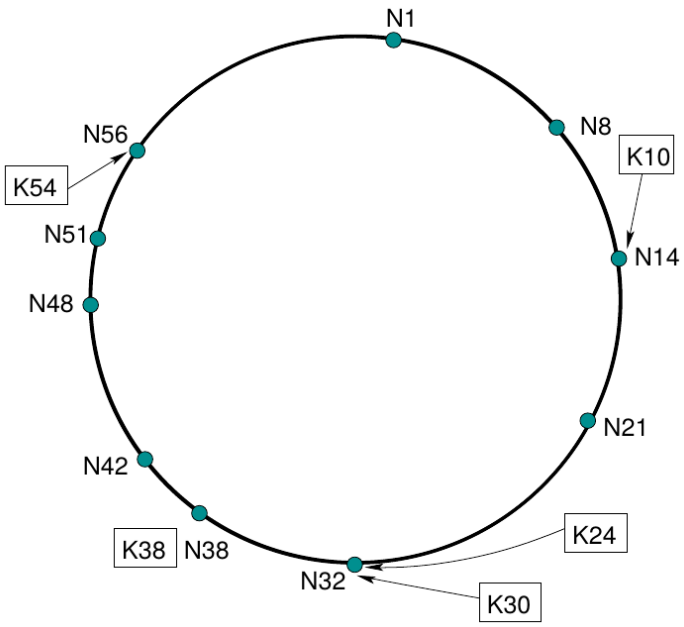
Protokół Chord określa jak odnaleźć lokalizację kluczy, jak nowe węzły są dołączane do systemu oraz jak poradzić sobie z odłączaniem węzłów (planowanym bądź nie).

3.2.1 Identyfikatory

Funkcja spójnego mieszania (ang. *consistent hash function*) przypisuje każdemu węzłowi oraz kluczowi m-bitowy identyfikator. Identyfikator węzła generowany jest z jego adresu IP, a identyfikator klucza bezpośrednio z wartości. Długość identyfikatora m musi być odpowiednio duża, aby zapewnić jak największą odporność na kolizje.

Identyfikatory są ułożone kolejno według wartości na *okręgu identyfikatorów*. Okrąg ten, z numerami od 0 do $2^m - 1$, nazywany będzie w tej pracy *pierścieniem*. Klucze są przypisane temu węzłowi, którego identyfikator znajduje się na okręgu jako pierwszy po danym kluczu k, lub jest jemu równy. Ten węzeł nazywa się *następnikiem* klucza k i oznaczany jest jako `successor(k)`. Przykładowy pierścień pokazany jest na rysunku 3.1.

Utrzymywanie pierścienia odbywa się w następujący sposób: gdy węzeł n dołączy do sieci, odpowiednie klucze, poprzednio przypisane następnikowi węzła n, zostają przypisane do n. Kiedy węzeł n opuści sieć, wszystkie klucze za które był odpowiedzialny są przypisane na nowo do następnika n. Dla przykładu, jeśli do sieci reprezentowanej przez pierścień na rysunku 3.1 dołączyłby węzeł o identyfikatorze 26, to klucz 24 zostałby przeniesiony z węzła 32 do nowo dołączonego węzła.



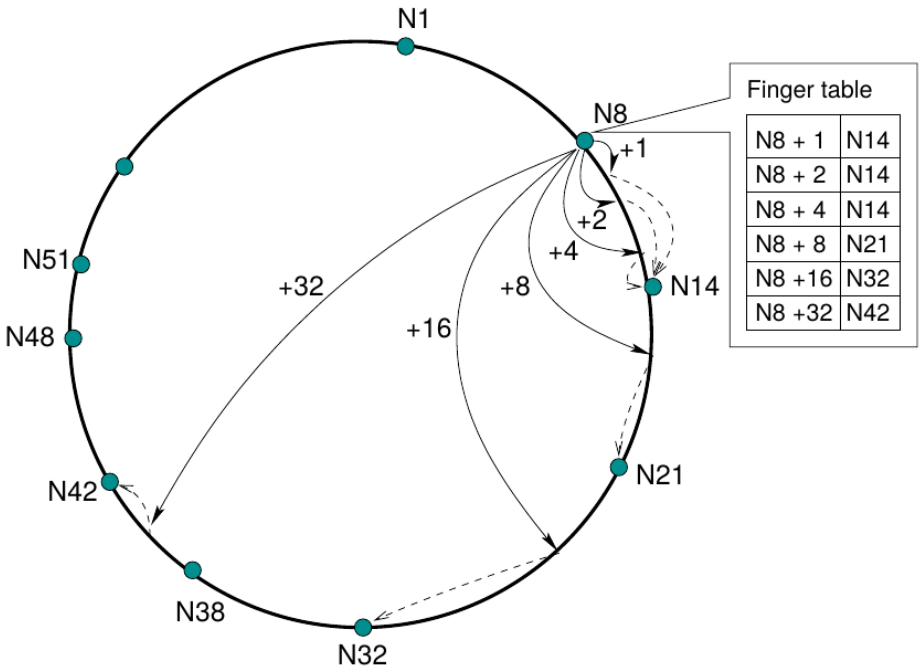
Rysunek 3.1: Okrąg identyfikatorów (pierścień) zawierający dziesięć węzłów, które przechowują w sumie pięć kluczy.

3.2.2 Odnajdywanie lokalizacji klucza

Każdy węzeł posiada wskaźnik na swojego następnika. Dzięki temu w prosty sposób możliwe jest odnalezienie węzła odpowiedzialnego za dany klucz poprzez rekurencyjne odpytywanie się kolejnych następników. Posiadanie aktualnego wskaźnika na swojego następnika jest warunkiem koniecznym otrzymania poprawnego wyniku. Aczkolwiek aby zmniejszyć liczbę komunikatów podczas operacji lookup, każdy węzeł posiada dodatkowe informacje trasowania. Informacje te przechowywanie są w specjalnej tablicy trasowania, którą autorzy nazywają *finger table* [23]. Niech m będzie liczbą bitów używanych do zapisywania identyfikatorów kluczy i węzłów. Wtenczas tablica trasowania posiada m wpisów. Każdy i-ty wpis tej tablicy węzła n posiada informacje o pierwszym węźle s (jego identyfikator oraz adres, który służy do komunikacji — jest nim najczęściej adres IP), który jest za n o przynajmniej 2^{i-1} na okręgu identyfikatorów. Wzór na identyfikator węzła s w i-tym wpisie węzła n, gdzie $1 \leq i \leq m$:

$$s = successor(n + 2^{i-1}) \tag{3.1}$$

Warto zwrócić uwagę, że pierwszy wpis w tablicy trasowania węzła n jest jego bezpośrednim następnikiem na pierścieniu. Na rysunku 3.2 pokazane są wpisy w tablicy trasowania węzła 8. Pierwszy wpis wskazuje na węzeł 14, gdyż jest on pierwszym węzłem który jest przed wartością $(8 + 2^0) \bmod 2^6 = 9$. Podobnie, ostatni wpis wskazuje na węzeł 42, ponieważ jest on pierwszym węzłem, który jest przed wartością $(8 + 2^5) \bmod 2^6 = 40$.



Rysunek 3.2: Wpisy tablicy trasowania węzła 8.

Schemat ten posiada dwie cechy. Po pierwsze, każdy węzeł posiada informację o małej liczbie węzłów, z czego większość tych węzłów znajduje się w małej odległości od niego na pierścieniu. Po drugie, korzystając z tablicy trasowania z reguły nie można bezpośrednio określić następnika dowolnego klucza. Dla przykładu, węzeł 8 na rysunku 3.2 nie może samodzielnie ustalić bezpośredniego następnika klucza 34, gdyż następnik ten (węzeł 38) nie znajduje się w tablicy trasowania węzła 8.

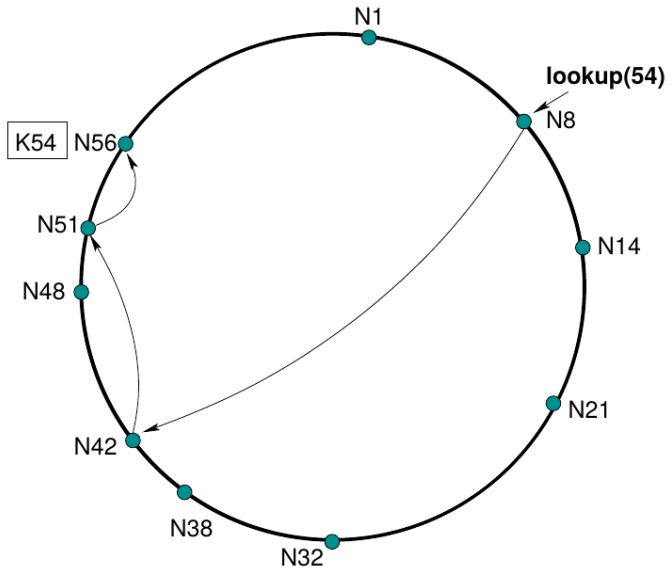
Listing 3.1 pokazuje pseudokod funkcji `find_successor`, która jest odpowiednikiem operacji lookup, czyli odnalezienia węzła odpowiedzialnego za dany klucz. Jeżeli identyfikator `id` znajduje się pomiędzy identyfikatorem aktualnego węzła `n` a identyfikatorem następnika `successor` tego węzła, to funkcja ta zwraca wskaźnik na następnik węzła `n`. W przeciwnym wypadku, wykorzystując funkcję `closest_preceding_node` pokazaną na listingu 3.2, węzeł `n` przeszukuje tablicę trasowania, aby znaleźć taki węzeł `n'`, którego identyfikator jest najbliższy lecz mniejszy od `id`; i następnie wywołuje zdalną metodę `find_succesor` węzła `n'`. Wybieranie węzła `n'` odbywa się w taki sposób, ponieważ im węzeł znajduje się bliżej `id` na pierścieniu, tym posiada więcej informacji o obszarze pierścienia wokół `id`.

```
1  n.find_successor(id):
2      if id ∈ (n, succesor>
3          return successor
4      else
5          n' = closest_preceding_node(id)
6          return n'.find_successor(id)
```

Listing 3.1: Pseudokod funkcji odpowiedzialnej za znalezienie następnika obiektu o danym identyfikatorze.

```
1  n.closest_preceding_node(id):
2      for i = m downto 1
3          if finger[i] ∈ (n, id)
4              return finger[i]
5      return n
```

Listing 3.2: Pseudokod funkcji odpowiedzialnej za znalezienie najbliższego poprzednika obiektu o danym identyfikatorze z tablicy trasowania.



Rysunek 3.3: Ścieżka zapytania o klucz 54 rozpoczynającego się od operacji lookup węzła 8.

Przykładowo węzeł 8 chciałby znaleźć następnika klucza 54. Sytuacja ta pokazana jest na rysunku 3.3. Ponieważ ostatnim wpisem w tablicy trasowania węzła 8, który poprzedza 54, jest węzeł 42, to węzeł 8 wyśle zapytanie do węzła 42. Dalej, węzeł 42 przeszuka swoją tablicę trasowania, aby odnaleźć największy węzeł, który poprzedza klucz 54, i będzie to węzeł 51. Wyśle więc do niego zapytanie, a węzeł 51 odpowie, że to jego następnik, węzeł 56, jest następnikiem klucza 54. Ostatecznie odpowiedź ta wróci do węzła 8.

Twórcy protokołu Chord udowadniają [23], iż złożoność komunikacyjna operacji lookup, w sieci składającej się z N węzłów, z dużym prawdopodobieństwem wynosi $\mathcal{O}(\log N)$. A średni czas odnalezienia węzła odpowiedzialnego za dany klucz podczas eksperymentów wyniósł $\frac{1}{2}\log N$.

3.2.3 Zmiany w sieci

W praktyce, Chord musi radzić sobie z ciągłym dołączaniem węzłów oraz ich nieprzewidywalnym odłączaniem. Aby operacja lookup zawsze zwracała poprawny wynik mimo zmian w sieci, protokół Chord musi zapewnić, aby w każdym węźle wskaźniki na następnika były aktualne. Dbają o to specjalne procedury wykonywane cyklicznie przez wszystkie węzły, które uaktualniają tablice trasowania oraz wskaźniki na następnika i poprzednika.

