POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA INTERNETOWA

PRACA DYPLOMOWA INŻYNIERSKA

Projekt i implementacja systemu P2P do przechowywania plików

Design and implementation of a P2P file storage system

AUTOR:
Piotr Kozerski

PROWADZĄCY PRACĘ:
Dr inż. Michał Kucharzak

OCENA PRACY:

WROCŁAW, 2020

Spis treści

Spis rysunków

Spis tabel

Rozdział 1

Wprowadzenie

1.1. Geneza pracy

W dzisiejszych czasach standardem jest używanie na co dzień więcej niż jednego komputera osobistego. Przeciętny użytkownik oprócz pracy na komputerze stacjonarnym bądź laptopie, posiada również różnego rodzaju urządzenia mobilne. Każde z nich posiada własną, oddzielną pamięć trwałą, na której przechowywane są wyprodukowane przez użytkownika dane. Z czasem okazuje się, że dane rozrzucone są po wielu miejscach, a dostęp do nich staje się problematyczny: wymaga zapamiętania gdzie znajduje się który plik oraz fizycznego dostępu do nośnika.

Pewnym rozwiązaniem tego problemu jest *synchronizacja* pomiędzy urządzeniami. Pojęcie to oznacza proces wymiany informacji między urządzeniem źródłowym a docelowym, którego celem jest doprowadzenie ich zbiorów danych do tej samej, spójnej postaci. Tematem pracy jest głównie *synchronizacja drzewa plików*. W praktyce oznacza to, że każde z urządzeń posiadałoby w pamięci trwałej folder, którego zawartość byłaby skanowana i porównywana z zawartością jego odpowiednika na innych urządzeniach. W przypadku gdy proces porównywania wykaże różnice w zawartości folderów, przykładowo na jednym z urządzeń pojawiłby się nowy plik, plik ten zostałby przesłany do wszystkich innych urządzeń aby wyrównać różnice.

Cechą szczególną implementowanego w ramach pracy dyplomowej systemu jest jego rozproszony charakter. Opiera się on na modelu *Peer-to-peer* (P2P) zgodnie z którym nie istnieje w sieci centralna jednostka zarządzająca wymianą danych. Zamiast tego każdy uczestnik (węzeł) sieci samodzielnie komunikuje się z innymi i zbiera dane.

1.2. Cel i zakres prac

Celem pracy jest zaprojektowanie i implementacja systemu do przechowywania plików działającego w modelu P2P. W jej ramach zbudowano program na komputery osobiste, który w stosunkowo prosty sposób pozwala użytkownikowi zsynchronizować dane pomiędzy wieloma urządzeniami. Program tworzy sieć urządzeń (węzłów sieci), które komunikują się ze sobą nawzajem, wymieniając się aktualizacjami przechowywanych na nich danych. Na całość projektu składają się trzy elementy: *system budujący sieć P2P*, *system synchronizacji danych* oraz *interfejs użytkownika*.

W ramach *systemu budującego sieć P2P* zaimplementowano autorski protokół komunikacyjny, który pozwala na automatyczne wykrywanie innych uczestników sieci, weryfikowanie ich tożsamości oraz szyfrowanie wymienianych przez nich danych. Użytkownik zobowiązany jest do podania adresu IP jedynie jednego z węzłów sieci, a adresy wszystkich innych zostaną wykryte automatycznie. Program monitoruje komunikację z innymi urządzeniami i w niektórych przypadkach podejmuje samodzielną próbę przywrócenia połączenia.

System synchronizacji danych wykorzystuje *system budujący sieć P2P* jako medium komunikacyjne i za jego pośrednictwem wymienia się informacjami z innymi urządzeniami w sieci. Potrafi on porównywać zawartości folderów (drzewa plików) pomiędzy urządzeniami i transmitować brakujące pliki. Służy do wypracowywania konsensusu w ramach sieci aby wytworzyć jeden, spójny zbiór danych.

Interfejs użytkownika w praktyce jest aplikacją działającą w terminalu. Za jego pośrednictwem użytkownik może wydawać komendy tekstowe i sterować siecią urządzeń, a także sposobem w jaki synchronizują dane.

1.3. Struktura pracy

W rozdziale 2 zawarto wyjaśnienie koncepcji pracy dyplomowej w ujęciu ogólnym, porównując ją do innych rozwiązań dostępnych na rynku. Rozdział 3 zawiera wymagania projektu, technologie użyte do jego zrealizowania oraz ogólny opis zaprojektowanych w ramach pracy mechanizmów koniecznych do osiągnięcia zamierzonego celu. Szczegóły na temat implementacji wraz ze schematami ideowymi zamieszczono w rozdziale 4. W rozdziale 5 można zobaczyć w jaki sposób aplikacja funkcjonuje w praktyce, natomiast w rozdziale 6 zawarto podsumowanie, wnioski oraz pomysły odnośnie dalszej rozbudowy projektu.

Rozdział 2

Istniejące rozwiązania a temat pracy

2.1. Sieci P2P

Peer-to-peer jest modelem komunikacji w sieci komputerowej, w którym wszystkie węzły (hosty) pełnią równorzędną rolę [1]. Każdy z jej członków jest zarówno klientem jak i serwerem: każdy zarówno pobiera jak i udostępnia dane. Wymiana danych w takiej sieci jest prowadzona bezpośrednio między hostami. Sposób ten kontrastuje z najbardziej powszechnym w dzisiejszych czasach modelem topologii gwiazdy, który przewiduje jeden, centralny serwer udostępniający treści i wiele klientów którzy te treści konsumują.

Model P2P, choć istniał już wcześniej, został szeroko spopularyzowany prze serwis *Napster* po roku 1999. Serwis ten umożliwiał dzielenie się plikami MP3. Każdy użytkownik mógł zarówno udostępniać własne pliki jak i korzystać z plików innych. Koncepcja ta funkcjonuje w różnych formach po dziś dzień, m.in. w formie protokołu BitTorrent.

Główną zaletą takich rozwiązań jest ich łatwa skalowalność. O ile w modelu klient-serwer duża liczba użytkowników oznacza zazwyczaj obciążenie serwera i spowolnienie szybkości transmisji, o tyle w sieciach P2P zależność jest niemalże odwrotna. Im więcej członków sieci tym więcej źródeł z którego można pobrać plik, a więc szybsza transmisja.

2.2. Oprogramowanie synchronizujące pliki

Obecnie najpowszechniejszym sposobem synchronizacji zawartości folderów są rozwiązania komercyjne, np. *Dropbox* lub *OneDrive*. Ich zaletą jest prostota użycia jak i stosunkowo duża, dostępna za darmo lub za dopłatą, przestrzeń dyskowa w Internecie. Aby z nich korzystać, użytkownik musi założyć konto na odpowiedniej stronie internetowej, zainstalować aplikację kliencką na kilku urządzeniach oraz wskazać synchronizowany folder. Ich sposób działania zazwyczaj opiera się o model klient-serwer: pliki przesyłane są w centralne miejsce w Internecie a dopiero potem są rozdystrybuowane na inne urządzenia. Czasem proces ten wspomagany jest przez bezpośrednią wymianę danych między klientami, czyli pewną odmianę modelu *P2P*. Przykładowo usługa *Dropbox* oferuje funkcjonalność *LAN sync* [5] która przyśpiesza pobieranie wymieniając się danymi między wieloma urządzeniami użytkownika. Wciąż jednak proces ten wymaga łączności z Internetem, ponadto wymogiem jest funkcjonowanie urządzeń w tych samych sieciach lokalnych.

Istnieje też szeroka gama niekomercyjnych programów, które wymagają od użytkownika samodzielnego skonfigurowania jakiegoś rodzaju zewnętrznego, dostępnego przez Internet dysku

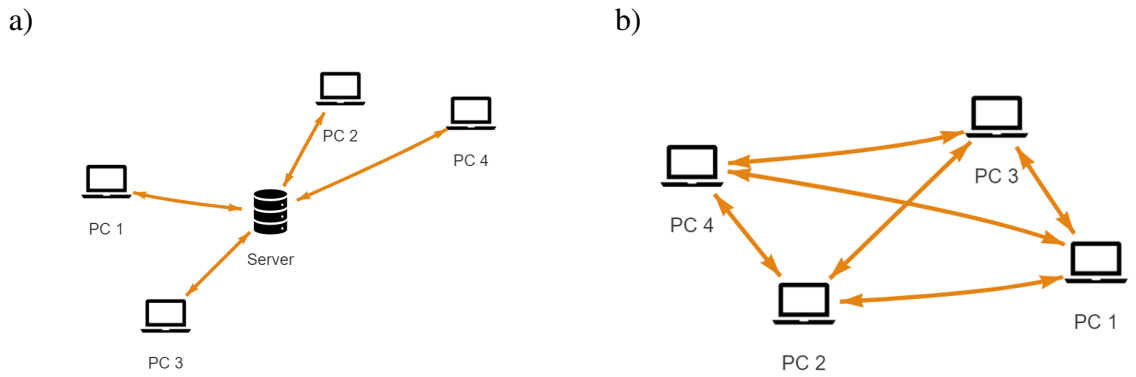
sieciowego. Jednym z przykładów jest *rsync*, który często jest dołączany domyślnie do niektórych dystrybucji systemu Linux. Warto zauważyć że system kontroli wersji *git* też w pewnym sensie pełni rolę systemu synchronizacji plików, jednak jego siła tkwi w analizie zmian w plikach tekstowych i nie jest on szczególnie przydatny w kontekście dużych plików binarnych.

Na szczególną uwagę w kontekście tematyki tej pracy dyplomowej zasługuje *InterPlanetary File System* (IPFS), który rozszerza koncepcję współdzielenia plików w modelu P2P na masową skalę [3]. W ramach tej sieci tworzony jest globalny, współdzielony system plików. Każdy z użytkowników tego systemu może udostępniać wybrane przez siebie pliki oraz metadane pomagające innym węzłom sieci znaleźć szukane przez siebie treści. Jednym z głównych powodów przywoływanych przez twórców, dla których powstał *IPFS* jest ulotność danych w internecie i potrzeba ich łatwej archiwizacji. Wiele treści znika z Internetu bezpowrotnie wraz z centralnym serwerem który je udostępnia, tymczasem rozproszenie ich na wiele urządzeń może ten proces powstrzymać.

2.3. Różnice między sieciami scentralizowanymi a rozproszonymi

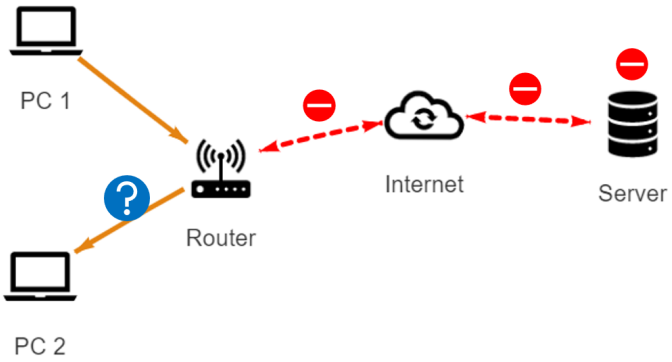
Cechą łączącą najczęściej stosowane obecnie rozwiązania synchronizacji plików jest ich centralizacja. Stosowany w nich model sieci składa się więc z jednego, centralnego punktu (serwera), najczęściej umieszczonego w Internecie, który koordynuje pracę wszystkich podłączonych do niego urządzeń (klientów). Schemat ideowy modelu klient-serwer został pokazany na rysunku 2.1. Takie podejście, choć całkowicie wystarczające dla przeciętnego użytkownika, rodzi wiele problemów:

- całkowita zależność od działania Internetu — brak połączenia z serwerem oznacza brak możliwości synchronizacji. Przykład takiej sytuacji pokazano na rysunku 2.2.
- całkowita zależność od usługodawcy — to na nim spoczywa zapewnienie bezpieczeństwa plików, potencjalna awaria może spowodować utratę danych.
- niepotrzebnie duże zużycie zasobów sieciowych — dane pokonują drogę dłuższą niż to konieczne. Skrajnym przykładem może być sieć urządzeń w sieci lokalnej we Wrocławiu, które wymieniają pliki poprzez serwer w USA, pomimo że możliwe jest przesłanie danych za pośrednictwem switcha w tym samym pokoju.



Rys. 2.1: Rozważane topologie sieci: a) klient-serwer b) model Peer-to-peer

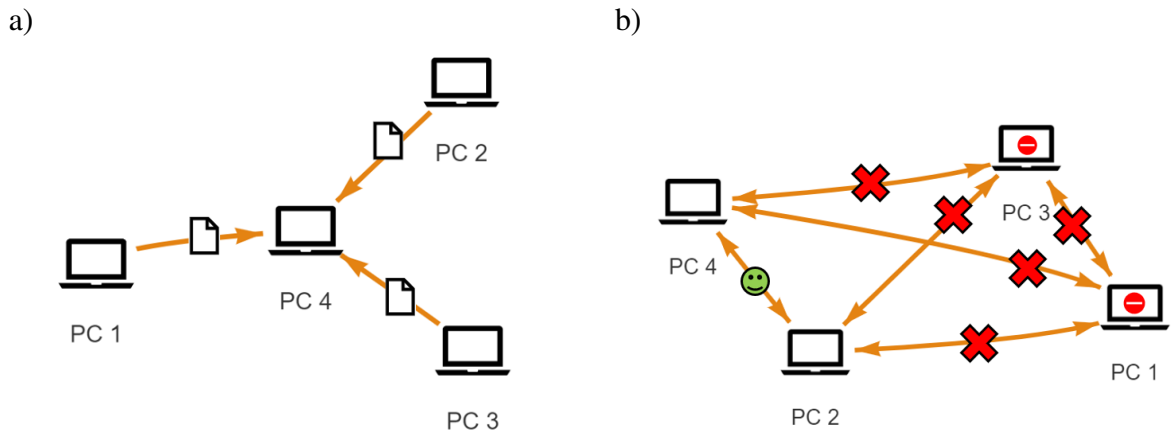
W ramach *Systemu do przechowywania plików* zostało zastosowane odmienne podejście. W zaproponowanym rozwiązaniu każde z urządzeń jest jednocześnie serwerem i klientem, a sieć



Rys. 2.2: Wizualizacja problemu braku dostępu do serwera

urządzeń tworzy topologię siatki. Schemat ideowy takiego modelu widnieje na rysunku 2.1 w podpunkcie b.

Zasada działania systemu synchronizacji plików w modelu P2P jest bliźniaczo podobna do systemu scentralizowanego, z tą jednak różnicą, że o ile w drugim przypadku każdy klient porównuje stan swojego systemu plików z jednym serwerem, tak w pierwszym przypadku wszyscy klienci wymieniają się informacjami pomiędzy sobą. Model ten posiada wiele zalet, między innymi taką, że sieć działa tak długo jak długo istnieje przynajmniej jedno połączenie pomiędzy urządzeniami (rys. 2.3b). W praktyce oznacza to, że w niektórych przypadkach system nie potrzebuje nawet dostępu do Internetu, ponieważ równie skutecznie można połączyć urządzenia kablem Ethernet. Inną zaletą faktu istnienia wielu serwerów udostępniających te same dane jest możliwość pobierania plików z wielu źródeł jednocześnie (rys. 2.3a), co w teorii pozwala na tym szybsze kompletowanie danych im więcej urządzeń partycypuje w sieci.



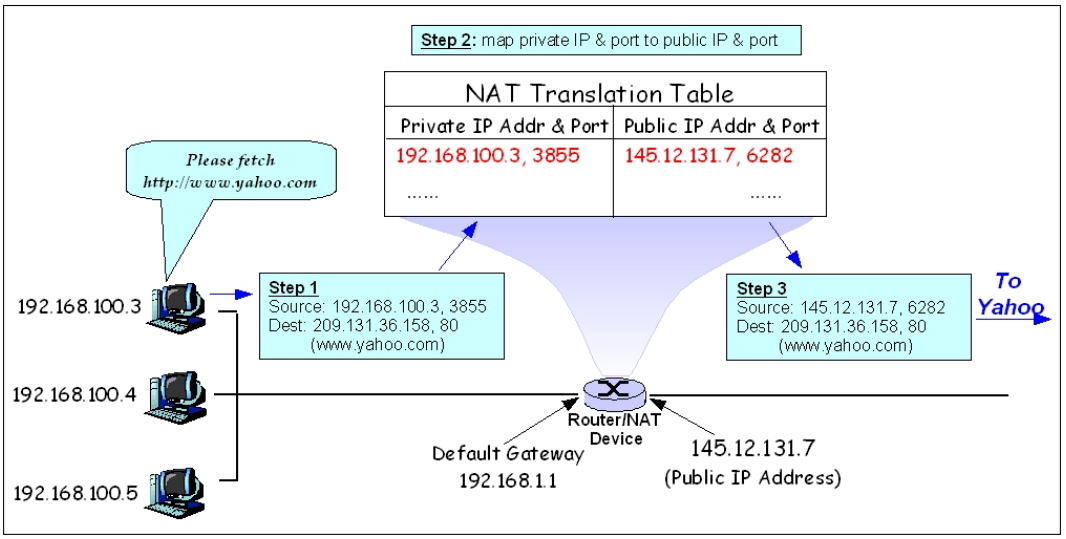
Rys. 2.3: Zalety zastosowania modelu P2P: a) wiele źródeł danych, b) odporność na awarie

Główną wadą rozwiązań zdecentralizowanych jest ograniczone zaufanie do każdego z węzłów sieci. W przypadku modelu z jednym, centralnym serwerem jest jeden potencjalny punkt ataku — serwer. Zastosowanie modelu P2P powoduje, że potencjalny haker może jako cel obrać każdy z węzłów. Zminimalizowanie tego ryzyka było jednym z głównych trudności w tym projekcie.

2.4. Wyzwania przy budowaniu sieci P2P

Podstawową trudnością przy budowaniu sieci *Peer-to-peer* jest nawiązywanie bezpośrednich połączeń między urządzeniami. Okazuje się że architektura Internetu jest dostosowana głównie do komunikacji klient-serwer. Zakłada się więc, że jeśli urządzenie ma funkcjonować jak serwer, potrzebuje publicznego adresu IP, który klient umieszcza jako adres docelowy wysyłanego pakietu.

Większość urządzeń klienckich nie posiada jednak publicznego adresu IP. Zamiast tego funkcjonują pod adresem prywatnym, co czyni je dostępnymi dla innych urządzeń jedynie w ramach sieci lokalnej. Jeżeli urządzenia chcą wysłać dane do Internetu, przekazują je najpierw do routera. Podmienia on adres źródłowy pakietów na swój własny adres IP wraz z dynamicznie wygenerowanym numerem portu, na który przychodzą ewentualne pakiety zwrotne. Technika tłumaczenia adresów IP przez router nosi nazwę *Network Address Translation (NAT)*, została ona zobrazowana na rysunku 2.4. W przypadku sieci P2P stanowi to pewien problem. W sytuacji próby bezpośredniego połączenia dwóch urządzeń z różnych sieci używających NAT, nie istnieje taka kombinacja adresu IP i portu która przekierowałaby pakiet z jednej sieci do drugiej. Oba urządzenia posiadają bowiem tylko prywatne adresy IP, nieistotne poza ich sieciami lokalnymi. Urządzenia takie uzyskują publiczną, istotną dla podmiotów z zewnątrz kombinację IP i portu dopiero w momencie wysłania pakietu za pośrednictwem routera.



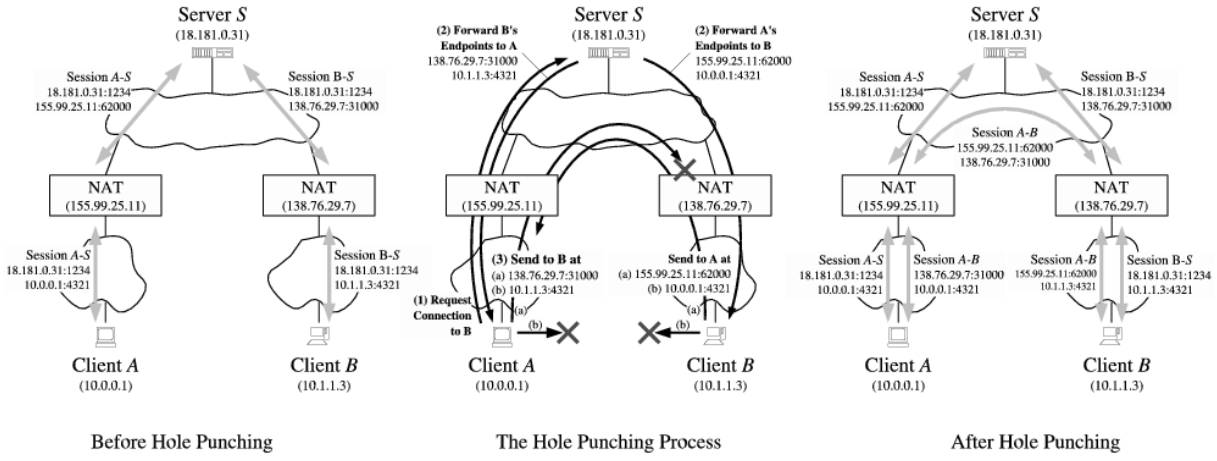
Rys. 2.4: Sposób działania techniki *NAT* [6]

Istnieje kilka rozwiązań tego problemu. Jednym z nich jest ręczna konfiguracja w routerze tzw. przekierowania portów. W większości routerów istnieje możliwość stałego powiązania portu routera z portem jednego z urządzeń w jego sieci. Dzięki temu adres publiczny staje się przewidywalny dla podmiotów z zewnątrz. Wystarczy więc że pakiet przychodzący z Internetu będzie miał adres docelowy równy adresowi routera wraz z tym wcześniej skonfigurowanym portem. Rozwiązanie ma jednak zasadniczą wadę: wymaga od użytkownika dostępu do ustawień routera, co nie zawsze jest możliwe. Potencjalny użytkownik aplikacji synchronizującej pliki wymagałby zapewne od niej działania za pośrednictwem dowolnego punktu dostępowego do sieci Internet, a nie tylko tego nad którym ma kontrolę.

Rozwiązaniem zastosowanym w projekcie *Systemu P2P do przechowywania plików* jest technika o nazwie *UDP Hole Punching* [2]. Technika ta pozwala na nawiązanie komunikacji między dwoma urządzeniami których adresy tłumaczone są przez NAT. Konieczny jest do tego jednak udział publicznie dostępnego serwera, do którego oba urządzenia będą w stanie wysłać pakiety danych. Zostało to zobrazowane na rysunku 2.5. Istnieje serwer S który nawiązał sesję zarówno

z klientem A jak i z klientem B. Wszystkie wiadomości docierające do serwera S z obu klientów mają przetłumaczone adresy IP i porty przez ich lokalny NAT. Oznacza to, że serwer S jest w posiadaniu ich tymczasowych „publicznych" adresów. Klient A może więc wysłać zapytanie o adres B do serwera.

W praktyce uzyskanie adresu drugiego urządzenia jednak nie wystarcza. Często routery nie przepuszczają wiadomości do sieci lokalnej, jeśli nie są one odpowiedzią na wiadomość wysłaną z jej wnętrza. Problem ten można łatwo obejść, jeśli oba urządzenia wysyłają do siebie nawzajem serie wiadomości jednocześnie. Zaprezentowano to na środkowej części rysunku 2.5. W kroku (1) klient A wysyła „prośbę o zapoznanie" do serwera S. Następnie serwer wysyła (2) klientowi A adres klienta B i vice versa klientowi B adres A. W momencie otrzymania takiej wiadomości, oba urządzenia zaczynają do siebie wysyłać w krótkich odstępach czasu serie pakietów. Jak widać w kroku (3), choć wiadomość z A do B została odrzucona przez router B (bo pochodzi ze źródła którego router B nie zna), to router A wpuszcza do sieci wiadomość z B, ponieważ z jego punktu widzenia jest to wiadomość będąca odpowiedzią na wcześniej wysłany pakiet.



Rys. 2.5: Sposób działania techniki *UDP Hole Punching* [2]

Rozdział 3

Projekt systemu współdzielenia plików

3.1. Wymagania

Aby system mógł być przydatny dla użytkownika, musi spełniać kilka podstawowych funkcji. Poniżej przedstawiono wymagania funkcjonalne oraz niefunkcjonalne, mające kluczową rolę przy projektowaniu aplikacji.

3.1.1. Wymagania funkcjonalne

F1: Automatyczna synchronizacja

Użytkownik może wybrać folder znajdujący się w pamięci urządzenia, którego zawartość będzie monitorowana przez program. Program powinien ponadto sam komunikować się z pozostałymi węzami sieci wypracowując konsensus odnośnie stanu drzewa plików.

F2: Automatyczne budowanie sieci

Użytkownik musi wpisać tylko jeden adres IP któregokolwiek z węzłów sieci. Następnie program, po pomyślnej próbie komunikacji z urządzeniem pod podanym adresem, powinien wymienić się z nim informacjami w taki sposób, aby połączyć się ze wszystkimi innymi aktywnymi węzłami sieci.

F3: Kontrola stanu sieci

Użytkownik ma możliwość ręcznej konfiguracji stanu sieci. Powinien więc móc decydować które z urządzeń mogą do niej dołączyć, a które nie.

F4: Możliwość synchronizacji więcej niż jednego folderu

Użytkownik ma możliwość wybrania które foldery synchronizować.

F5: Kontrola synchronizowanych danych

Użytkownik może w każdej chwili zatrzymać bądź wznowić synchronizację wybranych folderów

F6: Umożliwienie działania w sieciach korzystających z NAT

Aplikacja działa na urządzeniu niemającym publicznego adresu IP ani skonfigurowanego portu w routerze

3.1.2. Wymagania niefunkcjonalne

NF1: Niezawodność

System powinien posiadać mechanizmy, które przywrócą połączenie w razie chwilowej awarii łączności. W przypadku błędu, program ma podjąć konieczne czynności w celu przywrócenia działania systemu.

NF2: Bezpieczeństwo

System powinien szyfrować dane przesyłane w sieci Internet w celu zapobiegnięcia niepowołanego dostępu do nich. Ponadto węzły sieci muszą posługiwać się kluczami dostępu, które utrudniłyby atakującemu podszywanie się pod nie. Należy także utworzyć mechanizmy które zapobiegną utracie danych.