Pseudokod procedury odpowiedzialnej za dołączanie węzła pokazuje listing 3.4. Węzeł `n` dołącza się do istniejącego pierścienia przy pomocy innego węzła `n'`, który tworzy owy pierścień. Jeżeli jednak węzeł `n` ma zamiar stworzyć nowy pierścień, wywołuje procedurę `create` pokazaną na listingu 3.3. Samo dołączenie do sieci nie sprawia, iż reszta węzłów w sieci wie o nowym węźle. Dopiero procedura stabilizacji `stabilize`, pokazana na listingu 3.5 i uruchamiana cyklicznie przez każdy węzeł, uaktualnia informacje o węzłach w sieci. Za każdym razem, gdy węzeł `n` uruchomi stabilizację, sprawdzane jest czy to poprzednik `p` następnika `successor` węzła `n` nie powinien być jego następnikiem. Będzie tak, jeśli `p` jest nowo dołączonym węzłem. Dodatkowo, wywołując procedurę `notify` pokazaną na listingu 3.6, stabilizacja powiadamia następnika węzła `n` o istnieniu węzła `n` w sieci, dając szansę na uaktualnienie wskaźnika na poprzednika.

```
1 n.create():
2     predecessor = null
3     successor = n
```

Listing 3.3: Pseudokod procedury odpowiedzialnej za utworzenie nowego pierścienia Chord.

```
1 n.join(n'):
2     predecessor = null
3     successor = n'.find_successor(n)
```

Listing 3.4: Pseudokod procedury odpowiedzialnej dołączenie węzła do pierścienia Chord zawierającego węzeł `n'`.

Każdy węzeł okresowo uruchamia procedurę `fix_fingers` pokazaną na listingu 3.7, która uaktualnia wpisy w tablicy trasowania. Dzięki temu nowe węzły wypełniają swoje tablice, a węzły będące wcześniej w sieci, aktualizują je o ewentualne nowe węzły. Ponadto każdy węzeł wykonuje cyklicznie procedurę

```
1 n.stabilize():
2     p = successor.predecessor
3     if p ∈ (n, successor)
4         successor = p
5     successor.notify(n)
```

Listing 3.5: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za weryfikację bezpośredniego następnika `successor` węzła `n` oraz powiadomienie go o `n`.

```
1 n.notify(n'):
2     if predecessor is null or n' ∈ (predecessor, n)
3         predecessor = n'
```

Listing 3.6: Pseudokod procedury odpowiedzialnej za ewentualną aktualizację wskaźnika na poprzednika `predecessor`.

```
1 n.fix_fingers():
2     next = next + 1
3     if next > m
4         next = 1
5     finger[next] = find_successor(n + 2^(next−1))
```

Listing 3.7: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za aktualizację wpisów w tablicy trasowania.

```
1 n.check_predecessor():
2     if predecessor has failed
3         predecessor = null
```

Listing 3.8: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za wyzerowanie wskaźnika na poprzednika `predecessor` w przypadku jego awarii.

`check_predecessor` pokazaną na listingu 3.8, która zeruje wskaźnik na poprzednika, jeśli uległ on awarii. Wskaźniki te uaktualniane są podczas cyklicznej operacji `notify`.

W praktyce, przez ciągłe przypływy oraz odpływy węzłów, stan pierścienia Chord nigdy nie będzie stabilny. Dołączenia, odłączenia i cykliczny algorytm stabilizacji będą się przeplatały bez końca. Pierścień nie będzie miał czasu, aby ustabilizować się w pełni, nim nastąpi zmiana w zbiorze węzłów sieci. Jednakże autorzy protokołu zapewniają, że jeśli protokół stabilizacji będzie uruchamiany z odpowiednią częstotliwością, to operacje lookup będą wykonywać się sprawnie, zwracając poprawne wyniki [23].

3.3 Istniejące rozszerzenia

Protokół Chord jest prosty w swojej idei. Jego konstrukcja pozostawia duże pole do ewentualnych rozszerzeń. Sami autorzy zaproponowali kilka prostych usprawnień. Ponadto powstało wiele pomysłów jak rozszerzyć Chorda o dodatkowe funkcjonalności, takie jak replikacja danych [6]. Są też rozwiązania, które korzystają z Chorda, dostosowywując go do swoich potrzeb [30, 32]. Sam motyw przewodni tej pracy bazuje na rozszerzeniu protokołu Chordzie o dodatkowy mechanizm grupowania.

3.3.1 Usprawnienie działania

Aby poprawić działanie protokołu Chord, autorzy proponują dwa rozszerzenia. Po pierwsze, zamiast wskaźnika na następnika, każdy węzeł może utrzymywać listę następników. Dzięki temu mógłby odwołać się do innego następnika, jeżeli bliższy następnik uległ awarii. Poprawia to działanie protokołu, gdyż uodparnia na odpływy, zachowując wydajność operacji lookup na wysokim poziomie. Drugą rekomendacją autorów jest zaimplementowanie procedury odłączenia od sieci. W bazowej implementacji odejście węzła polega na jego nagłym odłączeniu, bez informowania sieci i traktuje się je tak jak awarie. Zamiast tego, węzeł mógłby poinformować swojego następnika oraz poprzednika, że opuszcza sieć, dzięki czemu mogłyby one uaktualnić wskaźniki. Ponadto węzeł mógłby przed opuszczeniem przetransferować klucze za które jest odpowiedzialny, swojemu następnikowi. Zamiany ta zwiększyłyby dostępność systemu, oraz poprawiły jego wydajność.

3.3.2 Replikacja

Jak zostało wspomniane w podrozdziale 2.2, replikacja danych zwiększa dostępność systemu. W protokole Chord nie występuje natywnie owy mechanizm, dlatego też podczas awarii węzła, wszystkie klucze, za które odpowiadał, zostają utracone, stając się niedostępnymi.

Jeżeli system oparty na Chordzie ma używać replikacji, to jednym z prosztych rozwiązań wydaje się być replikacja u sąsiadów, jak np. w *Neighbor Replication* [6]. W podejściu tym, każdy węzeł dba, aby dane replikować u swoich sąsiadów. Mechanizm ten jest elastyczny, gdyż pozwala indywidualnie (osobno przez każdy węzeł) kontrolować stopień replikacji, dzięki czemu można dobierać odpowiednie polityki według potrzeb; np. replikować popularne dane[1] na bardziej stabilnych węzłach, bądź na większej ich ilości. Zaletą tego podejścia są niskie koszty monitoringu. Wadą natomiast wysokie koszty utrzymywania replik.

Innym podejściem jest *Multi-Publication* [6]. Rozwiązanie te opiera się na replikowaniu danych na wielu, deterministycznie wyznaczonych pozycjach w pierścieniu Chord. Sposób wyznaczania pozycji może być dowolny, aczkolwiek jest stały dla całej sieci, a więc o wiele mniej elastyczny, niż replikacja u sąsiadów. Sugerowane

[1] Dane popularne to te, o które relatywnie często przychodzą żądania.

jest, aby obiekt był przechowywany pod wieloma kluczami, które wyznaczane są kilkoma funkcjami skrótu. Problemem są jednak odpływy, które trzeba monitorować, aby zachować ustalony stopień replikacji. Natomiast plusem może być dobre zrównoważenie obciążenia (w zależności od doboru metody wyznaczania pozycji).

3.3.3 Inne

Istnieje też wiele innych rozwiązań mających u swego rdzenia mechanizm Chord. Dobrym przykładem jest algorytm *Topology Aware Chord* [30], rozszerzający Chorda o mechanizmy kontrolujące topologię sieci nakładkowej w zależności od fizycznej sieci w której działa. Autorzy twierdzą, że dzięki temu uzyskali lepszą wydajność pod kątem trasowania wiadomości oraz zużycia pasma.

Chord został tak zaprojektowany, aby rozszerzanie go nie stanowiło problemu. Dlatego powstało wiele pomysłów jak go zmienić, aby poprawić różne parametry systemu. Sam Protokół Statycznych Grup, jest w swojej istocie rozszerzeniem Chorda, ale ponieważ jest on tematem przewodnim tej pracy, został mu poświęcony rozdział 5.

Rozdział 4

Grupowanie węzłów

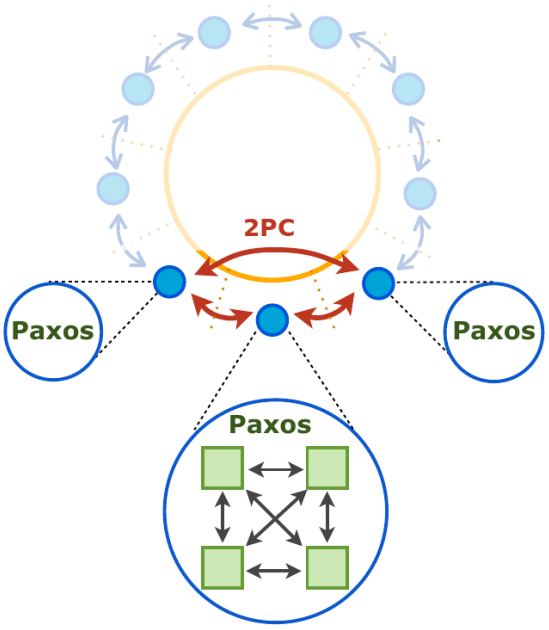
Zapewnianie istotnych własności systemów rozproszonych, takich jak skalowalność czy dostępność jest ważnym zagadnieniem, nad którym ciągle trwają badania. W wyniku prac badawczych powstaje wiele rozwiązań [1, 2, 5, 6, 23, 25, 26, 30, 31, 32]. Jednym z zaproponowanych podejść jest mechanizm grupowania węzłów w zbiory. Sugeruje on, aby podstawowym komponentem w sieci P2P nie był węzeł, lecz grupa węzłów, każda odpowiedzialna za dany zbiór kluczy. Komunikacja w sieci odbywałaby się na dwóch poziomach: międzygrupowym oraz wewnątrzgrupowym. Pomysł ten jest bezpośrednią inspiracją dla zaproponowanego w niniejszej pracy protokołu Statycznych Grup. Wprowadzenie do protokołu Statycznych Grup (rozdział 5) poprzedzone jest analizą istniejących rozwiązań grupowania węzłów w grupy. Dlatego też, mechanizmowi grupowania został poświęcony niniejszy rozdział, opisując jego ideę, sposób działania, oraz wyszczególniając co zapewnia, a co pomija.

4.1 System Scatter

Idea grupowania węzłów w środowisku DHT została wypromowana przez autorów systemu Scatter [31]. System ten powstał jako próba zapewnienia silnej spójności rozproszonych danych, zachowując jednocześnie dużą skalowalność. Autorzy twierdzą, że ich system osiąga zadowalającą spójność dla operacji na pojedynczej parze klucz-wartość pomimo zmiennych opóźnień w sieci, przypływów i odpływów oraz błędów komunikacyjnych. Twierdzą również, że ich system zapewnia dobrą skalowalność, w której to system sprawnie działa w niestabilnym środowisku nawet z ponad milionem węzłów. Zapewnienia te popierają eskperymentami.

Podstawowym komponentem w systemie Scatter jest samoorganizująca się grupa węzłów. Grupa wewnętrznie używa mechanizmów replikacji opartych na algorytmie konsensusu Paxos jako podstawy do zapewnienia spójności oraz odporności na awarie. Sam Scatter implementuje również kilka rozszerzeń i optymalizacji do podstawowego algorytmu Paxos, takich jak wybieranie lidera grupy, czy algorytm rekonfiguracji po utracie lub zyskaniu członka grupy. Ponieważ grupy zarządzają wewnętrzną integralnością używając sprawdzonych protokołów konsensusu, pro-

sty i agresywny detektor awarii wystarcza. Węzły, które zostały wyłączone z grupy, nie mogą brać udziału w jej akcjach. Gdyby jednak nie udało się dostatecznie szybko wykryć awarie węzła, grupa nadal mogłaby podejmować akcje, gdyż do działania potrzebne jest kworum węzłów.



Rysunek 4.1: Zagnieżdżony konsensus w pierścieniu Scatter.