NF3: Wieloplatformowość

Im więcej urządzeń będzie w stanie uruchomić instancję tego programu, tym większa korzyść dla użytkownika

NF4: Brak konieczności konfiguracji urządzeń sieciowych

Użytkownik powinien móc korzystać z programu z dowolnego miejsca na świecie poprzez dowolny punkt dostępowy do Internetu

3.2. Zastosowane technologie

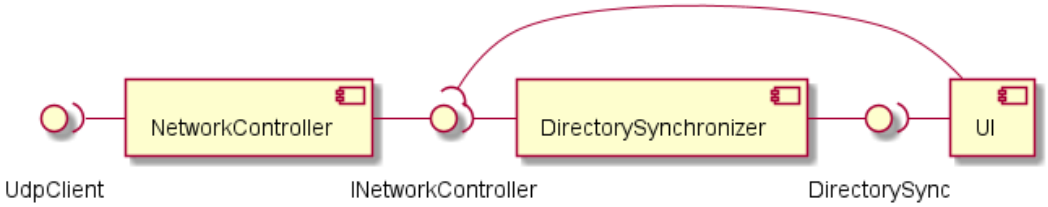
Projekt wykonano w języku C# w trybie zgodności ze specyfikacją *.NET Standard*. W praktyce oznacza to, że napisany w jego ramach kod będzie mógł być uruchomiony bez istotnych modyfikacji na wielu różnych platformach, począwszy od systemu Windows przez dystrybucje Linuxa po system Android.

3.3. Moduły aplikacji

System został zaprojektowany w sposób modułowy. Oznacza to, że składa się on z kilku odseparowanych od siebie części. Każdy moduł (z wyjątkiem interfejsu użytkownika) jest w praktyce osobnym projektem typu *Class library*, który może funkcjonować osobno, jako biblioteka programistyczna.

Relacje między modułami zostały zobrazowane na rysunku 3.1. Pierwszy z nich, *NetworkController* zajmuje się zarządzaniem siecią P2P. Znajdują się tam wszystkie klasy służące do komunikacji z innymi członkami sieci oraz zachowania jej integralności. Korzysta on bezpośrednio z klasy `UdpClient`, części *frameworka .NET*, która pozwala na wysyłanie i odbieranie danych z i do Internetu protokołem UDP. *NetworkController* udostępnia interfejs programistyczny `INetworkController` w którym znajdują się wszystkie funkcje dostępne dla programisty korzystającego z tego modułu jako osobnej biblioteki. Kolejny moduł — *DirectorySynchronizer* — importuje *NetworkController* jako bibliotekę i traktuje ją jak tzw. ”czarną skrzynkę” (ang.

black box) nie mając świadomości jak ona działa, a jedynie otrzymując interfejs który pozwala mu nią zarządzać.



Rys. 3.1: Podział na moduły

Podobnie funkcjonuje *DirectorySynchronizer*. W module tym zebrane zostały wszystkie klasy służące do zarządzania synchronizowanymi folderami. Większość klas jest schowana przed obserwatorem zewnętrznym oprócz klasy `DirectorySync`, która służy za interfejs pozwalający sterować całym modułem. Jest on przekazywany z kolei do *UI*, czyli interfejsu użytkownika poprzez który można osoba korzystająca z programu może wydawać polecenia.

Taka organizacja kodu ma kilka zalet. Przede wszystkim, jako że moduły są schowane za interfejsami, są łatwo wymienne. Przykładowo przekształcenie opisywanego w tej pracy programu do synchronizacji plików w systemie P2P na system scentralizowany polegałoby jedynie na wymianie modułu *NetworkController* na moduł (implementujący interfejs `INetworkController`) tworzący sieć w inny sposób. Ponadto, jako że moduł *NetworkController* nie zawiera żadnej logiki dotyczącej synchronizacji plików a jedynie budowania sieci, może być z powodzeniem wykorzystany w innych projektach wykorzystujących system P2P. Wreszcie jest to ułatwienie dla programisty, ponieważ ma on gwarancję, że modyfikując jeden moduł nie uszkodzi przypadkowo innego.

3.4. Protokół komunikacyjny

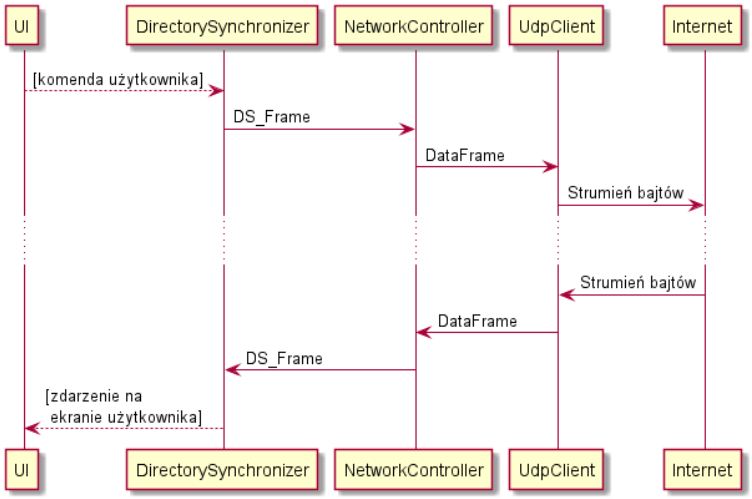
W ramach projektu konieczne było zaprojektowanie własnego protokołu internetowego warstwy aplikacji. Instancje programu komunikują się między sobą w sieci Internet używając wiadomości o ściśle określonej strukturze. Do każdej z nich automatycznie dołączane są metadane informujące odbiorcę jaki moduł i w jaki sposób powinien obsłużyć przychodzącą wiadomość.

Istnieje kilka powodów zaprojektowania protokołu. Przede wszystkim istnieje konieczność przyporządkowania napływających z Internetu danych do odpowiedniego nadawcy. Jako że użytkownik korzystający z aplikacji może w każdej chwili zmienić dostawcę Internetu (np. przełączyć się z sieci komórkowej na WiFi), jego adres IP może także w każdej chwili ulec zmianie. Z tego względu nie można było użyć adresów IP jako identyfikatora urządzeń w sieci P2P i trzeba do każdej wiadomości dołączać informację o identyfikatorze węzła źródłowego. Innym powodem było użycie UDP jako protokołu niższej warstwy. Nie gwarantuje on dostarczenia wiadomości, a jest to warunek konieczny do działania opisywanego systemu. Należało więc zawrzeć informacje o identyfikatorze wiadomości aby móc śledzić czy dotarła ona do celu.

3.4.1. Enkapsulacja

Aby rozwiązać wyżej wymienione problemy, zastosowano tzw. enkapsulację, podobnie jak ma to miejsce w protokołach niższych warstw modelu TCP/IP. To znaczy że każdy moduł, w sposób transparentny dla użytkownika, dołącza do wysyłanych danych własny nagłówek z potrzebnymi metadanymi. Zobrazowane zostało to na rysunku 3.2. Moduł *DirectorySynchronizer* chcąc wysłać do Internetu dane, opakowuje je w ramkę `DS_Frame` zawierającą m.in. identyfikator

powiązanego z tymi danymi folderu. Ramka ta konwertowana jest do tablicy bajtów i przekazana do modułu *NetworkController*, który przed wysłaniem dodaje swoje metadane, przykładowo identyfikator urządzenia źródłowego czy identyfikator wiadomości i pakuje całość w ramkę `DataFrame`. Tak przygotowana tablica bajtów przekazywana jest systemowi, tj. klasie `UdpClient` która z kolei przekaże ją dalej do Internetu.



Rys. 3.2: Przepływ danych przez moduły

Struktura ramki `DataFrame` zobrazowana została w postaci tabeli 3.1. Pierwszy wiersz tabeli zawiera numery bajtów ramki w danych kolumnach. Wiersze poniżej zawierają nazwy poszczególnych informacji przechowywanych w ramce, a szerokość każdej z komórki tabeli odpowiada wielkości tej informacji w bajtach. Aby odczytać lokalizację danej wiadomości w paczce bajtów ramki, należy pomnożyć numer wiersza tabeli z numerem bajtu z pierwszego wiersza. Dla przykładu komórka `SourceNodeId` rozpoczyna się w lokalizacji 0 i reprezentuje 16 bajtów danych, a zaraz po niej (począwszy od bajtu 16.) w ramce pojawia się `MessageType` o wielkości 4 bajtów.

Tab. 3.1: Struktura ramki `DataFrame`

<table>
<tr><th>0 3</th><th>4 7</th><th>8 11</th><th>12 15</th></tr>
<tr><td colspan="4">SourceNodeId</td></tr>
<tr><td>MessageType</td><td>RetransmissionId</td><td>E[1]</td><td>IV</td></tr>
<tr><td colspan="2">IV</td><td>PayloadSize</td><td>Payload</td></tr>
<tr><td colspan="4">Payload</td></tr>
</table>

Analogicznie tworzona jest ramka `DS_Frame`. W ramach jej tworzenia do właściwej wiadomości dołączana jest informacja o identyfikatorze folderu którego dotyczy przesyłana wiadomość. Ramka ta po konwersji do tablicy bajtów umieszczana jest w ramce `DataFrame` w polu `Payload`.

3.4.2. Rodzaje wiadomości modułu *NetworkController*

Jednym z typów danych przechowywanych w ramce `DataFrame` (tabela 3.1) jest czterobajtowa wartość o nazwie `MessageType`. Przechowuje ona informację jakiego typu wiadomość znajduje

[1] ExpectAcknowledge

się we fragmencie ramki oznaczonym jako `Payload`. Daje to możliwość poprawnego odczytania wiadomości i wykonania odpowiedniej instrukcji w aplikacji. Wszystkie wiadomości wraz z ich opisami zostały przedstawione w tabeli 3.2.

Tab. 3.2: Rodzaje wiadomości odczytywane przez moduł *NetworkController*

Nazwa wiadomości	MessageType	Zastosowanie
Ping	0	Sprawdzanie czy urządzenie wciąż jest aktywne. Żąda odpowiedzi w postaci PingResponse
PingResponse	1	Odpowiedź na wiadomość Ping
ReceiveAck	2	Wysyłane w celu poinformowania o otrzymaniu wiadomości
PublicKey	5	Rozpoczyna połączenie między dwoma węzłami. Zawiera klucz publiczny do zaszyfrowania wiadomości.
PrivateKey	6	Odpowiedź na PublicKey. Zawiera klucz prywatny.
AdditionalInfo	7	Zawiera listę innych znanych urządzeń oraz dodatkowe informacje o węźle
AdditionalInfoRequest	8	Prośba o wysłanie wiadomości AdditionalInfo
ResetRequest	9	Prośba o zresetowanie klucza szyfrującego
HolePunchingRequest	10	Prośba o ”zapoznanie” z innym węzłem sieci
HolePunchingResponse	11	Odpowiedż na HolePunchingRequest. Zawiera informacje o innym węźle sieci
ConnectionRestoreRequest	12	Węzeł który był chwilowo nieaktywny, a był wcześniej połączony, może poprosić tą wiadomością o wznowienie połączenia bez ponownej wymiany kluczy
ConnectionRestoreResponse	13	Odpowiedź na ConnectionRestoreRequest. Zgoda na wznowienie połączenia

Jeżeli wartość `MessageType` otrzymanej wiadomości nie odpowiada żadnej z powyższych typów, wiadomość jest przekazywana do modułów wyższego poziomu. W przypadku opisywanej aplikacji, wiadomość otrzymuje moduł *DirectorySynchronizer*.

3.4.3. Rodzaje wiadomości modułu *DirectorySynchronizer*

W momencie gdy *NetworkController* nie rozpozna identyfikatora wiadomości `MessageType`, uznaje że wiadomość nie była adresowana do niego i przekazuje ją do potomnych modułów. Wtedy wiadomość przetwarza *DirectorySynchronizer* szukając wiadomości opisanych w tabeli 3.3.

W przypadku niektórych wiadomości z tabeli 3.3, aby mogły one być przetworzone potrzebna jest dodatkowo informacja o synchronizowanym folderze którego ta wiadomość dotyczy. Z tego względu wiadomości pochodzące z *DirectorySynchronizer* są najpierw pakowane w ramkę `DS_Frame` zawierającą identyfikator folderu, a dopero potem (po przekazaniu do *NetworkController*) w ramkę `DataFrame`.

Tab. 3.3: Rodzaje wiadomości odczytywane przez moduł *DirectorySynchronizer*

Nazwa wiadomości	MessageType	Zastosowanie
SignalChange	100	Sygnalizuje pojedynczą zmianę w strukturze drzewa plików (usunięcie, utworzenie bądź edycja pliku)
FileDataRequestParts	101	Prośba o przesłanie konkretnych fragmentów pliku
FileDataRequestRange	102	Prośba o przesłanie fragmentów pliku z podanego zakresu
FilePart	103	Fragment pliku
RequestLock	104	Zasygnalizuj początek transmisji zmian, chwilowo blokując wysyłanie zmian z zewnętrznego węzła sieci
LockAck	105	Odpowiedź na RequestLock. Zasygnalizowanie że blokada przebiegła pomyślnie.
LockFinish	106	Zakończ blokadę zmian rozpoczętą przez RequestLock
MarkAsInterested	107	Poinformowanie węzła sieci, o chęci otrzymywania aktualizacji zmian dotyczących danego folderu
MarkAsInterestedAck	108	Potwierdzenie, że aktualizacje będą przesyłane
AskForDirectoriesListUpdate	109	Poproś o aktualne dane dotyczące metadanych wszystkich folderów
DirectoriesListResponse	110	Zawiera aktualne dane dotyczące metadanych wszystkich folderów
EFS_Reset	111	Poinformuj zewnątrzne urządzenie o utracie informacji dotyczących posiadanych przez niego plików i konieczności wymiany danych na nowo

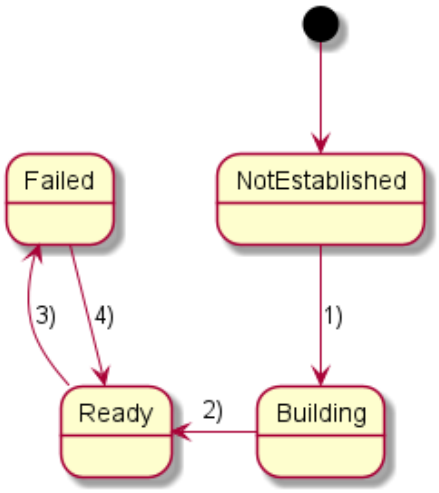
3.4.4. Śledzenie stanu

W ramach protokołu zostało zaimplementowane śledzenie stanu zewnętrznego węzła sieci. Diagram stanów urządzeń został przedstawiony na rysunku 3.3.

W stanie *NotEstablished* znajduje się urządzenie które wiadomo, że istnieje (np. inne urządzenia w sieci poinformowały o jego istnieniu) ale nie wysyłało jeszcze żadnej wiadomości. W momencie przekazania klucza publicznego (wiadomość `PublicKey`) (1), stan zmienia się na *Building* aż do momentu gdy wysłana zostanie wiadomość kończąca ustanawianie połączenia, tzw. `AdditionalInfo` (2). Wtedy urządzenie znajdzie się w stanie *Ready* i pozostanie w nim tak dlugo jak będzie odpowiadać na wiadomości `Ping`. W przeciwnym wypadku (3) zmieni stan na *Failed* z którego może wrócić gdy dotrze z niego jakakolwiek wiadomość (4).

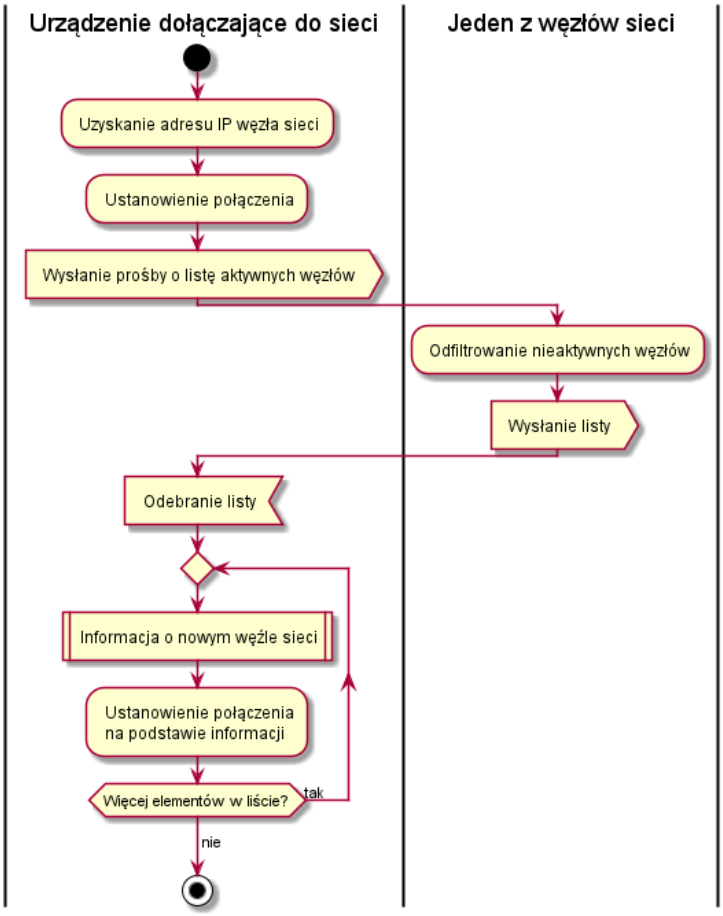
3.5. Wykrywanie węzłów sieci

Sposób wykrywania węzłów sieci został przedstawiony na diagramie 3.4. Dzieje się to w momencie dołączania urządzenia do istniejącej już sieci. W momencie ustanawiania połączenia,



Rys. 3.3: Stany węzłów sieci

obie strony wymieniają się wiedzą na temat wszystkich aktywnych urządzeń. Następnie, jeśli któreś z nich dowie się o nieznanym wcześniej urządzeniu, próbuje się z nim skontaktować.



Rys. 3.4: Algorytm wykrywania węzłów

3.6. Śledzenie listy synchronizowanych folderów

Zgodnie z wymaganiem *F4*, należało zaprojektować system wymiany informacji o synchronizowanych folderach. Nowo dodane urządzenie do sieci powinno móc zdobyć informację o tym jakie foldery może pobrać. Służą do tego wiadomości `AskForDirectoriesListUpdate` oraz `DirectoriesListResponse`. Pierwsza z nich służy jako prośba o wysłanie listy folderów do innego urządzenia, druga jest odpowiedzią na tą pierwszą i zawiera wspomnianą listę.

Istnieje także możliwość wybrania które foldery należy synchronizować. Gdy użytkownik jest zainteresowany pobraniem danego folderu, podaje w aplikacji jego Id oraz lokalizację na dysku w której chciałby umieścić jego zawartość. Wtedy moduł *DirectorySynchronizer* wyśle w jego imieniu wiadomość `MarkAsInterested` do innych użytkowników sieci, żeby poinformować ich o konieczności wysyłania ewentualnych sygnałów zmian.

3.7. Śledzenie zmian w drzewie plików

Projektując system należało wziąć pod uwagę, że będzie on działał w bardzo zmiennym i niestabilnym środowisku. Wpływ mają na to wymienione poniżej czynniki:

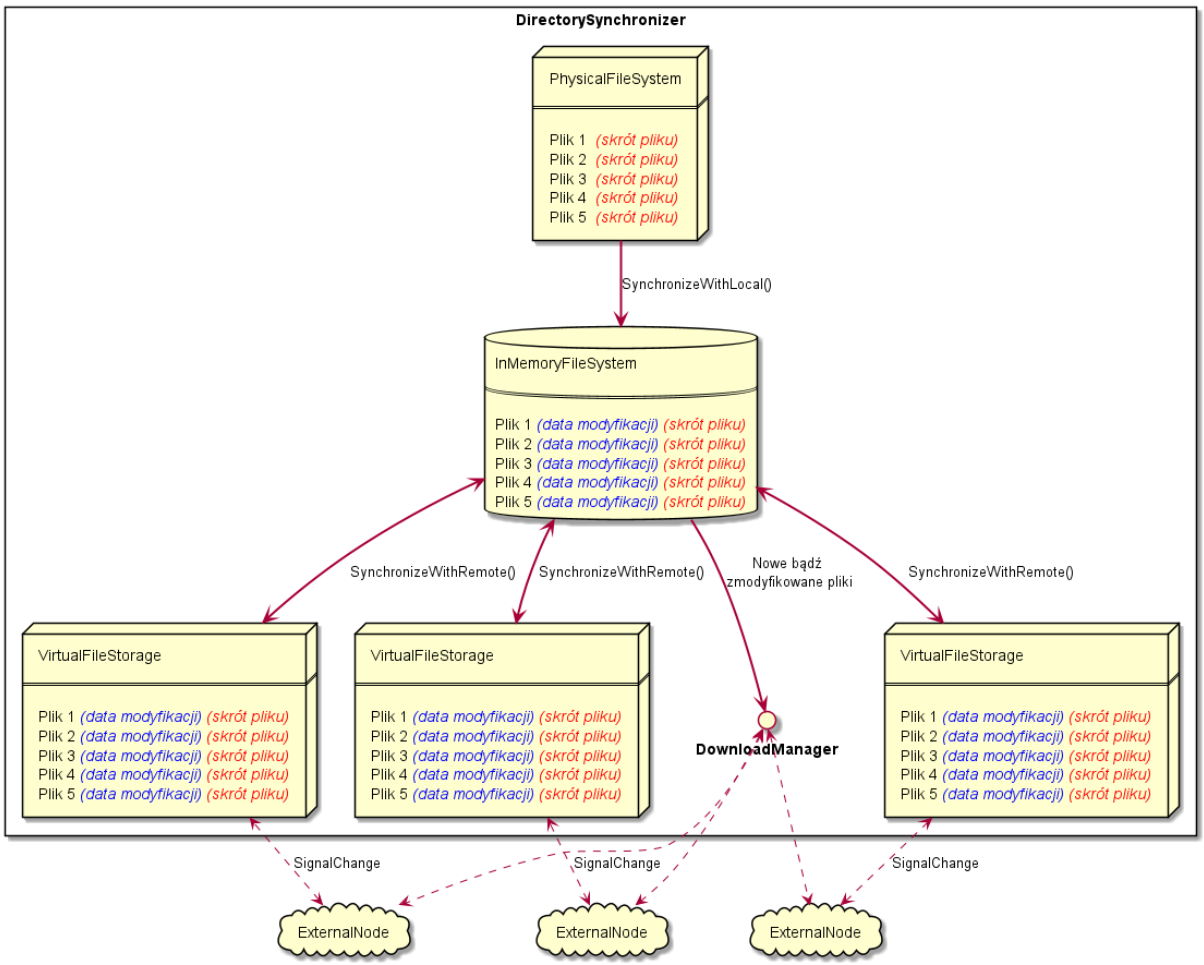
- Potencjalnie dużo sprzecznych danych może napływać z innych węzłów sieci, ponieważ użytkownik może pracować na wielu urządzeniach jednocześnie
- Nie należy ufać zbieranym przez system metadanym na temat plików. Jako że wbudowane w język C# narzędzia korzystają z funkcji systemowych [4], format i sposób ich przechowywania może się różnić w zależności od platformy na której funkcjonuje aplikacja
- Dane na temat drzewa plików mogą nie być aktualne. Nawet gdyby system informował na bieżąco aplikację o nowych plikach i ich modyfikacjach, nie ma gwarancji że aplikacja będzie uruchomiona w każdym momencie działania systemu, a przez to nie ma gwarancji że nie pominie żadnej zmiany.
- Należy założyć że stan wiedzy aplikacji na temat znajdujących się na dysku plików może ulec w każdej chwili zniszczeniu

Pozostaje także problem śledzenia stanu drzewa plików w zewnętrznych węzłach sieci (*ExternalNode*). Żeby synchronizacja mogła mieć miejsce, konieczna jest znajomość metadanych na temat wszystkich plików lokalnych jak i znajdujących się w urządzeniach zewnętrznych. Ze względów wydajnościowych, przesyłanie pełnego stanu drzewa plików za każdym razem gdy uruchamiana jest synchronizacja nie jest możliwe.

Z tego względu zdecydowano się podzielić synchronizację na etapy, a dane z różnych źródeł odseparować od siebie w tzw. *magazynach*. Schemat organizacji przepływu danych znajduje się na rysunku 3.5.

- `InMemoryFileSystem` — jest to centralny dla aplikacji magazyn metadanych o plikach. Nie musi on reprezentować stanu faktycznego, stanowi jedynie informację o wiedzy aplikacji od czasu ostatniej synchronizacji
- `PhysicalFileSystem` — reprezentuje stan faktyczny drzewa plików w momencie zrobienia tzw. "migawki", czyli pobrania listy plików wraz z sumą kontrolną
- `VirtualFileStorage` — reprezentuje konsensus na temat stanu drzewa plików dokonanego we współpracy z zewnętrznym węzłem sieci. Innymi słowy, urządzenie z którym powiązany jest ten magazyn ma w pamięci ten obiekt z identyczną zawartością. Dzięki temu oba urządzenia znają nawzajem swój stan wiedzy i są w stanie określić czy nastąpiła zmiana

Taki układ pozwala na zbieranie na bieżąco zmian pochodzących z Internetu (odnotowywane są one w odpowiednim `VirtualFileStorage`) oraz zmian lokalnych (poprzez



Rys. 3.5: Przepływ informacji o modyfikacjach plików

`PhysicalFileSystem`, bez wpływu na magazyn centralny `InMemoryFileSystem` (który w momencie zmiany mógłby być akurat zajęty przez synchronizację z którymś z magazynów).

W momencie gdy użytkownik aplikacji zażąda rozpoczęcia synchronizacji odpowiednią komendą w interfejsie użytkownika, następuje tzw. *synchronizacja lokalna*, która porównuje stan `PhysicalFileSystem` z `InMemoryFileSystem`. W ten sposób aplikacja wykrywa zmiany które pojawiły się w systemie plików urządzenia na którym działa. Gdy ten krok się zakończy, następuje *synchronizacja zdalna* czyli porównanie zawartości `InMemoryFileSystem` z każdym z magazynów `VirtualFileStorage`. `VirtualFileStorage` reprezentuje stan wiedzy węzła zewnętrznego, więc jest to równoważne z porównaniem drzewa plików z innymi członkami sieci.

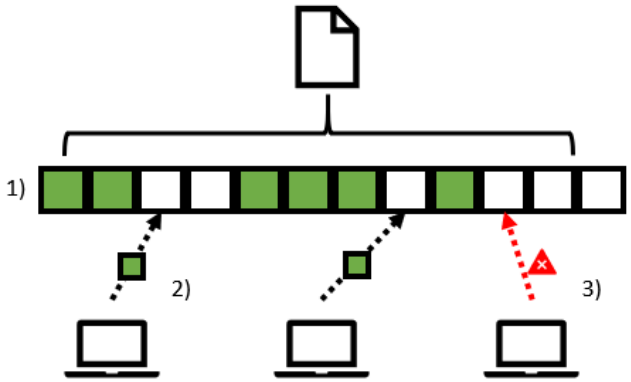
Gdy `VirtualFileStorage` odnotuje zmianę, natychmiast wysyła informację do swojego odpowiednika w węźle zewnętrznym wysyłając do niego wiadomość `SignalChange`. Zmiana ta zostaje zapisana w magazynie dopiero gdy adresat wiadomości potwierdzi że ją odebrał. Taka weryfikacja gwarantuje, że zawartość tego magazynu będzie identyczna na obu urządzeniach. Gdy `InMemoryFileSystem` odnotuje zmianę i jest to zmiana wymagająca pobrania pliku (np. utworzenie lub modyfikacja zawartości), informacja o tym trafia do kolejki w obiekcie `DownloadManager`.