Scatter implementuje prosty model DHT z przestrzenią kluczy w formie okręgu, którą nazywamy, podobnie jak w protokole Chord, pierścieniem. Przestrzeń ta jednak różni się od zaproponowanej w protokole Chord, tym że jest podzielona pomiędzy grupy, a nie węzły. Każda grupa posiada aktualne informacje o dwóch sąsiednich grupach: tej która ją poprzedza w pierścieniu, oraz następnej.

Sąsiednie grupy mogą wykonywać tzw. operacje międzygrupowe: podzielenie grupy na dwie mniejsze, połączenie dwóch grup, migracja członka z jednej grupy do drugiej oraz repartycja, czyli zmiana podziału odpowiedzialności za odpowiednie przedziały kluczy. Żeby nie dopuścić do niespójności pomiędzy węzłami, operacje międzygrupowe powinny być atomowe. Dlatego też, wykonywane są one w ramach rozproszonych transakcji, używając do tego protokołu zatwierdzania dwufazowego 2PC (ang. *two-phase commit protocol*). Autorzy rozwiązanie to nazywają zagnieżdżonym konsensusem. Ilustruje to rysunek 4.1, na którym widać, że operacje międzygrupowe wykonywane są przy pomocy 2PC (dzięki temu grupy ustalają "wspólną wersję wydarzeń"), a każdą decyzję, którą ma podjąć dana grupa ustalana jest pomiędzy węzłami należącymi do niej przy pomocy protokołu Paxos. W każdej grupie wyłaniany jest lider koordynujący akcje grupy. W przypadku jego awarii, inny węzeł może przejąć dowodzenie i kontynuować transakcję. Transakcja zakończy się pod warunkiem, że większość węzłów z każdej grupy pozostanie poprawna. W przeciwnym wypadku, po upływie ustalonego czasu, transakcja zostaje odrzucona. Ponadto, autorzy zalecają współbieżne wykonywanie operacji międzygrupowych.

Jednak w przypadku takiej implementacji, po zatwierdzeniu transakcji potrzebna jest przerwa na rekonfigurację.

Aby cały system działał poprawnie, każda grupa musi posiadać najświeższą wiedzę o jakimś podzbiorze członków każdej z grup sąsiednich. Aby to zapewnić, członkostwo grup jest rozsyłane sąsiadom za każdym razem, gdy ulegnie ono zmianie. Ponadto Scatter sam w sobie nie gwarantuje niezawodnego dostarczania wiadomości oraz nie zapewnia uporządkowania dostarczania wiadomości FIFO (ang. *First In, First Out* – pierwszy na wejściu, pierwszy na wyjściu).

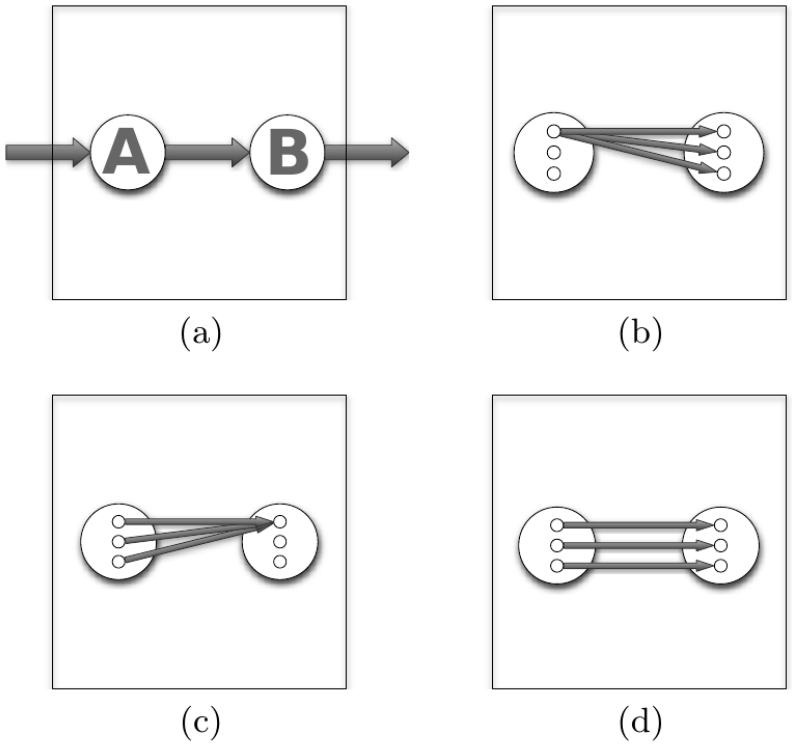
4.2 System Rollerchain

Inną, wartą uwagi implementacją mechanizmu grupowania węzłów jest Rollerchain [32]. Jest to system DHT, dla powstania którego bezpośrednią inspiracją był Scatter. Rollerchain implementuje kilka dodatkowych mechanizmów, doprecyzowując niektóre aspekty, których autorzy systemu Scatter celowo pominęli.

Rollerchain, w przeciwieństwie do Scatter, który przy wykonywaniu operacji opiera się na konsensusie, przez co są one kosztowne i mogą być blokowane przy dużych odpływach, bazuje na zalewaniu sieci wykorzystując mechanizmy plotkowania[1] (ang. *gossip-based mechanism*). Jednym z celów systemu Rollerchain jest zachowanie równomiernego obciążenia między węzłami, dlatego też strategia dołączania nowych węzłów do sieci uwzględnia aktualne obciążenie węzłów kluczami. Gdy węzeł chce dołączyć do sieci, odpytywane są losowe grupy. Nowy węzeł wybiera z tych grup najbardziej obciążoną i do niej dołącza.

Grupy w Rollerchain są połączone tzw. wirtualnymi łączami (ang. *virtual links*, patrz rysunek 4.2a), które tworzą logiczny pierścień (na wzór pierścienia Scatter). Łącza te powstają w wyniku istnienia bezpośrednich połączeń między węzłami z różnych grup — nazwijmy te połączenia *łączami fizycznymi*. Ponieważ replikowanie wszystkich danych na wszystkich członkach grupy nie jest wymagane, zbiory łączy fizycznych (każdy przechowywany na innym węźle) mogą się od siebie różnić. A więc, aby zapewnić odporność na awarie oraz zrównoważenie obciążenia na poziomie grupy, łącza fizyczne powinny być równomiernie rozłożone pomiędzy węzłami. Dla przykładu, sytuacja pokazana na rysunku 4.2b, w której tylko jeden węzeł jakiejś grupy posiada niepusty zbiór łączy fizycznych, jest silnie niepożądana. Podobnie niewskazana jest sytuacja, w której to wszystkie węzły jakiejś grupy posiadają jednakowy zbiór łączy fizycznych, z tylko jednym elementem (patrz rysunek 4.2c). Idealną strukturę łącza wirtualnego pokazuje rysunek 4.2d: każdy węzeł grupy A posiada łącze fizyczne do innego węzła z grupy B. Oczywiście zachowanie równomiernego rozłożenia łączy fizycznych wymaga dodatkowych operacji i monitoringu, lecz ze względu na ich nieistotność w kontekście protokołu Statycznych Grup, opis tych aspektów zostaje pominięty.

[1] Mechanizmy plotkarskie (ang. *gossiping*, *rumor monegering*), czy też epidemiczne (ang. *epidemic*) rozsyłają wiadomości w sposób zainspirowany rozchodzeniem się plotek między ludźmi, czy też rozprzestrzenianiem się epidemii na danym obszarze. Korzystają one z abstrakcji komunikacyjnej określanej jako probabilistyczne rozgłaszanie niezawodne (ang. *probabilistic reliable broadcast*) [22].



Rysunek 4.2: Przykłady łączy międzygrupowych.

4.3 Polityki

Sam mechanizm grupowania węzłów nie specyfikuje do której grupy trafi nowy węzeł, ani nie mówi o warunkach które muszą zajść, aby wykonać daną operację międzygrupową. Sposoby wybierania grup i warunki potrzebne do zajścia operacji międzygrupowych nazwijmy *politykami*. Zależą one od danej implementacji i mogą mieć różne motywacje. Mają one jednak istotny wpływ na budowę całej sieci, a więc i zachowanie całego systemu. Mogą więc pozytywnie wpłynąć na system, poprawiając niektóre parametry, lecz źle przemyślane mogą je tylko pogorszyć. Dla przykładu wspomniany w podrozdziale 4.2 Rollerchain stosuje opisaną tam politykę, aby zagwarantować równowagę obciążenia między grupami. Z kolei autorzy systemu Scatter nie specyfikują konkretnej polityki, pozostawiając swobodę w tej kwestii. Poniżej przedstawione jest kilka przykładowych polityk, które zostały zaproponowane jako warte uwagi [6].

4.3.1 Żywotność i równowaga obciążenia

Polityka nazwana przez autorów R-LB (ang. *Resilient Load-Balancing*), oprócz zrównoważenia obciążenia ma na celu zagwarantowanie żywotności grup. Aby zapewnić tą pierwszą własność, nowy węzeł dołącza do grup w których obciążenie przypadające na jeden węzeł jest największe, a gdy grupa musi się podzielić na dwie

mniejsze (bo np. stała się zbyt duża), to klucze są tak rozdzielone, aby obciążenie przypadające na węzeł było możliwie równe w obu grupach. Oprócz tego, polityka R-LB gwarantuje jeszcze własność żywotności grup. Aby ją zapewnić, nowe węzły wybierają z najbardziej obciążonych grup te najmniejsze i do nich dołączają. Ponadto grupy niebezpiecznie małe inicjują operację połączenia się z grupami sąsiednimi.

4.3.2 Duże grupy

Polityka nazwana *Supersize-me* polega na tym, aby wszystkie grupy były relatywnie duże, tzn. żeby składały się z większej ilości węzłów, niż jest to wymagane ze względu na zachowanie odpowiedniego stopnia replikacji. Przez to redundancja danych jest zwiększona, aczkolwiek szansa, że grupy będą małe (co może skutkować całkowitym zniknięciem grupy, bądź kosztownymi operacjami łączenia się grup) jest niska. W zamian za zmniejszenie transferu danych przy odpływach, transfer jest zwiększony przy przypływach.

4.3.3 Małe grupy

Polityka nazwana *Avoid-Surplus* jest przeciwieństwem polityki Supersize-me. Jej celem jest utrzymywanie rozmiaru grup na jak najniższym poziomie, który jednocześnie zapewniłby pożądany stopień replikacji. Osiąga się to poprzez dołączanie nowych węzłów do grup największych, co skutkuje częściej wykonywaną operacją podziału grup. Dzięki temu zredukowana jest redundancja danych w systemie i zmniejszone są koszty monitoringu (które zależą w dużym stopniu od rozmiaru grup). Wadą jest jednak fakt, iż system staje się bardziej wrażliwy na odpływy, które mogą doprowadzać do częstych operacji łączenia grup (ze względu na ich nieakceptowalny, zbyt mały rozmiar).

4.3.4 Węzły niestabilne odpowiedzialne za mniej kluczy

Polityka nazwana *Hotter-on-Ephemeral* polega na dołączaniu najbardziej zawodnych węzłów, do grup przechowujących najczęściej używane dane. Żeby zachować równowagę obciążenia, grupy te będą przechowywać najmniej kluczy. A więc polityka ta zapewnia, że do grup które trzymają mało kluczy, dołączać będą bardziej zawodne węzły. Przez to najwięcej dołączeń (które generują węzły niestabilne) będzie właśnie w ramach tych grup. Skutkuje to zmniejszeniem przesyłu danych (przy przypływach) i kosztów monitoringu, oraz podobnie jak w R-LB zapewnia dobre zrównoważenie obciążenia. Ponadto, aby zachować stabilność grup, najbardziej zawodne grupy mogą być większe niż inne. Polityka Hotter-on-Ephemeral zakłada, że znana jest wartość metryki stabilności pojedynczego węzła. Może być ona wyznaczana na różne sposoby. Podobnie też, w protokole Statycznych Grup znajomość poziomu stabilności każdego nowego węzła jest istotna, aby móc zdecydować, jaką akcję podjąć (więcej informacji o polityce protokołu Statycznych Grup w rozdziale 5).