Poszczególne magazyny metadanych o plikach wymieniają się informacjami wysyłając między sobą sygnały. Są to `CreateFile`, `RemoveFile`, `UnremoveFile` (w przypadku gdy plik jest lokalnie usunięty w czasie *t*, ale na zewnętrznym urządzeniu przywrócono go lub zmodyfikowano później, w czasie *t+1*), `UpdateFileId` (w przypadku gdy ten sam plik ma inne Id

na różnych urządzeniach), `MoveFile` i `ModifyFile`. Innymi słowy jedna zmiana odpowiada jednemu sygnałowi.

3.8. Transmisja plików

Sposób w jaki pobierany jest każdy plik został przedstawiony na rysunku 3.6. Plik jest dzielony na *n* fragmentów równej wielkości (z wyjątkiem ostatniej która może być mniejsza). Poszczególne fragmenty mogą być wysyłane przez różne urządzenia (2). Gdy fragment zostanie pobrany i zapisany w pliku, odnotowane jest to w specjalnej tablicy (1). Tablica ta jest przydatna w późniejszej identyfikacji brakujących elementów pliku, które z różnych powodów mogły nie dotrzeć do odbiorcy (3).



Rys. 3.6: Wizualizacja sposobu pobierania plików: 1) tablica śledząca dostępność części 2) transmitujące węzły sieci 3) część nie dociera

Transmisja plików rozpoczyna się w wyniku odebrania wiadomości `FileDataRequestParts` lub `FileDataRequestRange` od innego urządzenia. Pierwsza z nich określa które konkretnie fragmenty pliku powinny zostać wysłane, druga pozwala na określenie zakresu fragmentu. Obie zawierają informację o identyfikatorze pliku oraz wielkości fragmentu.

Gdy istnieje wiele urządzeń w sieci z których można pobrać dany plik, aplikacja rozdystrybuuje fragmenty na jak największą liczbę źródeł. Sytuacja wyglądałaby analogicznie do tej na rysunku 3.6, gdzie każdy z węzłów zajmuje się wysyłaniem przydzielonego mu zakresu fragmentów.

Rozdział 4

Implementacja

4.1. Organizacja kodu

Aplikacja została zaprojektowana w sposób modułowy. Składa się z trzech niezależnych od siebie projektów. Wyróżnione są trzy części aplikacji:

- Moduł odpowiedzialny za komunikację w modelu P2P (*NetworkController*)
- Moduł zarządzający plikami (*DirectorySynchronizer*)
- Interfejs użytkownika (*UI*)

Dwa pierwsze moduły funkcjonują w środowisku programistycznym jako projekty *Class Library*. Projekty tego typu nie mogą funkcjonować samodzielnie, udostępniają jedynie interfejs programistyczny (API) który można używać jako komponent w innych aplikacjach. Są one skonstruowane tak, aby mogły być używane przez innych programistów jako biblioteki.

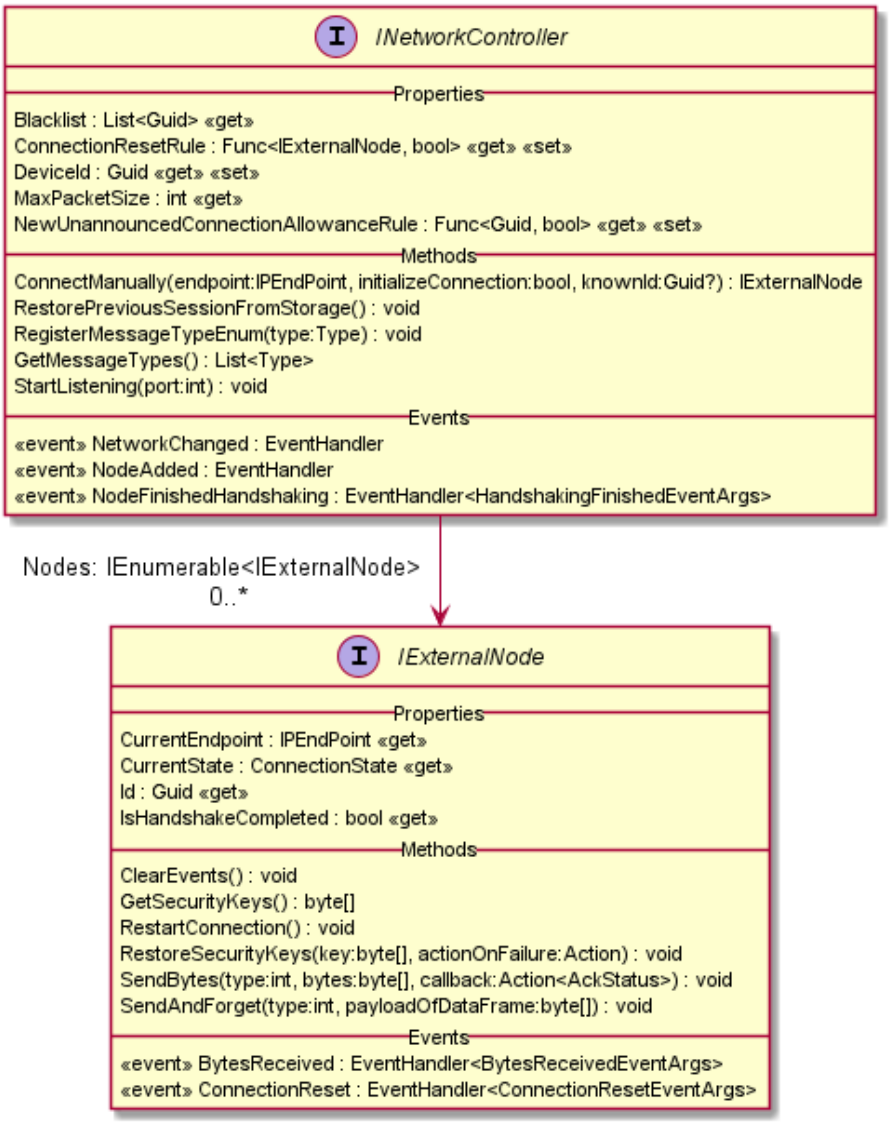
W dalszej części rozdziału opisano poszczególne części tego podziału, sposób ich działania oraz funkcje które spełniają.

4.2. Moduł odpowiedzialny za komunikację

Moduł występuje w kodzie źródłowym pod nazwą *NetworkController*. Pozwala programiście na utworzenie oraz zarządzanie siecią urządzeń typu *każdy z każdym* poprzez dwa interfejsy: `INetworkController` oraz `IExternalNode`. Ich relacja została zaprezentowana na rysunku 4.1.

Zadaniem interfejsu `INetworkController` jest zarządzanie całością sieci. Pozwala na wykonanie następujących czynności:

- Określenie portu na którym urządzenie nasłuchuje wiadomości — metodą `StartListening`
- Określenie identyfikatora urządzenia na którym wykonywany jest program — poprzez zmienną `DeviceId`
- Dodawanie nowych węzłów sieci — poprzez funkcję `ConnectManually`
- Przeglądanie węzłów sieci — poprzez dostęp do listy `Nodes`
- Dodawanie i zarządzanie typami przesyłanych w sieci wiadomości — poprzez funkcje `RegisterMessageTypeEnum` oraz `GetMessageTypes()`
- Przywracanie poprzedniego stanu sieci — funkcją `RestorePreviousSessionFromStorage()`
- Określenie jak ma zachować się system w przypadku konkretnych zdarzeń:
  - Gdy inne urządzenie zarząda resetu klucza szyfrującego — poprzez delegatę `ConnectionResetRule`



Rys. 4.1: Diagram — interfejsy modułu *NetworkController*

  - Gdy otrzymana zostanie prośba o dołączenie do sieci przez nieznane urządzenie — poprzez delegatę `NewUnannouncedConnectionResetRule`
- Określenie maksymalnej wielkości przesyłanego pakietu — zmienna `MaxPacketSize`
- Ponadto udostępnione zostały obiekty typu Event które mogą zostać użyte aby zareagować na konkretne zdarzenia w sieci

Każdy węzeł sieci do którego podłączone jest urządzenie reprezentowane jest poprzez obiekt implementujący interfejs `IExternalNode`. Pozwala on na:

- Pobranie informacji o węźle — jego adresu IP i portu (`CurrentEndpoint`), identyfikatora (`Id`) oraz stanu połączenia z nim (`CurrentState`).
- Wysłanie strumienia bitów
  - Upewniając się że wiadomość dotarła i używając mechanizmów retransmisji jeśli to konieczne — metodą `SendBytes`
  - Wysyłając wiadomość jako zwykły datagram UDP akceptując fakt że wiadomość może nie dojść — metodą `SendAndForget`

- Ręczne ustalenie klucza szyfrującego (zazwyczaj w przypadku przywrócenia połączenia po restarcie aplikacji) — `RestoreSecurityKeys`
- Poproszenie połączonego węzła o restart połączenia i metadanych z nim związanych — `RestartConnection`
- Pobranie kluczy szyfrujących w celu zapisania ich w pamięci stałej — `GetSecurityKeys`
- Usunięcie obserwatorów zdarzeń — `ClearEvents`. Funkcja powiązana ze sposobem w jaki działają zdarzenia w języku C#: brak usunięcia wszystkich obserwatorów może spowodować wyciek pamięci.
- Ponadto udostępnione zostały obiekty typu *Event* które mogą zostać użyte aby zareagować na konkretne zdarzenia w sieci (np. odebranie danych od połączonego węzła)

W listingu 4.1 zamieszczono przykładowy sposób jak wygenerować obiekt `INetworkController`. Do tego celu należy użyć klasy o nazwie `NetworkManagerFactory`. `NetworkManagerFactory` jest klasą typu *fabryka* a jej zadaniem jest umożliwienie konfiguracji modułu *NetworkController* w sposób intuicyjny dla programisty, chowając przed nim szczegóły implementacji modułu.

Listing 4.1: Przykład użycia modułu NetworkController

```
1  // Przykład utworzenia modułu tworzącego sieć P2P
2
3  INetworkController network = new NetworkManagerFactory()
4      .AddLogger(logger)
5      .AddConnectionResetRule((node) =>
6      {
7          return true;
8      })
9      .AddNewUnannouncedConnectionAllowanceRule((guid) =>
10     {
11         return true;
12     })
13     .AddPersistentNodeStorage(
14         new PlainTextFileNodeStorage("nodes.txt"))
15     .Create(new Guid());
16
17 network.StartListening(13000);
```

W linii 4. dodano instancję interfejsu *Logger* która służy do zbierania logów aplikacji. W liniach 5–8 ustalono regułę według której program będzie postępował gdy inny węzeł sieci poprosi o reset połączenia. W tym przypadku program domyślnie zwraca wartość *true*, czyli automatycznie wyraża zgodę. W liniach 9–12 ustalona jest reguła akceptacji nowych węzłów chcących dołączyć do sieci. Zazwyczaj programista będzie chciał w tym miejscu dokonać weryfikacji urządzenia. W podanym przykładzie każde urządzenie jest akceptowane, ponieważ funkcja zwraca zawsze wartość *true*. W liniach 13–14 dodano opcjonalny mechanizm o nazwie `PersistentStorage`. Pozwala on zapamiętać identyfikatory innych węzłów w pamięci stałej, aby po restarcie aplikacji móc przywrócić połączenia z nimi. Dane zostaną zapisane w pliku *nodes.txt*. W linii 15 tworzony jest obiekt `NetworkController` a w linii 17 system rozpoczyna nasłuchiwanie przychodzących wiadomości na wybranym porcie.

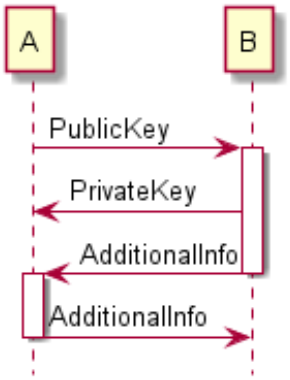
4.2.1. Nawiązywanie połączenia

W przypadku połączenia bezpośredniego, tj. gdy użytkownik aplikacji zna adres IP urządzenia docelowego, komunikacja przebiega w sposób ukazany na rysunku 4.2. W kodzie źródłowym

proces ten określany jest nazwą *Handshake* i służy do bezpiecznej wymiany kluczy szyfrujących. Celem było takie skonstruowanie połączenia, aby potencjalny pośrednik internetowy pomiędzy węzłem A i B nie mógł odczytać wysyłanych przez nie wiadomości. Klucz szyfrujący służy także do uwierzytelnienia wiadomości przychodzących z Internetu: w momencie gdy wiadomości są zaszyfrowane, podszycie się pod jakikolwiek węzeł sieci nie jest możliwe: system spróbuje odszyfrować wiadomość swoim kluczem i w przypadku gdy będzie on nieprawidłowy nie będzie w stanie odczytać wiadomości i odrzuci ją.