Rozdział 5

Zaproponowane rozwiązanie

Celem niniejszej pracy magisterskiej jest zaproponowanie implementacji rozszerzenia protokołu Chord. Rozszerzeniem tym jest protokół Statycznych Grup (SG), rozszerzający protokół Chord o mechanizm grupowania węzłów. Niniejszy rozdział opisuje zaproponowany protokół, wskazuje jego zalety, wady, aspekty które pomija, oraz przedstawia pseudokod, szczegółowo opisując działanie protokołu.

5.1 Opis ogólny

Głównym celem protokołu Statycznych Grup jest zapewnienie infrastruktury dla replikacji. Dlatego też, zdecydowano się na strukturę grup, w której węzły replikują dane między sobą. Podobnie jak węzły w protokole Chord tworzą pierścień Chord (patrz paragraf 3.2.1), tak grupy w protokole Statycznych Grup, tworzą *pierścień Statycznych Grup*. Istotnym zagadanienim, opisanym w tym podrozdziale, jest stabilności pojedynczych węzłów, która pozwala na podejmowanie decyzji odnośnie członkostwa węzłów w grupach.

5.1.1 Grupy statyczne

Aby zapewnić infrastrukturę sprzyjającą replikacji, protokół Statycznych Grup używa podejścia opartego na agregowaniu węzłów w grupy. Mechanizm grupowania opisany został w rozdziale 4. Jakkolwiek SG korzysta z ogólnego podejścia opartego na grupowaniu, to nie implementuje kosztownych operacji międzygrupowych, wykorzystywanych m.in. w protokołach Scatter (patrz podrozdział 4.1), czy Rollerchain (patrz podrozdział 4.2) — stąd też określenie *statyczny*. Grupy w protokole Statycznych Grup są statyczne, ponieważ nie łączą się wzajemnie, gdy jest w nich za mało węzłów, ani nie dzielą na kilka mniejszych grup, gdy jest w nich zbyt dużo węzłów, lecz każda grupa, ma określony maksymalny rozmiar oraz istnieje tak długo, dopóki jest w sieci przynajmniej jeden poprawny węzeł będący jej członkiem. Jeżeli jedyny członek opuści sieć, to owa grupa w naturalny sposób przestaje istnieć w sieci. Dzięki takiemu podejściu, nie ponosimy kosztów operacji międzygrupowych, które są bardzo wymagające (patrz zagnieżdżony konsensus w podrozdziale 4.1).

Narażamy się jednak na sytuację, w której tworzona jest nowa grupa za pomocą węzła, który po chwili może odejść; a po jego odejściu może dołączyć do sieci nowy węzeł i ponownie utworzyć grupę, lecz po chwili i on może opuścić sieć. Owa sytuacja jest niepożądana, gdyż każdemu utworzeniu nowej grupy, towarzyszy przesłanie części danych z innej grupy. Również przy zniknięciu grupy ponoszone są koszty: jeżeli węzeł uległ awarii, bądź odłączył się nie informując innych, dane za które odpowiadał mogą zaginąć[1]; a jeżeli nie uległ awarii i przed odejściem wykonał operację przeniesienia danych do innej grupy[2], ponoszone są koszty owego przeniesienia.

5.1.2 Stabilność węzła

Aby uniknąć sytuacji opisanej w powyższym paragrafie, w której to nowe węzły tworzą nową grupę, by po chwili opuścić sieć, protokół Statycznych Grup stosuje politykę, która nakłada ograniczenie na powoływanie do życia nowych grup. Mianowicie, tylko węzły określone jako stabilne mogą tworzyć nowe grupy. Reszta węzłów może jedynie dołączać do grup już istniejących. Wyjątkiem jest sytuacja, gdy wszystkie grupy w sieci są już przepełnione, tzn. posiadają maksymalną liczbę członków, która jest określona odpowiednim parametrem. Protokół Statycznych Grup nie specyfikuje sposobu określania stabilności węzła, pozostawiając wolność w tej kwestii. Przykładowo, stabilność węzła mogła by być określana na podstawie jego historii bycia członkiem sieci, która mogłaby być zapisywana u niego lokalnie, na innych węzłach (sąsiednich, lub specjalnych — wyróżnionych), bądź przy użyciu jakiegoś serwisu zewnętrznego. Ponadto, do określania stabilności, mogłyby być brane pod uwagę parametry węzła, takie jak jego opóźnienia komunikacyjne, przepustowość jaką dysponuje, czy nawet sposób łączenia się do sieci (połączenia bezprzewodowe z reguły są mniej stabilne).

5.1.3 Replikacja

Głównym celem grupowania węzłów jest replikacja danych między nimi. Dzięki abstrakcji, którą zapewniają grupy węzłów, protokół Statycznych Grup może działać na wzór Chorda, jednocześnie zapewniając środowisko do replikacji, lecz abstrahując od konkretnego sposobu powielania danych. Zalecane jest, aby repliki danych, za które odpowiedzialna jest grupa, były na każdym węźle będącym członkiej danej grupy. Sposób uspójniania replik nie jest wyspecyfikowany w protokole Statycznych Grup i zależy od konkretnej implementacji.

[1]Sytuacji w której część danych ginie może zapobiec podejście, w którym to grupa poprzednio odpowiedzialna za owe dane (jeszcze przed powstaniem zaginionej grupy) zachowa je u siebie i odtworzy.

[2]Operacji przeniesienia danych nie specyfikuje protokół Statycznych Grup. Podobnie jak w przypadku protokołu Chord, za ową, ewentualną operację przeniesienia odpowiedzialna jest aplikacja korzystająca z protokołu.

5.2 Opis szczegółowy

Aby szczegółowo opisać działanie protokołu Statycznych Grup, w podrozdziale tym przedstawiony i omówiony jest pseudokod opisujący logikę działania protokołu. Ponieważ SG rozszerza protokół Chord, w wielu miejscach działając w taki sam sposób — w podrozdziale tym pominięto kwestie, które zostały już opisane w podrozdziale 3.2. Istotne różnice pojawiają się w momencie dołączania węzła do sieci, gdzie potrzebna jest dodatkowa logika oraz funkcje odpowiedzialne za znalezienie odpowiedniej grupy. Zmiany wprowadza również dodatkowa struktura reprezentująca grupę węzłów, przez którą można wywoływać zdalne metody. Ponadto dochodzi okresowe sprawdzanie poprawności członków grupy.

5.2.1 Struktury i parametry

Bardzo ważnym elementem w protokole Statycznych Grup jest przedstawiona na listingu 5.1 struktura `Group`. Reprezentuje ona zbiór węzłów tworzących jedną grupę. Składa się z identyfikatora grupy `id`, oraz listy adresów `addresses` wszystkich jej członków. Poprzez obiekt owej struktury, można odwołać się do węzła danej grupy. W listingach wywołanie zdalnej metody dla dowolnego członka grupy[3] oznaczane jest symbolem `->`. Przykładowo `group->find_successor(3)` oznacza wywołanie zdalnej metody `find_successor(3)` dla członka grupy `group`. W podobny sposób można odwołać się do konkretnego węzła, żądając wykonania zdalnej metody, np. `group.address[2]->find_successor(3)`.

Istotnym, w kontekście protokołu Chord, typem złożonym jest struktura `Finger`, przedstawiona na listingu 5.2. Reprezentuje ona pojedynczy wpis w tablicy trasowania, który składa się z numeru `i`, początku zakresu kluczy `start`, końca zakresu kluczy `end` oraz grupy `group` odpowiedzialnej za dany zakres kluczy.

```
1 Group:
2     Int id
3     List<String> addresses
```

Listing 5.1: Pseudokod struktury reprezentującej grupę węzłów.

Główną jednostką protokołu jest struktura węzła, której pseudokod przedstawiony jest na listingu 5.3. Widać, że węzeł składa się ze zmiennej `next`, adresu `address`, pola określającego stabilność węzła `stability`, obiektów grupy `group`, poprzednika `predecessor` i następnika `successor`, oraz tablicy trasowania `fingerTable`. Zmienna `next` wykorzystywana jest w procedurze `fix_fingers` (listing 5.15) i oznacza indeks następnego wpisu w tablicy trasowania do sprawdzenia. Adres zawiera unikalny adres danego węzła, przez który inne węzły mogą się z nim

[3]Wybór członka, który wykona zdalną metodę zależny jest od implementacji. Sugerowane jest aby był to członek losowy. Możliwa jest również taka implementacja, w której to wszyscy członkowie grupy wykonują zdalną metodę.

```
1 Finger:
2     Int i
3     Int start
4     Int end
5     Group group
```

Listing 5.2: Pseudokod struktury reprezentującej pojedynczy wpis w tablicy trasowania.

```
1 Node:
2     Int next
3     String address
4     Float stability
5     Finger[] fingerTable
6     Group group
7     Group successor
8     Group predecessor
```

Listing 5.3: Pseudokod struktury reprezentującej węzeł.

komunikować (najczęściej jest to adres IP). Węzły należące do tej samej grupy uspójniają między sobą pola `fingerTable`, `group`, `successor` i `predecessor`. Sposób uspójniania tych pól między węzłami w grupie, podobnie jak sposób uspójniania replik między węzłami w grupie, nie jest ściśle określony i zależy od konkretnej implementacji.

Istotnym elementem wpływającym na działanie protokołu Statycznych Grup są trzy parametry: `M` — oznaczający, podobnie jak w protokole Chord, liczbę bitów przeznaczonych na pojedynczy identyfikator oraz liczbę wpisów w tablicy trasowania; `MAX_GROUP_SIZE` — mówiący o maksymalnej liczbie węzłów, które mogą tworzyć jedną grupę; oraz `STABILITY_REQUIREMENT` — wymagany poziom stabilności do utworzenia nowej grupy.

5.2.2 Lokalizacja grupy odpowiedzialnej za klucz

Podobnie jak w protokole Chord, główną operacją SG jest lookup, czyli odnalezienie lokalizacji określonego obiektu. Operacja ta różni się jednak od zaproponowanej w protokole Chord (patrz paragraf 3.2.2) tym, że szukana jest grupa węzłów, a nie pojedynczy węzeł. Za operację lookup odpowiedzialna jest funkcja `find_successor` pokazana na listingu 5.4. Działa ona analogicznie jak jej odpowiednik w protokole Chord i korzysta z funkcji `closest_preceding_group` pokazanej na listingu 5.5, która również ma swojego odpowiednika w protokole Chord (`closest_preceding_node`).

```
1 n.find_successor(id):
2     if id ∈ (group.id, successor.id>
3         return successor
4     else
5         g = closest_preceding_node(id)
6         return g->find_successor(id)
```

Listing 5.4: Pseudokod funkcji odpowiedzialnej za znalezienie lokalizacji obiektu o danym identyfikatorze.

```
1 n.closest_preceding_group(id):
2     for i = M downto 1
3         if fingerTable[i].group.id ∈ (group.id, id)
4             return fingerTable[i].group
5     return group
```

Listing 5.5: Pseudokod funkcji odpowiedzialnej za znalezienie najbliższego poprzednika obiektu o danym identfikatorze z tablicy trasowania.

5.2.3 Dołączanie do sieci

Węzeł może dołączyć do istniejącego pierścienia Statycznych Grup dzięki węzłowi będącemu już w istniejącym pierścieniu, lub może stworzyć nowy pierścień. Pokazuje to listing 5.6, na którym przedstawiona jest procedura `start`, która działa w następujący sposób: jeżeli nie podano adresu `addr`, z którym węzeł `n` ma się połączyć, to wywołuje on procedurę `create` (listing 5.7), tworząc nowy pierścień. W przeciwnym wypadku, węzeł `n` wywołuje zdalną procedurę `find_group_to_join` (listing 5.11) dla węzła reprezentowanego przez adres `addr`, która zwraca obiekt grupy `g`. Węzeł `n` tworzy nową grupę, jeżeli węzeł `addr` zwróci pusty obiekt `null`, lub gdy wartość stabilności `stability` węzła `n` jest większa lub równa parametrowi `STABILITY_REQUIREMENT` i jednocześnie rozmiar grupy `g` jest większy lub równy połowie wartości parametru `MAX_GROUP_SIZE`. Jeżeli tak nie jest, to węzeł `n` dołącza do grupy `g`. Dzięki takiej logice, nowe grupy tworzone są tylko przez węzły stabilne, lub gdy nie ma wolnej grupy do dołączenia; jednocześnie, jeżeli zaproponowana grupa `g` jest mała (jej rozmiar jest mniejszy od połowy wartości parametru `MAX_GROUP_SIZE`) to węzeł stabilny zamiast stworzyć nową grupę, dołaczą do `g`. Zapobiega to tworzeniu nowych grup w przypadku, gdy inne grupy są małe.