Głównym problemem jaki należało rozwiązać na tym etapie była bezpieczna wymiana klucza szyfrującego między obiema stronami: zapobiegnięcie atakowi typu *man-in-the-middle*. Przekazanie klucza bezpośrednio nie byłoby bezpiecznym rozwiązaniem, istnieje bowiem ryzyko przechwycenia klucza przez urządzenia pośredniczące w sieci Internet. Wykorzystano wobec tego tzw. *Kryptografię klucza publicznego*. Węzeł A rozpoczynający połączenie wysyła klucz asymetryczny (`PublicKey`). Do wygenerowania klucza wykorzystano algorytm Rivesta-Shamira-Adlemana (RSA). Po odebraniu wiadomości, węzeł B generuje własny, symetryczny klucz (AES, Advanced Encryption Standard) służący później do szyfrowania wszystkich następnych wiadomości. Klucz ten jest umieszczany w wiadomości *PrivateKey*, a wiadomość szyfrowana jest wysłanym przez A kluczem publicznym.

W momencie gdy węzeł A odczyta wiadomość *PrivateKey*, obie strony komunikacji posiadają unikalny i tajny klucz prywatny służący do szyfrowania danych wychodzących. Od tej pory wszystkie wiadomości, z paroma wyjątkami, są nim szyfrowane. Jeżeli którakolwiek ze stron utraci klucz, będzie musiała rozpocząć proces resetu połączenia (w kodzie źródłowym `ConnectionReset`). To czy taki reset dojdzie do skutku, determinowane jest poprzez wykonanie tzw. delegaty o nazwie `ConnectionResetRule` w interfejsie `INetworkController`.



Rys. 4.2: Nawiązywanie połączenia pomiędzy dwoma urządzeniami

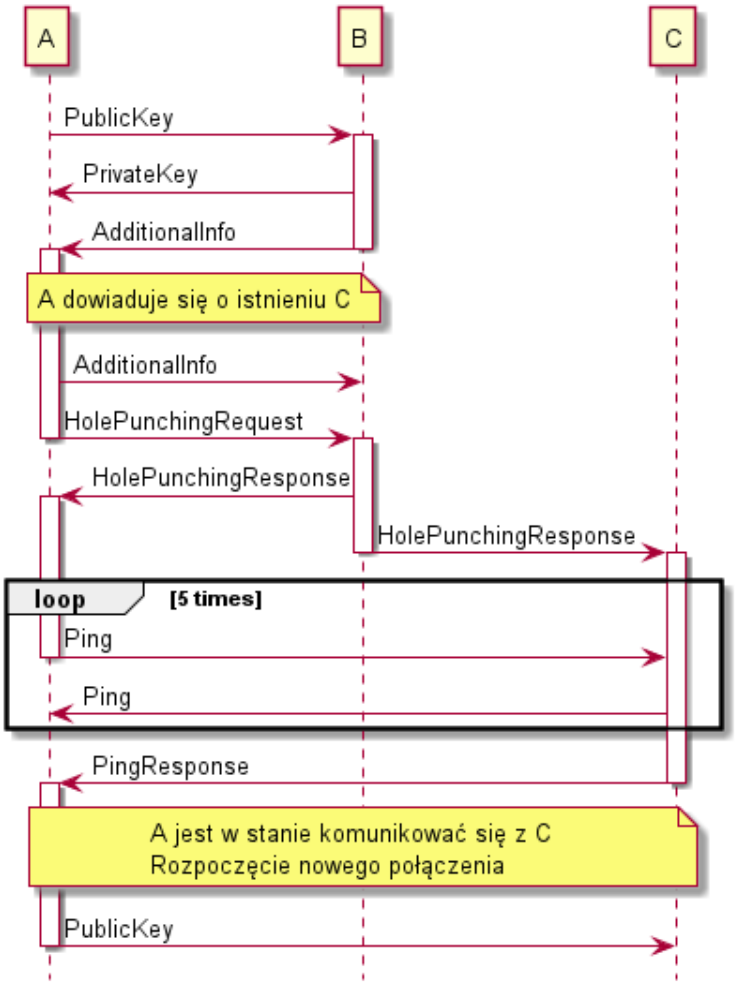
Ostatnim etapem nawiązywania połączenia jest wysłanie przez obie strony wiadomości `AdditionalInfo`. Znajduje się w niej lista identyfikatorów innych węzłów sieci. W ten sposób węzły A i B informują się nawzajem o obecnym stanie sieci. W ramach wiadomości wysyłane są także wartości `ClaimedPrivateIPv4` oraz `ClaimedPrivatePort`. Mogą one być przydatne w procesie nawiązywania połączenia za pośrednictwem innego węzła, opisanego w sekcji 4.2.2.

4.2.2. UDP Hole Punching

W trakcie budowania sieci może zaistnieć możliwość, że węzeł do którego urządzenie użytkownika chce się połączyć nie będzie miał publicznego adresu IP. Taka sytuacja może wystąpić gdy urządzenie docelowe funkcjonuje w prywatnej sieci (problem został szerzej opisany w sekcji 2.4 tego dokumentu). Ponadto istnieje także konieczność umożliwienia nawiązania połączenia na podstawie samego identifikatora węzła, otrzymanego przykładowo w ramach wiadomości

`AdditionalInfo` opisanej w sekcji 4.2.1. W takich sytuacjach uzyskanie łączności jest możliwe za pośrednictwem innego, znanego już węzła.

Na rysunku 4.3 zaprezentowano przykładowy scenariusz w którym urządzenie A łączy się do urządzenia C za pośrednictwem węzła B.



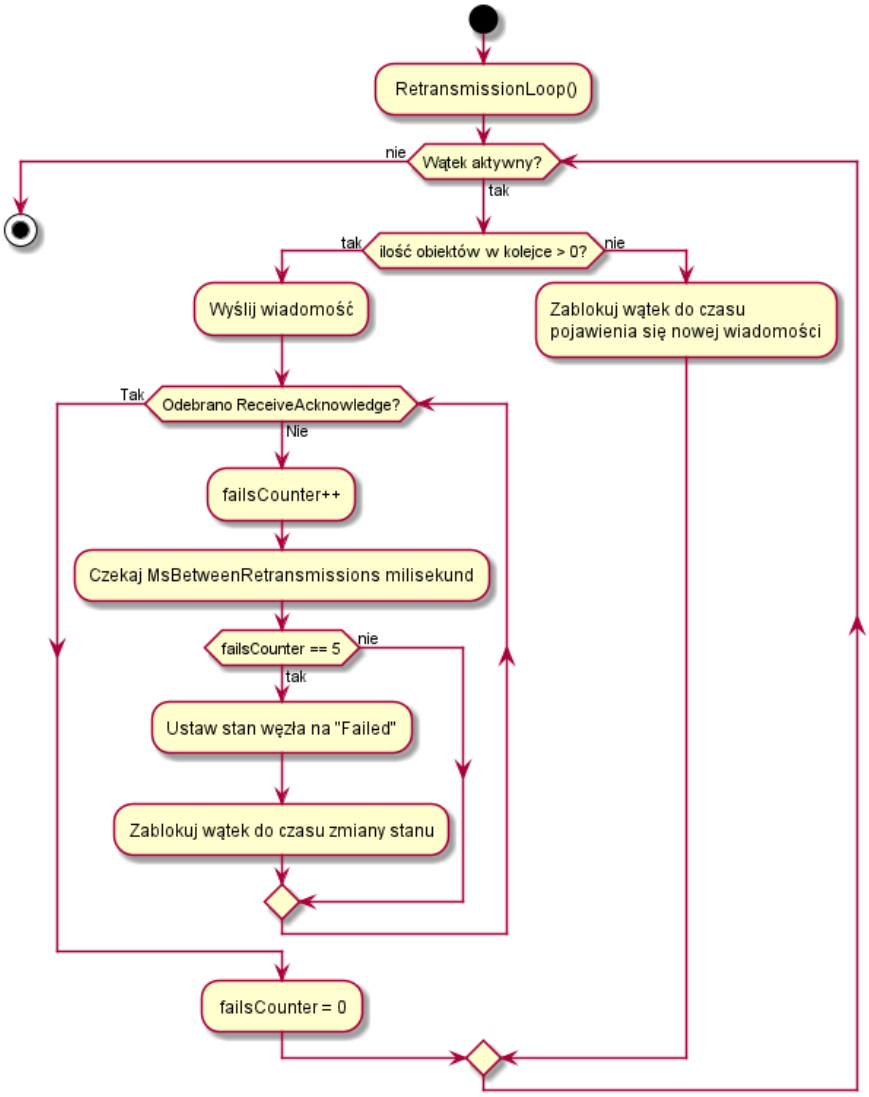
Rys. 4.3: Nawiązywanie połączenia urządzeń A i C poprzez węzeł B

Początkowo następuje znana z sekcji 4.2.1 procedura nawiązywania połączenia z węzłem o znanym adresie IP (węzeł B). Po jej zakończeniu, węzeł A uzyskuje identyfikator nieznanego urządzenia C. Aby się z nim połączyć, wysyła do B ”prośbę o zapoznanie” (`HolePunchingRequest`). W odpowiedzi, w wiadomości `HolePunchingResponse`, węzeł B wysyła do węzła A adresy IP i porty węzła C oraz analogicznie adresy i porty węzła C do A. Wiadomość `HolePunchingResponse` zawiera zawsze po dwa adresy: adres ”publiczny” czyli taki jakiego węzeł wysyłający używa do komunikacji z ogłaszanym węzłem oraz adres ”prywatny” czyli adres który ogłaszany węzeł zaproponował w wiadomości `AdditionalInfo`. Takie rozróżnienie zwiększa szansę na nawiązanie połączenia. Różne adresy mogą być przydatne w zależności od wzajemnego położenia względem siebie w sieci Internet nawiązujących kontakt urządzeń.

Po uzyskaniu adresów IP oba węzły (A i C) zaczynają wysyłać do siebie serię wiadomości `Ping`, po których otrzymaniu natychmiast wysyłają `PingResponse`. W momencie gdy węzeł A (rozpoczynający proces łączenia) otrzyma wiadomość `PingResponse` rozpoczyna standardową procedurę wymiany klucza z rysunku 4.2.

4.2.3. Mechanizm retransmisji

Jako że moduł korzysta z protokołu UDP w celu przesyłania danych, należało wciąć pod uwagę fakt, że wiadomość może nie dotrzeć do odbiorcy. Należało więc zaimplementować mechanizm, który dbałby o dostarczenie wiadomości wysyłając ją ponownie w przypadku gdy odbiorca nie informuje o jej odebraniu. Na rysunku 4.4 umieszczono uproszczony diagram czynności tego procesu.



Rys. 4.4: Retransmisja wiadomości

W przypadku braku uzyskania odpowiedzi od innego węzła w postaci wiadomości `ReceiveAck`, system będzie próbował przesłać wiadomość ponownie. Próby podejmowane są w stałych odstępach czasu, pięciokrotne niepowodzenie powoduje zmianę stanu połączenia z węzłem sieci na *Failed*.

4.3. Moduł odpowiedzialny za synchronizację plików

W kodzie źródłowym występuje pod nazwą *DirectorySynchronizer*. Jego głównym zadaniem jest zarządzanie synchronizowanymi folderami. Dba o wymianę informacji na temat folderów

w sieci, śledzi zmiany w drzewach plików na dysku urządzenia oraz zarządza wysyłaniem i odbieraniem plików z i do innych węzłów w Internecie.

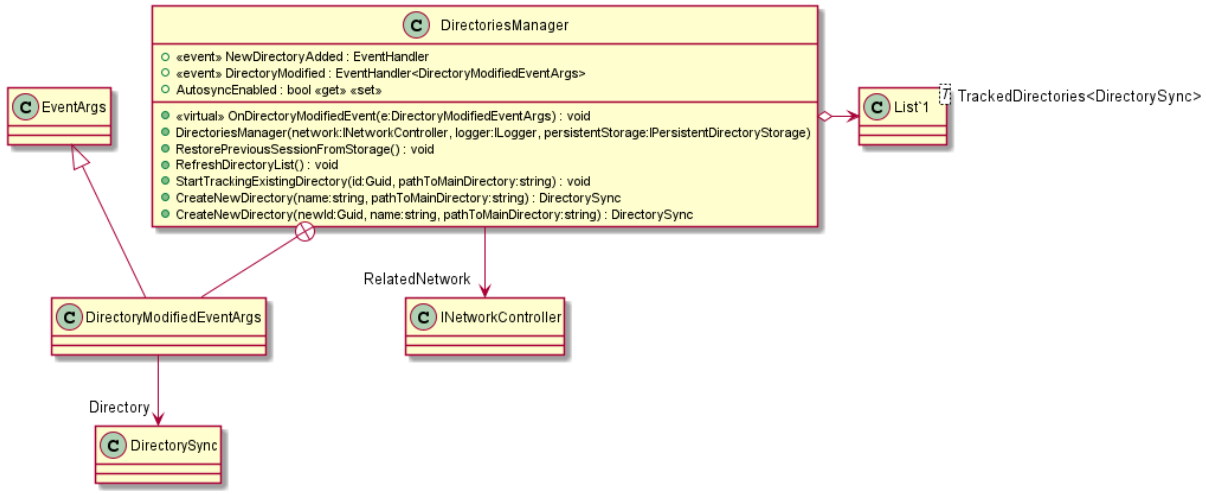
Przykład utworzenia tego modułu zaprezentowano na listingu 4.2. Od programisty stosującego moduł wymaga się jedynie przekazania instancji obiektu zarządzającego siecią implementującego interfejs `INetworkController` (zmienna `network`), obiektu zbierającego logi (zmienna `logger`) oraz opcjonalnie obiektu przechowującego w pamięci trwałej ustawienia modułu. W przykładzie pokazano klasę `PlainTextDirectoryDataStorage` która zapisuje nazwy folderów oraz ich ścieżki w pliku tekstowym o podanej nazwie: *directories.txt*. Zapisane przez nią dane przywracane są wraz startem systemu w linii 7. funkcją `RestorePreviousSessionFromStorage`.