Wartą omówienia jest funkcja `find_group_to_join`, która zwraca wolną grupę do dołączenia. Zaimplementowana jest w taki sposób, aby znaleźć w obrębie zbioru grup, grupę najmniejszą. Owy zbiór składa się z grup będących w tablicach trasowania tych grup, które posiada w swojej tablicy trasowania węzeł `n`. W zależności od polityki, funkcja ta mogłaby wybierać grupę na odmiennej zasadzie. Dla przykładu mogłaby brać pod uwagę stabilność węzła i na tej podstawie dobrać taką grupę, aby zachować różnorodność węzłów w grupie.

```
1 n.start(addr):
2     if addr is null
3         create()
4     else
5         g = addr->find_group_to_join()
6         if g is null or (stability ≥ STABILITY_REQUIREMENT and
              size(g.addresses) ≥ (MAX_GROUP_SIZE / 2))
7             join(addr)
8         else
9             join_to_group(g)
```

Listing 5.6: Pseudokod procedury dołączania węzła do istniejącego pierścienia, bądź utworzenie własnego pierścienia.

```
1 n.create():
2     predecessor = group
3     successor = group
```

Listing 5.7: Pseudokod procedury utworzenia własnego pierścienia.

```
1 n.join(addr):
2     predecessor = null
3     successor = addr->find_successor(group.id)
```

Listing 5.8: Pseudokod procedury dołączenia do istniejącej sieci, przy pomocy węzła reprezentowanego przez adres `addr`, tworząc nową grupę.

```
1 n.join_to_group(g):
2     group = g
3     g->add_address(address)
```

Listing 5.9: Pseudokod procedury dołączenia węzła do istniejącej grupy `g`.

```
1 n.add_address(addr):
2     group.addresses.add(addr)
```

Listing 5.10: Pseudokod procedury dodania adresu do listy adresów węzłów w grupie.

5.2.4 Stabilizacja

Każdy węzeł okresowo uruchamia procedury stabilizacji, aby mieć aktualne wskaźniki na grupę następną, poprzednią oraz aktualną tablicę trasowania. Stabilizacja w protokole Statycznych Grup przebiega analogicznie do stabilizacji w protokole Chord, opisanej w paragrafie 3.2.3. Uruchamiane są procedury `stabilize` (listing 5.13), `fix_fingers` (listing 5.15) oraz `check_predecessor` (listing 5.16). Ale po-

```
1  n.find_group_to_join():
2      smallestGroup =
           fingerTable[1].group->smallest_group_from_fingertable()
3      for i = 2 to M
4          g = fingerTable[1].group->smallest_group_from_fingertable()
5          if size(g.addresses) < size(smallestGroup.addresses)
6              smallestGroup = g
7      if size(group.addresses) is MAX_GROUP_SIZE
8          return null
9      else
10         return smallestGroup
```

Listing 5.11: Pseudokod funkcji odpowiedzialnej za zwrócenie grupy do dołączenia.

```
1  n.smallest_group_from_fingertable():
2      smallestGroup = fingerTable[1].group
3      for i = 2 to M
4          if size(fingerTable[i].group.addresses) <
               size(smallestGroup.addresses)
5              smallestGroup = fingerTable[i].group
6      return smallestGroup
```

Listing 5.12: Pseudokod funkcji zwracającej najmniejszą grupę z grup w tablicy trasowania węzła `n`.

nieważ, w odróżnieniu od protokołu Chord, dochodzi jeszcze jeden komponent — grupa, należy dodatkowo sprawdzać jego integralność. Zajmuje się tym procedura `check_group` pokazana na listingu 5.17, w której to sprawdzana jest poprawność wszystkich węzłów w grupie, aby mieć ich aktualną listę.

```
1  n.stabilize():
2      p = successor->predecessor
3      if p.id ∈ (group.id, successor.id)
4          successor = p
5      successor->notify(group)
```

Listing 5.13: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za weryfikację bezpośredniego nastepnika `successor` grupy węzła `n`, oraz powiadomienie jej o jego grupie `group`.

```
1 n.notify(g):
2     if predecessor is null or g.id ∈ (predecessor.id, group.id)
3         predecessor = g
```

Listing 5.14: Pseudokod procedury odpowiedzialnej za ewentualną aktualizację poprzednika grupy.

```
1 n.fix_fingers():
2     next = next + 1
3     if next > m
4         next = 1
5     fingerTable[next].group = find_successor(fingerTable[next].start)
```

Listing 5.15: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za aktualizację wpisów w tablicy trasowania.

```
1 n.check_predecessor():
2     if predecessor failed
3         predecessor = null
```

Listing 5.16: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za wyzerowanie wskaźnika na poprzednika `predecessor` w przypadku braku odpowiedzi od jakiegokolwiek węzła tej grupy.

```
1 n.check_group():
2     for a in group.addresses:
3         if a failed
4             group.addresses.remove(a)
```

Listing 5.17: Pseudokod procedury wywoływanej okresowo i odpowiedzialnej za weryfikację poprawności węzłów tworzących grupę.

Rozdział 6

Testy symulacyjne

Rozdział ten przedstawia środowisko symulacyjne *PeerSim*, w którym przetestowano protokół Statycznych Grup. Przedstawione są parametry protokołu (takie jak maksymalna liczność grup, czy wymagana stabilność do utworzenia nowej grupy) oraz parametry sieci (takie jak liczba węzłów, czy częstotliwość dołączeń i odłączeń). Dalej, opisane są przeprowadzone testy wpływu parametrów protokołu i parametrów sieci na symulowaną sieć w której działa protokół Statycznych Grup. Rozdział kończą wnioski dotyczące wyników testów.

6.1 PeerSim

Do zaimplementowania i przetestowania protokołu Statycznych Grup wykorzystano środowisko symulacyjne PeerSim [42], które pozwala na uruchamianie i testowanie protokołów. Rozpowszechniane jest na licencji wolnego oprogramowania *GNU General Public License* (GPL). Napisane jest w języku Java i składa się z dwóch silników symulacyjnych. Pierwszym jestem silnik wykorzystujący do działania cykle, w którym to akcje wybranych obiektów wywoływane są w wybranych cyklach. Drugim jest silnik korzystający ze zdarzeń, w którym to akcje obiektów inicjowane są przez zdarzenia. W implementacji protokołu Statycznych Grup wykorzystywany jest silnik oparty na cyklach, który abstrahuje od warstwy transportowej oraz od współbieżności, a więc obiekty węzłów komunikują się ze sobą bezpośrednio i wszystkie operacje w sieci wykonywane są sekwencyjnie.

Działanie symulatora uzależnione jest od pliku konfiguracyjnego, w którym ustawia się wartości globalnych parametrów, takich jak liczność sieci, czy liczba eksperymentów. Ponadto, w pliku tym wskazuje się klasy, z których symulator ma korzystać podczas symulacji, oraz definiuje się ewentualne parametry tych klas. Przykładowy plik konfiguracyjny wykorzystywany w implementacji protokołu Statycznych Grup pokazany jest na listingu 6.1. Widać na nim ustawione wartości globalnych parametrów (pierwsze 4 linie), kolejno: liczba eksperymentów, ziarno dla generatora liczb pseudolosowych, liczba cykli symulatora oraz liczność sieci. Dalej, w linii 6 wskazano klasę, która ma zostać uruchomiona na początku każdego eksperymentu. Wskazuje na to słowo kluczowe `init`, po którym następuje dowol-

```
1  simulation.experiments 1
2  random.seed 1234567890
3  simulation.cycles 1000
4  network.size 1000
5
6  init.create staticgroups.CreateInitialNodes
7
8  protocol.sg staticgroups.StaticGroupsProtocol
9  {
10     DEBUG 1
11     protocol sg
12     M 11
13     MAX_GROUP_SIZE 10
14     STABILITY_REQUIREMENT 0.5
15 }
16
17 control.dnet staticgroups.RandomDynamicNetwork
18 {
19     RANDOM 1
20     RANDOM_ADD_PROBABILITY 0.8
21     step 1
22     init.0 staticgroups.StaticGroupsInitializer
23 }
24
25 control.maintain staticgroups.StaticGroupsMaintainer
26 {
27     step 1
28 }
29
30 control.tests staticgroups.StaticGroupsTests
31 control.metrics staticgroups.StaticGroupsMetrics
```

Listing 6.1: Przykładowy plik konfiguracyjny środowiska symulacyjnego PeerSim.

ny identyfikator klasy, w tym wypadku `create`. Klasą, na którą wskazuje linia 6 jest `CreateInitialNodes` z pakietu `staticgroups`. Następnie, w linii 7, na podobnej zasadzie wskazana jest klasa protokołu `StaticGroupsProtocol` z pakietu `staticgroups` o identyfikatorze `sg`. Dalej wskazano klasy, które oznaczone są jako `control`, czyli wykonują jakąś akcję co cykl, dzięki czemu mogą wpływać na przebieg symulacji, bądź zbierać informacje o niej. Wskazane klasy, wszystkie z pakietu `staticgroups`, to kolejno: `RandomDynamicNetwork`, `StaticGroupsMaintainer`, `StaticGroupsTests`, oraz `StaticGroupsMetrics`.

Klasa `CreateInitialNodes` odpowiada za dołączenie węzłów do sieci przed rozpoczęciem eksperymentu[1]. `StaticGroupsProtocol` jest klasą protokołu, której obiekt posiada każdy węzeł. Klasa ta przyjmuje parametry pokazane w liniach od 10 do 15 na listingu 6.1:

[1]Dołączenie węzłów do sieci przed rozpoczęciem eksperymentu jest istotne, aby móc zbadać zachowanie się istniejącej już sieci bez potrzeby tworzenia jej podczas trwania eksperymentu.

- `DEBUG`: gdy parametr ten ustawiony jest na wartość 1, na standardowym wyjściu wypisywane są informacje o przebiegu symulacji.
- `protocol`: identyfikator protokołu w symulatorze PeerSim.
- `M`: liczba bitów przeznaczonych na pojedynczy identyfikator obiektu oraz liczba wpisów w tablicy trasowania.
- `MAX_GROUP_SIZE`: maksymalny rozmiar grupy.
- `STABILITY_REQUIREMENT`: wartość zmiennoprzecinkowa wyznaczająca minimalną wartość stabilności węzła, potrzebną do utworzenia przez niego nowej grupy.

Klasa `RandomDynamicNetwork` odpowiada za zmiany w sieci (dołączanie węzłów do sieci lub odłączanie węzłów od sieci). W liniach od 20 do 24 na listingu 6.1 zdefiniowane są wybrane parametry tej klasy:

- `RANDOM`: w przypadku zmiany w sieci, wartość 1 tego parametru wymusza pseudolosowość w określaniu, czy w danym cyklu do sieci ma zostać dodany węzeł, czy z sieci ma zostać usunięty węzeł.
- `RANDOM_ADD_PROBABILITY`: parametr ten określa prawdopodobieństwo dodania węzła do sieci (zamiast jego usunięcia), jeżeli w sieci ma nastąpić zmiana. Ma znaczenie tylko w przypadku, gdy `RANDOM` równe jest 1.
- `step`: parametr określający, co ile cykli ma nastąpić zmiana w sieci.
- `init.0`: parametr wskazujący klasę, która ma zostać użyta w przypadku dodania węzła do sieci.

Następna klasa znajdująca się na listingu 6.1, `StaticGroupMaintainer`, odpowiada za uruchamianie na węzłach procedur stabilizacji. Pokazany w linii 28 na listingu 6.1 parametr `step` określa co ile cykli mają być uruchamiane procedury stabilizacji na każdym węźle. Dalej, klasa `StaticGroupTests` odpowiedzialna jest za testy poprawności wskaźników grup (pola `successor` i `predecessor`, oraz wpisy w `fingerTable`). Ostatnią klasą pokazaną na listingu 6.1 jest `StaticGroupMetrics`, która zbiera informacje o przebiegu symulacji, takie jak czas trwania symulacji, czy średnia liczność grup i wypisuje je na standardowym wyjściu.