Listing 4.2: Przykład użycia modułu DirectorySynchronizer

```
1 // Przykład utworzenia modułu zarządzającego folderami
2
3 DirectoriesManager dirManager = new DirectoriesManager(network, logger,
4     new PlainTextDirectoryDataStorage(
5         "directories.txt", logger));
6
7 dirManager.RestorePreviousSessionFromStorage();
```

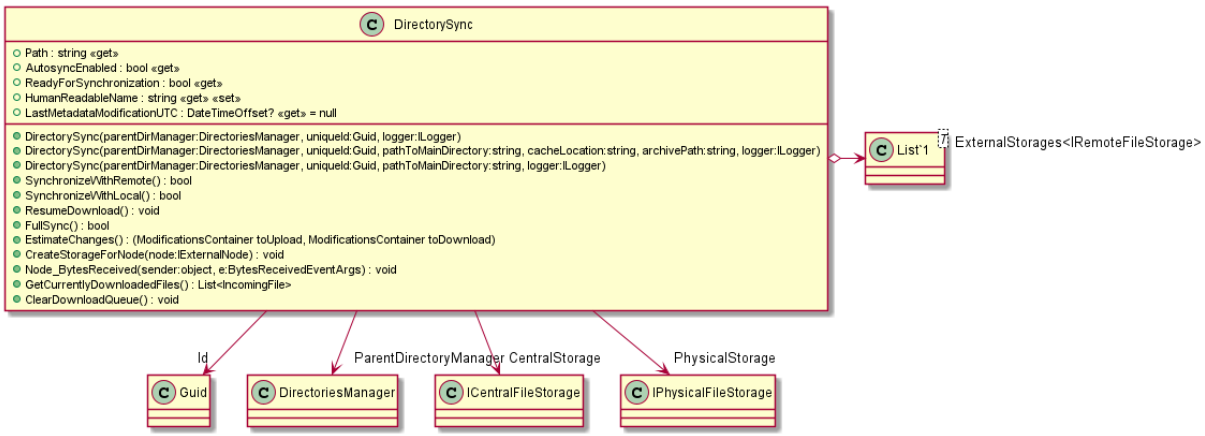
Diagramy kluczowych klas dla tego modułu zostały ukazane na rysunkach 4.5 oraz 4.6.

Zadaniem klasy `DirectoriesManager`, której diagram widnieje na rysunku 4.5, jest śledzenie synchronizowanych folderów. Pozwala ona zarówno na manualne utworzenie nowych (metodą `CreateNewDirectory`) jak i na pobranie folderów utworzonych w innych węzłach sieci (`RefreshDirectoryList`). Wszystkie śledzone foldery przechowywane są w postaci instancji klasy `DirectorySync` obiekcie typu *List* w zmiennej `TrackedDirectories`.



Rys. 4.5: Diagram — klasa `DirectoriesManager`

Klasa `DirectorySync` (rysunek 4.6) zajmuje się jednym z synchronizowanych folderów. Znajduje się tu informacja o fizycznej lokalizacji folderu na dysku urządzenia. Ponadto możliwe jest za jej pomocą rozpoczęcie synchronizacji, oszacowanie różnic pomiędzy stanem obecnym drzewa plików a jego stanem w innych węzłach sieci, a także rozpoczęcie bądź wznowienie procesu pobierania danych.



Rys. 4.6: Diagram — klasa `DirectorySync`

4.3.1. Porównywanie magazynów

Porównywanie zawartości systemów plików między urządzeniami opiera się na wartościach funkcji skrótu każdego z plików oraz na datach ich modyfikacji. Aby system działał prawidłowo, konieczne jest więc prawidłowa konfiguracja zegarów w urządzeniach. Proces decyzyjny, któremu podlega każdy rekord w magazynie danych o plikach, został przedstawiony w postaci diagramu aktywności na rysunku 4.7. W skrócie, w przypadku każdego pliku w jednym z synchronizowanych magazynów, szukany jest jego odpowiednik w magazynie drugim. Porównywane są następnie ich metadane, w tym np. wartość `LastModificationUTC` w której znajdują się dane na temat daty ostatniej modyfikacji pliku. W zależności od różnych czynników, między magazynami przekazywane są różne sygnały, np. `Created` lub `Modified` informujące magazyny o konieczności wprowadzenia do nich zmian.

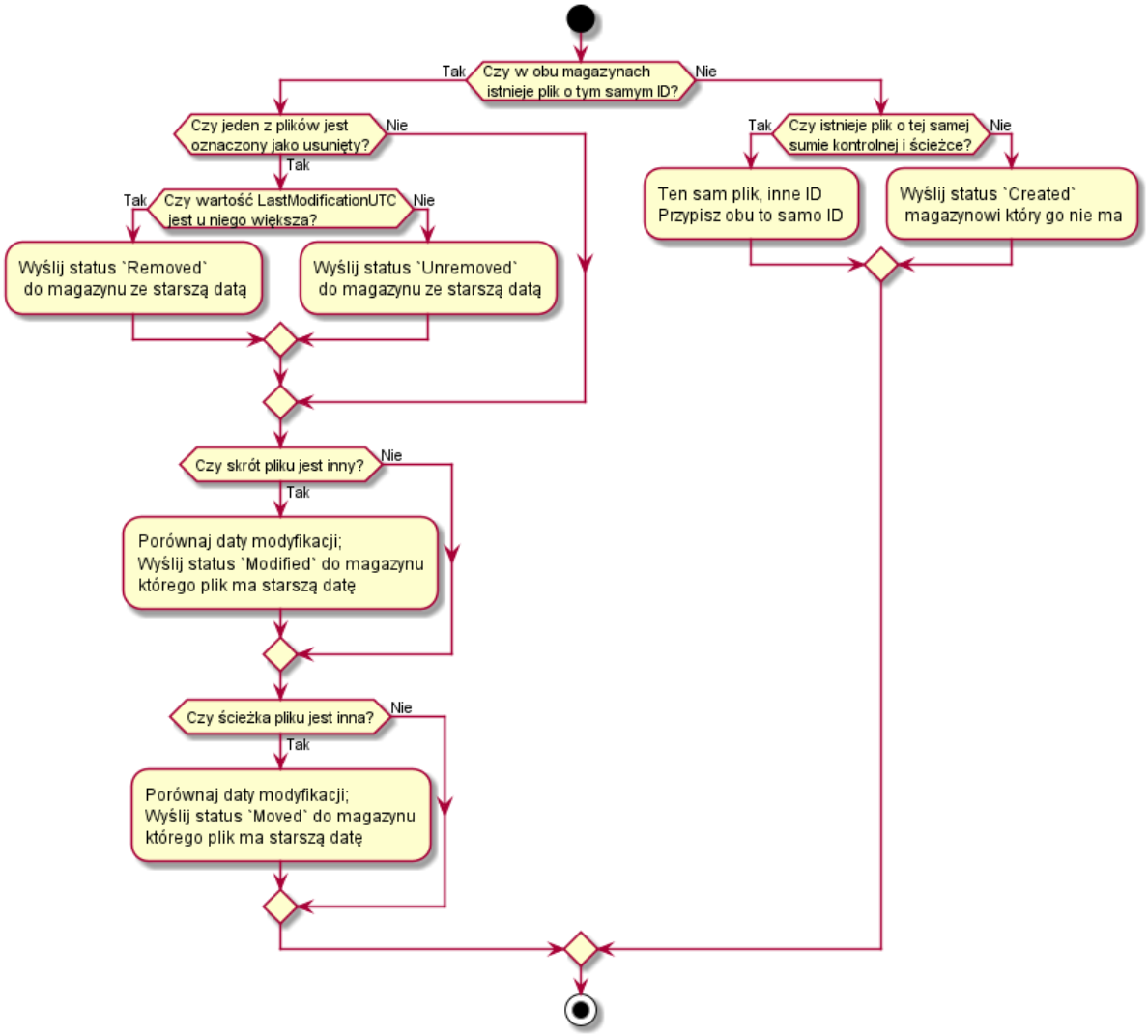
4.3.2. Pobieranie i wysyłanie plików

Sygnały niosące ze sobą potrzebę pobrania plików od innych węzłów sieci (jak np. *Created* lub *Modified*), które docierają do magazynu `InMemoryFileSystem` są następnie przekazywane do klasy `DownloadManager`. Obiekt ten zbiera informacje na temat plików czekających na pobranie. Dla każdego pliku tworzona jest instancja klasy `IncomingFile`. Przechowywane są tam: Id pliku, jego suma kontrolna, ścieżka docelowa oraz lista źródeł z ktöych można ten plik pobrać.

W momencie wywołania funkcji `ResumeDownloading()` w klasie `DownloadManager` wznawiane jest pobieranie wszystkich oczekujących plików. Do tzw. *źródeł* czyli węzłów sieci które posiadają ten plik wysłana jest jedna z dwóch wiadomości:

- `FileDataRequestRange` — żądanie wysłania fragmentu pliku z pewnego zakresu indeksów (przekazywane są wartości `From` oraz `To` definiujące minimalną i maksymalną wartość indeksu)
- `FileDataRequestParts` — żądanie wysłania części pliku o konkretnych indeksach (wysłana jest tablica indeksów których brakuje do skompletowania pliku)

Domyślnie używana jest opcja pierwsza (żądanie zakresu), gdyż w przypadku potrzeby pobrania całego pliku, nie ma sensu wymieniać wszystkich brakujących indeksów. W przypadku gdy pobieranie zostało zakończone, ale niektóre z części pliku z różnych powodów nie dotarły, możliwe jest pobranie ich opcją drugą.



Rys. 4.7: Uproszczony algorytm porównywania różnic w systemach plików

4.3.3. Katalogi *.cache* oraz *.archive*

W trakcie używania programu użytkownik może zauważyć w synchronizowanych folderach samoczynnie pojawiające się katalogi, których nazwa zaczyna się kropką. Mogą one mieć nazwy *.archive* oraz *.cache*. Przechowywane są w nich tymczasowo dane powiązane z procesem synchronizacji folderów w których się znajdują. Ich zawartość jest niewidoczna dla algorytmu porównującego magazyny plików i nie zostanie wysłana do innych węzłów sieci.

W katalogu *.cache* umieszczane są pliki będące w trakcie pobierania. Pojawiają się tam w momencie gdy system wykryje brakujący plik i wyśle do innych węzłów prośbę o przesłanie ich fragmentów. W momencie gdy pobieranie zostanie ukończone, plik ten zostanie przeniesiony we właściwe miejsce w synchronizowanym folderze.

Katalog *.archive* jest natomiast częścią realizacji wymagania niefunkcjonalnego dotyczącego bezpieczeństwa (NF2). Jako że proces synchronizacji działa całkowicie transparentnie dla użytkownika, to znaczy sam pobiera, przemieszcza i usuwa pliki w ramach folderu w którym działa, użytkownik nie jest w stanie śledzić na bieżąco dokonywanych zmian i ewentualnie zareagować gdy coś dzieje się nie po jego myśli. W związku z tym, węzły sieci które otrzymają wiadomość `RemoveFile`, nie usuwają pliku tylko przenoszą go do folderu *.archive*. Daje to możliwość łatwego przywrócenia usuniętych danych.