6.2 Testy

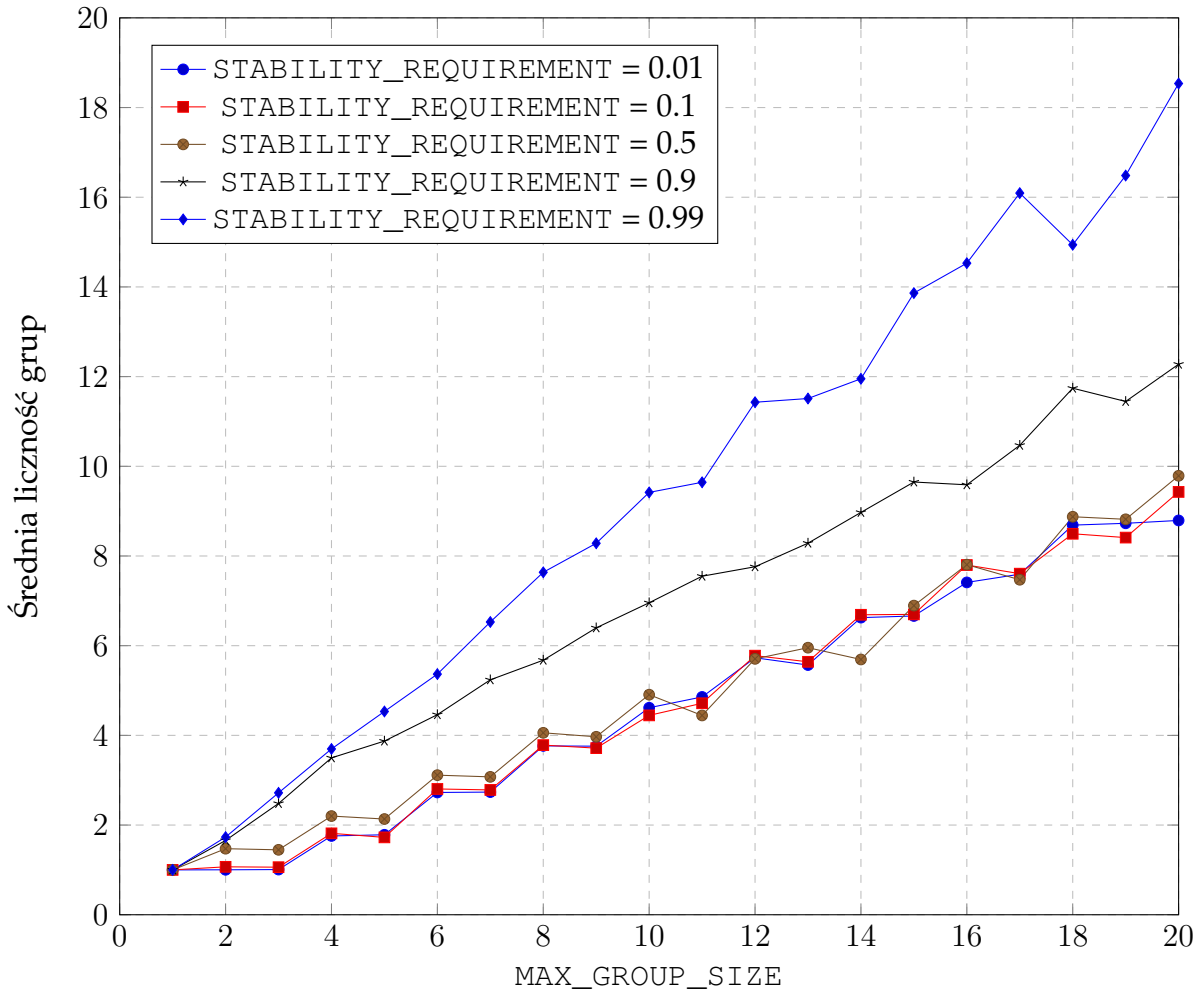
Działanie protokołu Statycznych Grup jest zależne od parametrów sieci takich jak liczba węzłów, częstotliwość zmian w sieci (co ile cykli symulatora następuje dołączenie lub odłączenie węzła), czy prawdopodobieństwo dołączenia węzła w przypadku zmiany w sieci (`RANDOM_ADD_PROBABILITY`), oraz od parametrów protokołu Statycznych Grup takich jak maksymalny rozmiar grupy (`MAX_GROUP_SIZE`),

czy wymagana stabilność do utworzenia nowej grupy (STABILITY_REQUIREMENT). W podrozdziale tym przedstawiono wpływ tych parametrów na działanie protokołu Statycznych Grup w środowisku symulacyjnym PeerSim. Dla wszystkich testów parametr M klasy StaticGroupsProtocol równy był 11 (aby w sieci było dostatecznie dużo wolnych identyfikatorów), parametr RANDOM klasy RandomDynamicNetwork równy był 1 (aby dołączanie i odłączanie węzłów od sieci odbywało się w kolejności losowej), parametr step klasy RandomDynamicNetwork równy był 1 (aby w każdym cyklu odbywała się zmiana w sieci), parametr step klasy StaticGroupsMaintainer równy był 1 (aby w każdym cyklu węzły uruchamiały procedury stabilizacji). Wartości pozostałych, istotnych dla przebiegu poniższy testów, parametrów podane są w ramach opisów każdego z testów[2], a wartości parametrów nieistotnych dla przebiegu poniższych testów zostały pominięte. Wynikiem pojedynczego eksperymentu był jeden pomiar. Ponadto, stabilność każdego węzła była losowana z zakresu od 0 do 1, a każde odłączenie węzła odbywało się w następujący sposób: losowane były cztery węzły z całej sieci i z nich odłączał się ten, który miał najmniejszą stabilność.

6.2.1 Średnia liczność grup

Wykres 6.1 przedstawia wpływ parametru MAX_GROUP_SIZE na średnią liczność grup. Wpływ ten zbadano dla wybranych wartości parametru STABILITY_REQUIREMENT. Parametry testu, dla którego powstał wykres 6.1: liczba eksperymentów — 20, początkowa liczba węzłów — 1000, liczba cykli — 1000, prawdopodobieństwo dołączenia nowego węzła w przypadku zmiany w sieci — 0.5. Z wykresu wynika, że średnia liczność grup rośnie liniowo wraz ze wzrostem maksymalnego rozmiaru grup i jest jednocześnie zależna od parametru STABILITY_REQUIREMENT. Widać, że gdy parametr STABILITY_REQUIREMENT równy jest 0.99, to średnia liczność grup jest bliska maksymalnej liczności grup, a gdy STABILITY_REQUIREMENT równy jest 0.01, 0.1, lub 0.5, to średnia liczność grup jest około dwa razy niższa, niż maksymalna liczność grup. Jest tak dlatego, ponieważ parametr STABILITY_REQUIREMENT odpowiada za minimalną wartość stabilności węzła potrzebną, aby mógł on utworzyć nową grupę, a więc im wyższa wartość tego parametru, tym tworzenie nowych grup w sieci rzadziej ma miejsce, co skutkuje tym, iż nowe węzły częściej dołączają do istniejących grup, podwyższając ich liczność. Ponadto średnia liczność grup nie spada znacząco zmniejszając parametr STABILITY_REQUIREMENT z wartości od 0.5 do 0.01. Jest tak, ponieważ protokół dąży do tego, aby grupy miały rozmiar większy niż połowa MAX_GROUP_SIZE (patrz opisany w paragrafie 5.2.3 warunek w linii 6 na listingu 5.6).

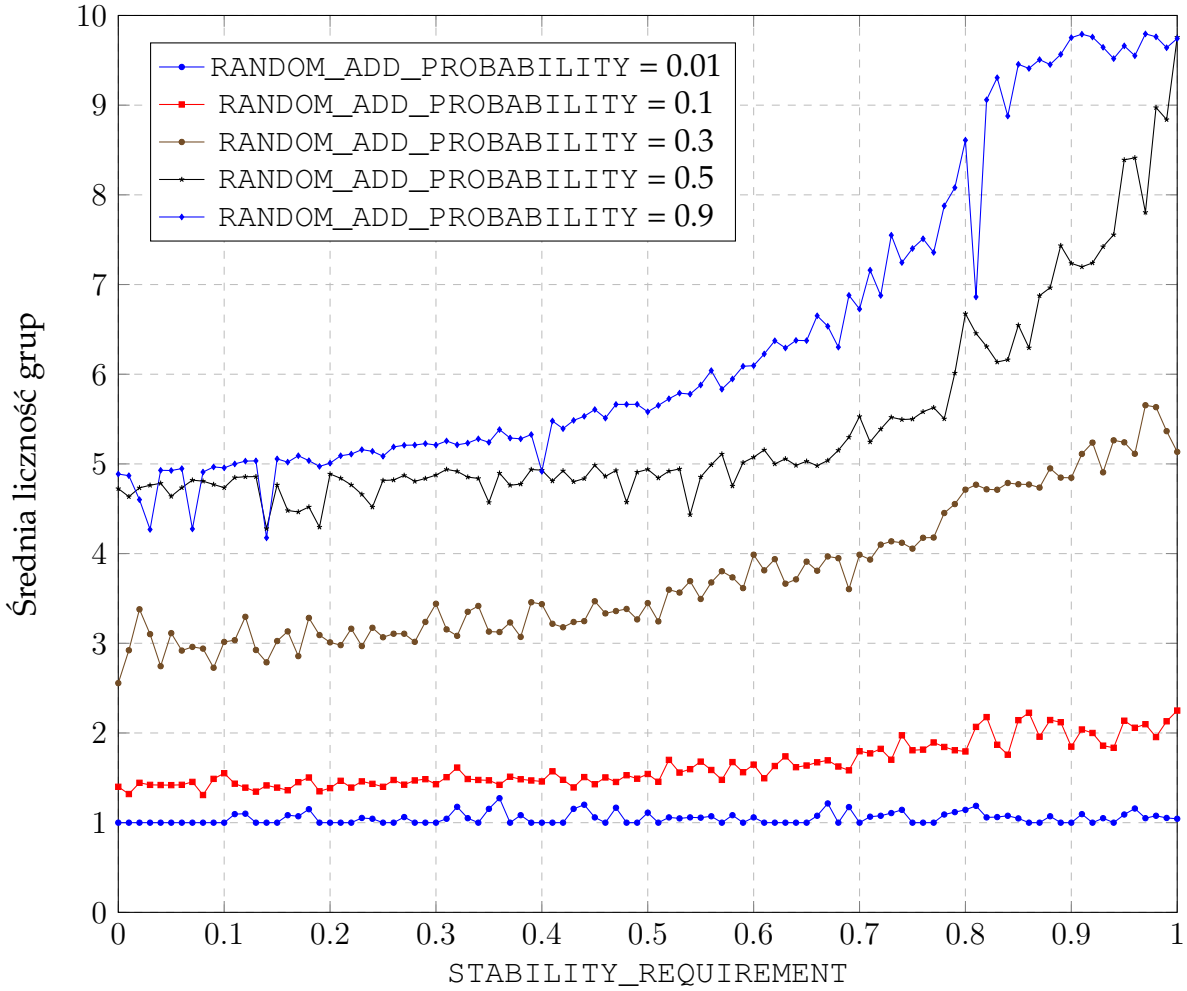
[2]Liczba cykli oraz liczba węzłów w sieci zostały tak dobrane w każdym teście, aby zachować sensowny czas wykonania testu.



Wykres 6.1: Wpływ parametru `MAX_GROUP_SIZE` wraz z wybranymi wartościami `STABILITY_REQUIREMENT` na średnią liczność grup.

Wykres 6.2 przedstawia wpływ parametru `STABILITY_REQUIREMENT` na średnią liczność grup. Wpływ ten zbadano dla różnych wartości parametru `RANDOM_ADD_PROBABILITY`. Parametry testu, dla którego powstał wykres 6.2: liczba eksperymentów — 101, początkowa liczba węzłów — 1000, liczba cykli — 1000, maksymalny rozmiar grupy — 10. Z wykresu wynika, że średnia liczność grup rośnie wraz ze wzrostem parametru `STABILITY_REQUIREMENT`. Wzrost średniej liczności grup jest największy dla sytuacji, w której to jest więcej dołączeń niż odłączeń węzłów (parametr `RANDOM_ADD_PROBABILITY` przyjmuje wartości powyżej 0.5); gdy jest jednak więcej odłączeń, niż dołączeń (parametr `RANDOM_ADD_PROBABILITY` przyjmuje wartości poniżej 0.5), wzrost średniej liczności grup jest zdecydowanie mniejszy, a dla skrajnie niskich wartości parametru `RANDOM_ADD_PROBABILITY` (0.01), wzrost średniej liczności jest niezauważalny. Wynika to z faktu, iż gdy w sieci jest więcej odłączeń węzłów, niż dołączeń węzłów, to grupy częściej tracą członków, niż zyskują. Jeżeli jednak więcej węzłów się dołącza niż odłącza, grupy się wypełniają węzłami, co zwiększa ich liczność. Ponadto, przy wysokich wymaganiach odnośnie

tworzenia nowych grup (gdy parametr STABILITY_REQUIREMENT przyjmuje duże wartości), rzadko który węzeł jest na tyle stabilny, aby utworzyć nową grupę, zamiast dołączyć do istniejącej; a więc węzły częściej dołączają do istniejących grup, niż tworzą nowe, co zwiększa średnią liczność grup.



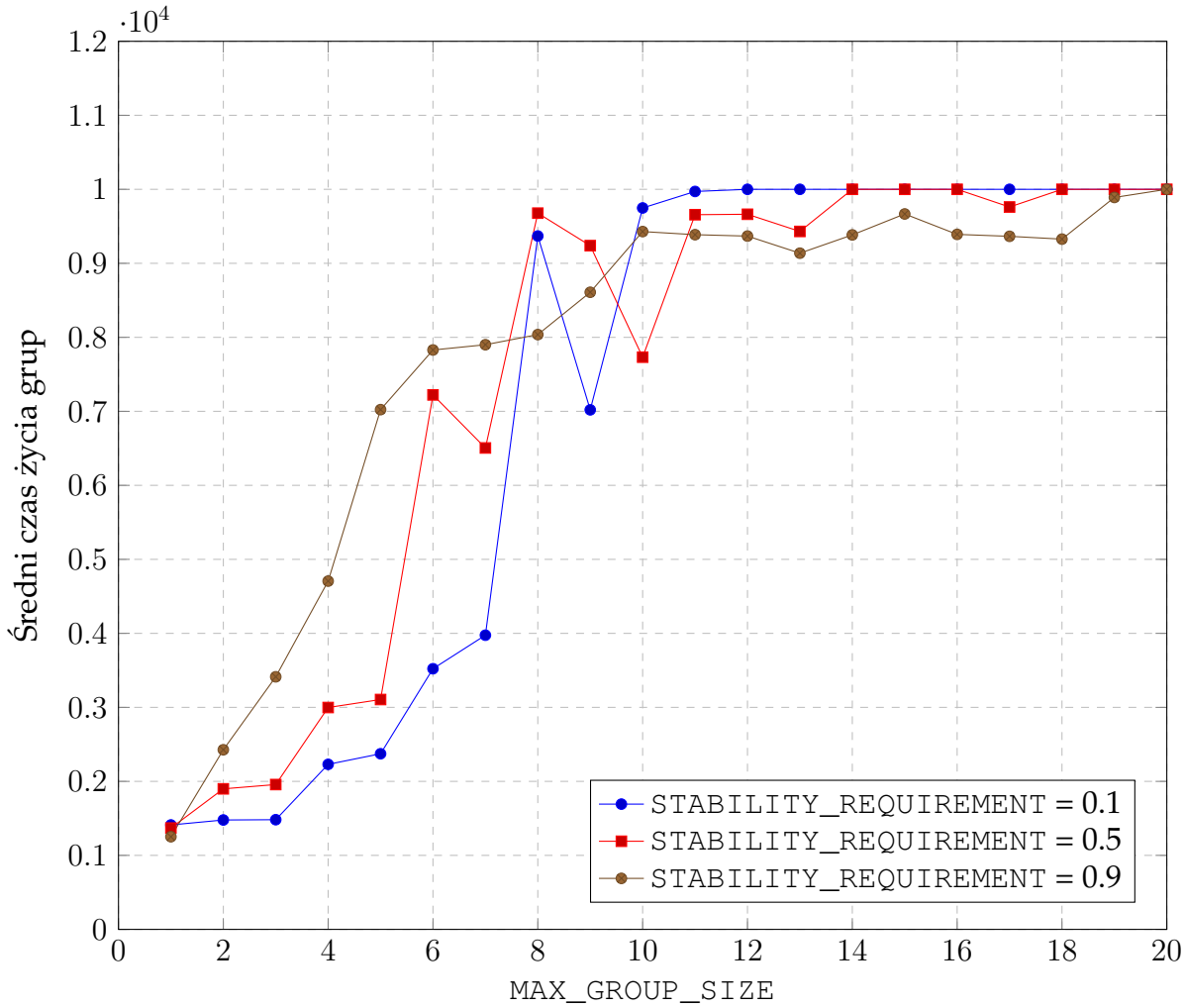
Wykres 6.2: Wpływ parametru STABILITY_REQUIREMENT wraz z wybranymi wartościami RANDOM_ADD_PROBABILITY na średnią liczność grup.

6.2.2 Średni czas życia grup

Poniżej przedstawiono wyniki testów wpływu parametrów MAX_GROUP_SIZE i STABILITY_REQUIREMENT na średni czas życia grup. Czas życia grupy oznacza liczbę cykli, które minęły od momentu powstania grupy, aż do odłączenia się ostatniego członka grupy.

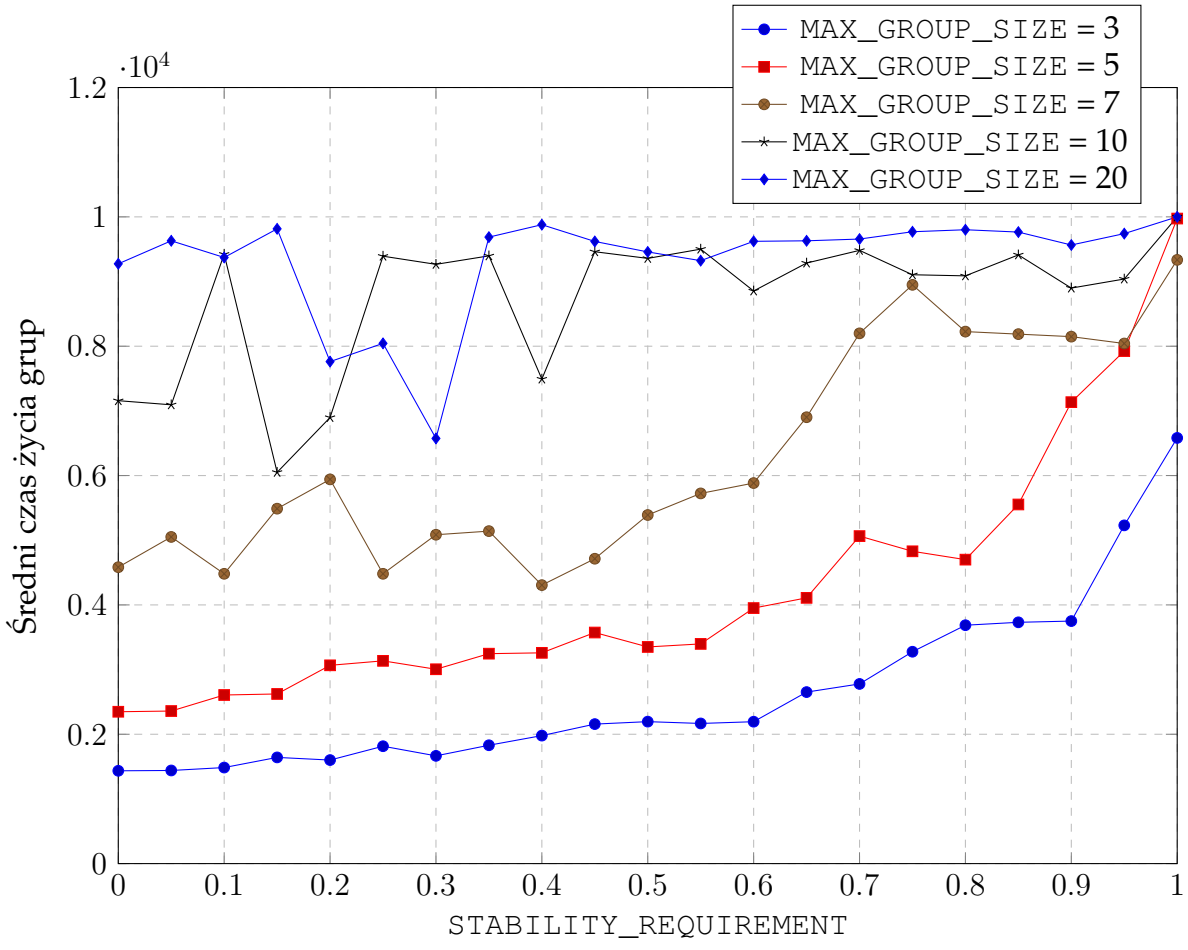
Wykres 6.3 przedstawia wpływ parametru MAX_GROUP_SIZE na średni czas życia grup. Wpływ ten zbadano dla różnych wartości STABILITY_REQUIREMENT. Parametry testu, dla którego powstał wykres 6.3: liczba eksperymentów — 20, początkowa liczba węzłów — 1000, liczba cykli — 10000, prawdopodobieństwo dołącze-

nia nowego węzła w przypadku zmiany w sieci — 0.5. Na wykresie 6.3 widać, że dla MAX_GROUP_SIZE mniejszego niż 8, średni czas życia grup rośnie wraz ze wzrostem MAX_GROUP_SIZE. Ponadto, wzrost ten jest większy dla większych wartości parametrów STABILITY_REQUIREMENT. Dla MAX_GROUP_SIZE większego od 7, średni czas życia grup jest podobny dla różnych wartości parametru STABILITY_REQUIREMENT, i wynosi między 7000 a 10000 – czyli przez większość cykli pojedynczego eksperymentu. Rozbieżność ta wynika z faktu losowości zachowań w sieci (parametr RANDOM klasy RandomDynamicNetwork równy jest 1) oraz losowości stabilności węzła. Z wykresu 6.3 wynika, że maksymalna liczność grupy, dla której odczuwana jest zmiana wartości parametru STABILITY_REQUIREMENT przy symulacji trwającej 10000 cykli — to 7. Wynika to z faktu, iż im większa grupa, tym mniejsza szansa, że wszyscy członkowie odejdą zanim do grupy dołączy nowy węzeł (co skutkuje zakończeniem życia grupy). Maksymalna liczność grupy równa 8, jest wystarczająca, aby grupy żyły średnio przez większość z 10000 cykli.



Wykres 6.3: Wpływ parametru MAX_GROUP_SIZE wraz z wybranymi wartościami STABILITY_REQUIREMENT na średni czas życia grup.

Wykres 6.4 przedstawia wpływ parametru STABILITY_REQUIREMENT na średni czas życia grup. Wpływ ten zbadano dla różnych wartości MAX_GROUP_SIZE. Parametry testu, dla którego powstał wykres 6.4: liczba eksperymentów — 21, początkowa liczba węzłów — 1000, liczba cykli — 10000, prawdopodobieństwo dołączenia nowego węzła w przypadku zmiany w sieci — 0.5. Na wykresie 6.4 widać, że dla maksymalnej liczności grup równej 3, 5 lub 7, wraz z wzrostem parametru STABILITY_REQUIREMENT wzrasta średni czas życia grup — przy czym wzrost ten jest największy dla wysokich wartości STABILITY_REQUIREMENT (od 0.6 do 1). Zależność ta nie jest widoczna dla przypadku, w którym parametr MAX_GROUP_SIZE równy jest 10, lub 20 – wtenczas średni czas życia grup waha się od 6000 do 10000 dla dowolnej wartości STABILITY_REQUIREMENT, a dla STABILITY_REQUIREMENT wyższego od 0.4, średni czas życia grup waha się od 8854 od 10000. Wahania te wynikają z losowości zachowań węzłów w sieci oraz losowości w wyznaczaniu stabilności. Zmiana parametru STABILITY_REQUIREMENT nie wpływa znacząco na średni czas życia grup dla MAX_GROUP_SIZE równego 10, lub 20, ponieważ w tych wypadkach grupy są na tyle liczne, że mała jest szansa, aby w przeciągu 10000 cykli zginęły z powodu odejścia wszystkich członków.