Rozdział 5

Instrukcja użytkowania

5.1. Interfejs użytkownika

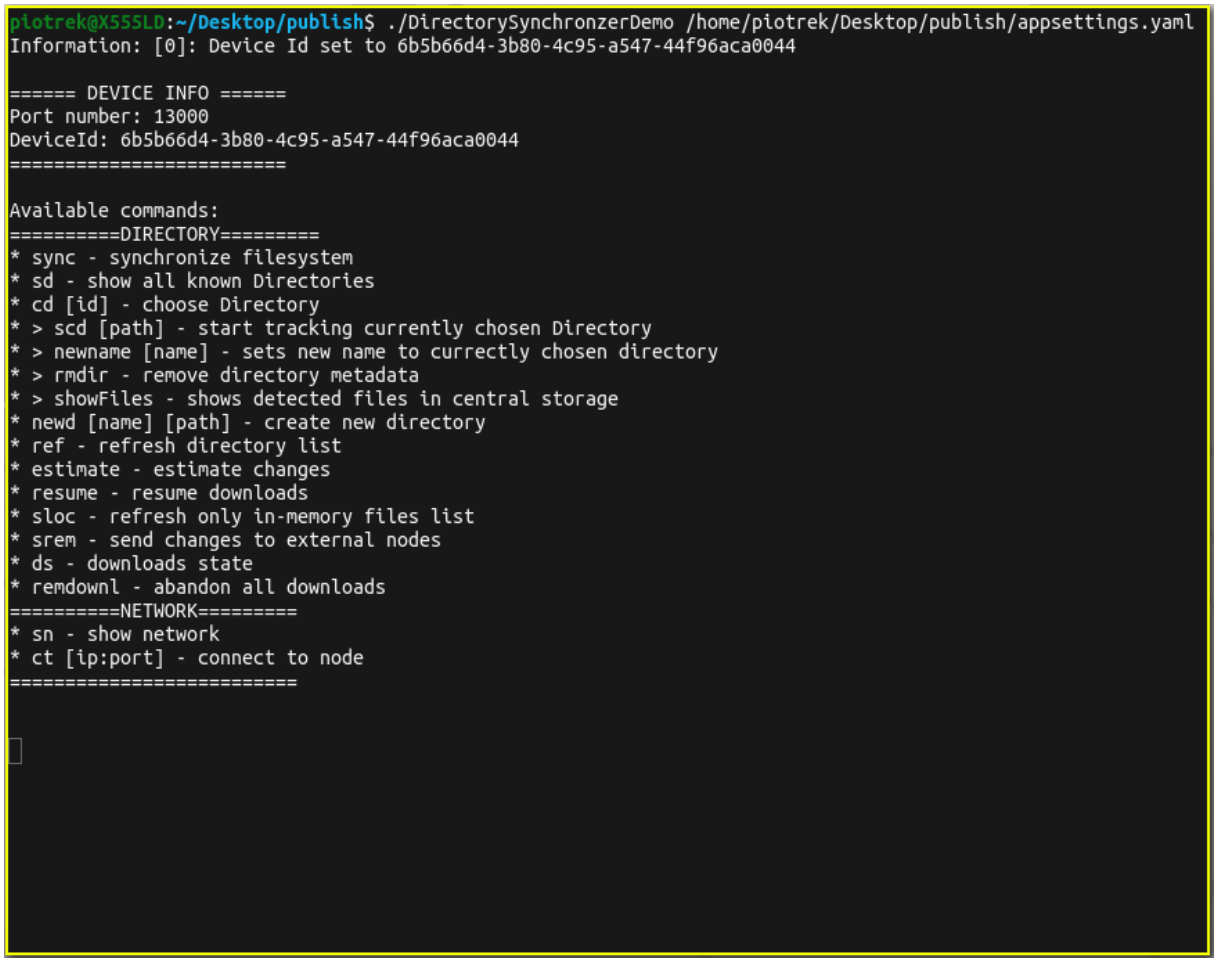
W ramach pracy inżynierskiej zaprojektowano aplikację konsolową. Umożliwia ona zarządzenie systemem synchronizacji plików poprzez interfejs tekstowy. Wygląd interfejsu został zamieszczony na rysunku 5.1. Program uruchomiono poprzez wpisanie jego lokalizacji na dysku (`./DirectorySynchronizerDemo`) natomiast jako argument podano ścieżkę do pliku `appsettings.yaml` zawierającego konfigurację startową programu.

Po uruchomieniu aplikacja zaczyna wypisywać na ekranie informacje dla użytkownika. Zawierają się w tym logi systemowe (na rysunku zaczynające się od sekwencji znaków *Information: [0]*) a także informacje związane z interfejsem użytkownika (na rysunku są to pozostałe wyświetlone linie tekstu). Wyświetlone zostają: port na którym aplikacja nasłuchuje przychodzących z Internetu pakietów, a także przypisanym do niej identyfikatorze. Niżej wypisane są wszystkie możliwe do wykorzystania przez użytkownika komendy.

5.2. Ustawienia startowe

Każdorazowe uruchomienie programu wiąże się z koniecznością konfiguracji sieci P2P, a także ustalenia ścieżek synchronizowanych folderów. Wielokrotne powtarzanie tych czynności może być dla użytkownika uciążliwe, zdecydowano się więc na zautomatyzowanie procesu. Istnieje możliwość utworzenia tzw. ustawień startowych, które zostają wprowadzone w życie w momencie uruchomienia programu.

Jednym ze sposobów jest utworzenie pliku w formacie `*.yaml` i podanie jego lokalizacji w pierwszym argumencie programu (analogicznie do sytuacji na rysunku 5.1). Przykładowa zawartość takiego pliku została przedstawiona na listingu 5.1. Ustalono w nim stały identyfikator urządzenia który będą widzieć inne węzły sieci (`devideGuid`) a także port sieciowy na którym aplikacja będzie nasłuchiwać przychodzących pakietów (`listeningOnPort`). Lista `nodesToCall` pozwala na podanie adresów IP z którymi aplikacja powinna się spróbować połączyć. W `directories` można ustalić listę synchronizowanych folderów: ich identyfikator, nazwę oraz lokalizację w pamięci urządzenia. Opcją `autosync` można określić czy aplikacja powinna automatycznie pobierać wszystkie nowe (wykryte na innych urządzeniach w sieci) pliki (wartość *True*) czy czekać aż użytkownik sam tego zażąda wpisując komendę `sync` (wartość *False*). `persistentNodeStoragePath` oraz `persistentDirectoryStoragePath` to opcjonalne lokalizacje plików w których będzie przechowywany stan sieci oraz stan folderów. Użycie tych opcji pozwoli aplikacji na odtworzenie poprzedniej sesji działania programu po jego restarcie. `minLogLevel` oznacza jak bardzo szczegółowe logi systemowe powinny zostać



Rys. 5.1: Interfejs programu *DirectorySynchronizerDemo*

wyświetlane użytkownikowi na ekranie. Wartość *2* jest wartością domyślną, jako że wyświetla tylko najistotniejsze dane, np. informacje o stanie sieci oraz ewentualne błędy. Wreszcie opcja `allowUnannouncedConnection` pozwala ustalić czy konfigurowana instancja aplikacji powinna pozwalać na dołączenie do sieci innych urządzeń. Ze względów bezpieczeństwa, wartość *True* powinna być tu ustawiona jedynie gdy użytkownik planuje na bieżąco kontrolować stan tego urządzenia.

Listing 5.1: Przykładowa zawartość pliku z konfiguracją startową (appsettings.yaml)

```yaml
startupSettings:
    deviceGuid: 'aaaf77c3-3b80-4c95-a547-44f96aca0044'
    listeningOnPort: 13002

    nodesToCall:
        - '127.0.0.1:13000'
    directories:
        - id: '3e85a3b2-feac-4ada-bb4d-7dbdaa0fbc8b'
          name: 'my_directory'
          physicalPath: '/home/piotrek/Desktop/temp/Sync1'

    autosync: False
    persistentNodeStoragePath: "./keys3.txt"
    persistentDirectoryStoragePath: "./dirs3.txt"
    minLogLevel: 2
    allowUnannouncedConnection: True
```

5.3. Dostępne komendy

Aby wykonać jakąkolwiek czynność, użytkownik wpisuje odpowiednie komendy w oknie terminala. Istnieją trzy rodzaje komend: dotyczące sieci (tabela 5.1), dotyczące modułu synchronizującego foldery (tabela 5.2) oraz dotyczące wybranego folderu (tabela 5.3). W ramach zarządzania siecią użytkownik może wyświetlać jej stan oraz dołączać nowe urządzenia. W przypadku próby połączenia z zewnątrz, gdy nieznane urządzenie wyśle wiadomość `PublicKey`, wyświetli się pytanie czy użytkownik zgadza się na dołączenie go do sieci.

Typowymi krokami jakie użytkownik musi wykonać po pierwszym uruchomieniu aplikacji są dołączenie nowych urządzeń komendą `ct` a następnie utworzenie nowego folderu komendą `newd`. Z kolei na pozostałych urządzeniach użytkownik powinien wybrać folder (komendą `cd`) i określić dla niego lokalizację na dysku (komendą `scd`). Od tej chwili system jest gotowy do pracy i na bieżąco będzie śledził zmiany przychodzące z sieci. Żadne zmiany w strukturze plików na dysku nie będą jednak miały miejsca dopóki użytkownik nie wyda polecenia synchronizacji (`sync`) lub nie zmieni w ustawieniach startowych opcji `autoSync` na wartość *True*.

Tab. 5.1: Komendy pozwalające zarządzać siecią

Komenda	Zastosowanie
sn	Drukuje listę urządzeń podłączonych do sieci P2P
ct [ip:port]	Próbuje nawiązać połączenie z urządzeniem pod podanym adresem i portem

Tab. 5.2: Komendy pozwalające na zarządzanie aplikacją

Komenda	Zastosowanie
sync	Rozpoczyna proces synchronizacji. Lokalna zawartość folderu zostanie porównana z zawartością na innych urządzeniach. Brakujące pliki zostaną pobrane.
sd	Wyświetla wszystkie śledzone foldery
cd [id]	Pozwala wybrać jeden z folderów w celu dalszej konfiguracji. W miejsce `[id]` powinien zostać wstawiony indeks folderu widoczny po użyciu komendy `sd`
newd [name] [path]	Rozpoczyna śledzenie folderu w lokalizacji `[path]`. Użytkownik może nadać mu dowolną nazwę podając ją w miejsce `[name]`
ref	Odświeża listę dostępnych folderów. W tym celu wysyłaja prośbę o nią do innych węzłów sieci
estimate	Drukuje na ekranie informację o ilości zmian jakie będą dokonane po wywołaniu komendy `sync`
resume	Pozwala wznowić pobieranie plików jeśli z jakiegoś powodu zostało ono przerwane
sloc	Synchronizuje listę plików tylko z zawartością lokalną (nie wysyłając tych zmian do Internetu)
srem	Synchronizuje stan wiedzy aplikacji o plikach z innymi węzłami sieci (pomijając synchronizację lokalną)
ds	Drukuje listę obecnie pobieranych plików
remdownl	Pozwala zatrzymać pobieranie plików. Pobieranie zostanie wznowione przy następnej synchronizacji komendą `sync`.

Tab. 5.3: Komendy pozwalające na zarządzanie pojedynczym folderem

Komenda	Zastosowanie
scd [path]	Aktywuje folder o którym informacja nadeszła z innego węzła. Od tego momentu zacznie on pobierać dane z Internetu i zapisywać je w lokalizacji `[path]`
newname [name]	Zmienia nazwę folderu na podaną w miejscu `[name]`. Informacja ta jest rozgłaszana do innych węzłów sieci.
rmdir	Przestaje śledzić wybrany folder i usuwa informacje o nim. Będzie on jednak wciąż widnieć po użyciu komendy `sd` jeśli będzie on istniał na innych urządzeniach w sieci.
showFiles	Drukuje pełną listę plików w ramach folderu

Rozdział 6

Podsumowanie

6.1. Wykonane prace

Realizacja *systemu P2P do przechowywania plików* stanowiła spore wyzwanie zarówno w zakresie planowania jego architektury jak i samej implementacji w kodzie źródłowym. Osiągnięcie zamierzonego efektu wymagało wgłębienia się w szczegóły działania protokołów sieci Internet jak i modelu sieci *Peer-to-peer*. Przetwarzanie danych w tak zmiennym i nieprzewidywalnym środowisku wymusiło wykorzystanie wielu technik zarządzania wątkami i współdzielonymi przez nie zasobami.

W ramach projektu udało się zrealizować wszystkie cele wymienione w rozdziale 3. W momencie pisania niniejszej pracy istnieje już w pełni funkcjonalny system wymiany plików między urządzeniami. W trakcie testów udało się z powodzeniem połączyć kilka urządzeń funkcjonujących w różnych sieciach i przetransmitować między nimi zbiór plików.

6.2. Możliwości dalszej rozbudowy

Pomimo realizacji wymagań funkcjonalnych, aplikacja nie jest jeszcze gotowa do publikacji i codziennego użytku. Testy wykazały słabą odporność systemu na działanie pod dużym obciążeniem: pewne jego części wymagają optymalizacji. Wciąż czasami zdarzają się błędy, które skutkują koniecznością ponownego uruchomienia aplikacji. Konieczne jest też solidne przetestowanie systemu pod względem bezpieczeństwa, aby być pewnym że nie dojdzie nigdy do wycieku danych bądź też ich nieoczekiwanej utraty.

Innym mankamentem jest mało intuicyjny dla przeciętnego użytkownika wygląd aplikacji. Należy rozważyć osadzenie systemu w graficznym interfejsie użytkownika. Pewne kroki w tym celu zostały już podjęte: utworzono prototyp aplikacji na system Android, jest on jednak jeszcze w bardzo wczesnej fazie rozwoju. Sam interfejs konsolowy także wymaga dopracowania. Na chwilę obecną wymaga on jawnego podawania komend od użytkownika, jednakże wiele z wykonywanych poprzez komendy operacji może zostać zautomatyzowana, a sam program mógłby funkcjonować jako proces działający w tle.

Warto też skorzystać z faktu modułowej architektury systemu. Jako że moduł budujący sieć P2P jest wydzieloną, logicznie odseparowaną od reszty systemu biblioteką programistyczną, można ją wykorzystać do budowy innych, opartych na modelu P2P zastosowań.

Literatura

[1] J. Buford, H. Yu, E. K. Lua. *P2P Networking and Applications*. Morgan Kaufmann, wydanie 1, 2008.

[2] B. Ford, P. Srisuresh, D. Kegel. Peer-to-Peer Communication Across Network Address Translators. https://bford.info/pub/net/p2pnat/. dostęp dnia 19 listopada 2020.

[3] IPFS. https://ipfs.io/. dostęp dnia 14 listopada 2020.

[4] File.GetCreationTime(String) Method. https://docs.microsoft.com/en-us/dotnet/api/system.io.file.getcreationtime?view=netcore-3.1. dostęp dnia 3 listopada 2020.

[5] What is LAN sync? https://help.dropbox.com/installs-integrations/sync-uploads/lan-sync-overview. dostęp dnia 14 listopada 2020.

[6] Network address translation. https://en.wikipedia.org/wiki/Network_address_translation. dostęp dnia 19 listopada 2020.