Wykres 6.4: Wpływ parametru STABILITY_REQUIREMENT wraz z wybranymi wartościami MAX_GROUP_SIZE na średni czas życia grup.

6.3 Wnioski

Z powyższych wykresów wynika, że protokół Statycznych Grup zachowuje się zgodnie z oczekiwaniami — średnia liczność grup oraz średni czas życia grup zależą od dobranych parametrów i mogą być dostosowywane do potrzeb. Dla przykładu, jeżeli celem byłoby zapewnienie długiego czasu życia grup, należałoby zwiększyć maksymalną liczność grup — co naturalnie podniosłoby żywotność grup, gdyż grupy byłyby większe, przez co mniej narażone na odpływ wszystkich członków, lub ustawić parametr `STABILITY_REQUIREMENT` na wysoką wartość — wówczas tylko bardzo stabilne węzły tworzyłyby nowe grupy, a więc żywotność grup byłaby większa. Innym celem mogło by być utrzymywanie grup w jak największej liczności — należałoby wtedy ustawić parametr `STABILITY_REQUIREMENT` na dużą wartość (większą niż 0.9). Wówczas zamiast tworzyć nowe grupy, znaczna większość węzłów dołączałoby do grup już istniejących. Lecz jeżeli celem byłoby utrzymywanie liczności grup na średnim poziomie (oscylującym wokół połowy maksymalnej liczności grup), należałoby ustawić parametr `STABILITY_REQUIREMENT` na niższe wartości (mniejsze niż 0.9). Wówczas nowe grupy będą częściej tworzone. Protokół jednak wymusza, aby grupy miały rozmiar większy niż połowa `MAX_GROUP_SIZE` (odpowiedzialny jest za to opisany w paragrafie 5.2.3 warunek w linii 6 na listingu 5.6). Dlatego, jeżeli celem byłoby utrzymywanie liczności grup na niskim poziomie, to nie można by tego osiągnąć ustawiając parametry protokołu — można by to osiągnąć jedynie poprzez specyficzne zachowanie się sieci, tzn. odłączenia od sieci powinny być znacznie liczniejsze, niż dołączenia do sieci (w powyższych testach odpowiedzialny za to jest parametr `RANDOM_ADD_PROBABILITY`).

Rozdział 7

Podsumowanie

Niniejsza praca przedstawiła propozycję protokołu Statycznych Grup, który rozszerza protokół Chord o infrastrukturę dla replikacji, wykorzystując do tego mechanizm grupowania węzłów w grupy. Wyniki przeprowadzonych eksperymentów pokazują, że protokół zachowuje się zgodnie z oczekiwaniami, tzn. przy odpowiednim doborze parametrów, można utrzymać średnią liczność i średni czas życia grup na pożądanym poziomie.

Celem powstania protokołu Statycznych Grup było zapewnienie infrastruktury dla replikacji, nie wnikając w szczegóły dotyczące sposobu uspójniania danych między replikami. Pozostawia to pole do dalszych badań dotyczących replikacji z wykorzystaniem protokołu Statycznych Grup — np. wpływ sposobu uspójniania danych i wielkość grup na ruch komunikacyjny w sieci.

Przedstawiony w tej pracy magisterskiej protokół Statycznych Grup abstrahuje od sposobu określania stabilności węzła. Można by zbadać również wpływ sposobu określania stabilności węzła na zachowanie się grup. Ponadto, węzły w protokole Stabilnych Grup, gdy dostają żądanie o zaproponowanie grupy do dołączenia, zwracają grupę najmniejszą z podzbioru wszystkich grup w sieci. Sposób wyznaczania grupy mógłby być inny. Przykładowo, węzły mogłyby brać pod uwagę stabilność węzła, który wysłał żądanie, i zaproponować mu odpowiednią grupę — taką, aby zachować w grupach różnorodność pod względem stabilności węzłów. Inną polityką mogłoby być proponowanie najbardziej obciążonej grupy, aby zachować zrównoważenie obciążenia między wszystkimi węzłami w sieci.

Jest wiele aspektów, w których protokół Statycznych Grup mógłby zostać rozszerzony. Przykładowo, gdy ostatni węzeł z grupy bezawaryjnie odchodzi z sieci, mógłby przesłać wszystkie dane, za które jest odpowiedzialny i które przechowuje u siebie, do innej grupy. Dzięki temu można by uniknąć utraty części danych.

Wyżej wymienione aspekty sprawiają, iż niniejsza praca magisterska jest tylko początkiem badań, jeżeli chodzi o ewentualne wykorzystanie pomysłu, jakim jest protokół Statycznych Grup, w realnie działających aplikacjach.

Bibliografia

[1] Vidal Martins, *Data Replication in P2P Systems*, Réseaux et télécommunications [cs.NI], Université de Nantes, 2007, Français

[2] Ms. F. Golda Jeyasheeli, L. Rajashree, *Cost Effective File Replication in P2P File Sharing Systems*, 2012 International Conference on Computing, Electronics and Electrical Technologies [ICCEET]

[3] Liran Einav, Chiara Farronato, Jonathan Levin *Peer-to-Peer Markets*, Annual Review of Economics

[4] Samuel P. Fraiberger, Arun Sundararajan *Peer-to-Peer Rental Markets in the Sharing Economy*, Annual Review of Economics

[5] C. Ye, D. M. Chiu, *Peer-to-Peer Replication with Preferences*, Department of Information Engineering, The Chinese University of Hong Kong

[6] J. Paiva, L. Rodrigues, *Policies for Efficient Data Replication in P2P Systems*, INESC-ID, Instituto Superior Técnico, Universidade Técnica

[7] D. Wu, Y. Tian, K.-W. Ng, *Analytical study on improving DHT lookup performance under churn*, 2006, IEEE P2P

[8] C. Blake, R. Rodrigues, *High availability, scalable storage, dynamic peer networks: Pick two*, 2003, USENIX HotOS

[9] A. Pace, V. Quema, V. Schiavoni, *Exploiting node connection regularity for DHT replication*, 2011, IEEE SRDS

[10] I. Woungang, F.-H. Tseng, Y.-H. Lin, L.-D. Chou, M. S. Obaidat, *MR-Chord: Improved Chord Lookup Performance in Structured Mobile P2P Networks*, 2014, IEEE Systems Journal

[11] S. Blond, F. Dessant, E. Merrer, *Finding good partners in availability-aware P2P networks*, SSS, 2009

[12] S. Cherbal, A. Boukerram, A. Boubetra, *RepMChord: A novel replication approach for mobile Chord with reduced traffic overhead*, 2017, Commmunication Systems, Volume 30, Issue 14

[13] T. Shafaat, B. Ahmad, S. Haridi, *Id-replication for structured peer-to-peer systems*, 2012, Euro-Par

[14] N. Shahriar, S. R. Chowdhury, R. Ahmed, *Availability in P2P based online social networks*, 2017, 4th International Conference on Networking, Systems and Security

[15] S. Ktari, M. Zoubert, A. Hecker, H. Labiod, *Performance evaluation of replication strategies in dhts under churn*, 2007, Proc. of the 6th MUM, s.90-97

[16] M. al mojamed, M. Kolberg, *Structured Peer-to-Peer overlay deployment on MANET: A survey*, 2015, Computer networks

[17] N. J. Navimipour, F. S. Milani, *A comprehensive study of the resource discovery techniques in Peer-to-Peer networks*, 2015, Peer-to-Peer Networking and Applications, s. 474-492

[18] J. Tsai, J.-S. Liu, T.-Y. Chang, *Optimality of a Simple Replica Placement Strategy for Chord Peer-to-Peer Networks*, 2017, IEICE TRANSACTIONS on Communications

[19] D. Novak, *Load Balancing in Peer-to-Peer Data Networks*, Masaryk University, Brno, Czech Republic

[20] P. Felber, P. Kropf, E. Shiller, *Survey on Load Balancing in Peer-to-Peer Distributed Hash Tables*, 2014, IEEE Communications Surveys and Tutorials, s. 473-492

[21] Anna Kobusińska, *Systemy Rozprosozne Dużej Skali*, Wykłady, 2017, Instytut Informatyki, Politechnika Poznańska

[22] Jerzy Brzeziński, *Przetwarzanie rozproszone. Mechanizmy rozgłaszania niezawodnego.*, Wykłady, [online] `http://wazniak.mimuw.edu.pl/images/e/ea/Pr-1st-1.1-w12.tresc-kolor.pdf` [dostęp: 08.09.2018 r.]

[23] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, *Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications*, MIT Laboratory for Computer Science

[24] Smruti R. Sarangi, *Distributed Hash Tables, Chord*, Lectures, Department of Computer Science, Indian Institute of Technology, New Delhi, India

[25] P. Knezevic, A. Wombacher, T. Risse, *Enabling High Data Availability in a DHT*, 16th International Workshop on Database and Expert Systems Applications, 2005 (pp. 363-367). [10.1109/DEXA.2005.84] Los Alamitos: IEEE Computer Society Press. DOI: 10.1109/DEXA.2005.84

[26] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, I. Stoica, *Load Balancing in Structured P2P Systems*, Peer-to-Peer Systems II. IPTPS 2003. Lecture Notes in Computer Science, vol 2735. Springer, Berlin, Heidelberg

[27] R. Schollmeier, *A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications*, Proceedings First International Conference on Peer-to-Peer Computing

[28] L. Braubach, A. Pokahr, *Addressing Challenges of Distributed Systems Using Active Components*, Springer Berlin Heidelberg, Intelligent Distributed Computing V, 2012, s. 141–151

[29] J. Varia, *Architecting for the Cloud: Best Practices*, 2011, Amazon Web Services

[30] Javad Taheri , Mohammad Kazem Akbari, *TAC: A Topology-Aware Chord-based Peer-to-Peer Network*, Advanced Information Technologies Lab., Department of Computer Engineering and IT, Amirkabir University of Technology, Tehran, Iran

[31] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, Thomas E. Anderson, *Scalable consistency in Scatter*, SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles

[32] J. Paiva, J. Leitao, L. Rodrigues, *Rollerchain: A DHT for Efficient Replication*, Proceedings — IEEE 12th International Symposium on Network Computing and Applications, NCA 2013. 17-24. 10.1109/NCA.2013.29

[33] *Gtk-Gnutella*, [online] `https://sourceforge.net/projects/gtk-gnutella/` [dostęp: 08.09.2018 r.]

[34] *Kazaa*, [online] `https://computer.howstuffworks.com/kazaa3.htm` [dostęp: 08.09.2018 r.]

[35] *FreeHaven*, [online] `https://www.freehaven.net/` [dostęp: 08.09.2018 r.]

[36] *Pastry*, [online] `https://www.freepastry.org/` [dostęp: 08.09.2018 r.]

[37] A. Rowstron, P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, Springer Berlin Heidelberg, Middleware 2001, s. 329–350

[38] Ryan Jay Huebsch, *PIER: Internet Scale P2P Query Processing with Distributed Hash Tables*, Electrical Engineering and Computer Sciences University of California at Berkeley

[39] *Edutella*, [online] `https://sourceforge.net/projects/edutella/` [dostęp: 08.09.2018 r.]

[40] Navaneeth Krishnan, *The Jxta solution to P2P*, [online] `https://www.javaworld.com/article/2075733/enterprise-java/the-jxta-solution-to-p2p.html` [dostęp: 08.09.2018 r.]

[41] P. Maymounkov, D. Mazieres *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*, Springer Berlin Heidelberg, Peer-to-Peer Systems, s. 53–65

[42] *PeerSim P2P Simulator*, [online] http://peersim.sourceforge.net/ [dostęp: 08.09.2018 r.]