TCS-FILESHARE

Sieciowy system wymiany plikami

Autorzy: Michał Dyrek, Patryk Urbański

Spis treści:

1. Wstęp

Celem naszego przedsięwzięcia było zaprojektowanie wydajnej i skalowalnej sieci komputerów zdolnych do swobodnego przesyłu danych. Utworzyliśmy w tym celu aplikację, która pozwala użytkownikowi na połączenie się z innymi użytkownikami, ustalając między nimi strukturalną sieć. Każda osoba może zapytać całą znaną jej sieć w celu sprawdzenia, kto posiada konkretny plik. Jako efekt otrzymuje listę adresów hostów, z których może potem wybrać komputery, od których chciałaby pobrać dane. Maszyny działają niezależnie od siebie jako klienci-serwery, dzięki czemu nie potrzebujemy centralnego ośrodka z superkomputerami, ani bazy danych do przesyłania i przechowywania plików.

2. Wymagania wobec aplikacji

2.1 Wymagania funkcjonalne:

- ❏ Aplikacja powinna być w stanie zapytać inne komputery w sieci o plik i dostać odpowiedź w formie zbioru odnośników do maszyn, które go posiadają.
- ❏ Użytkownik może wybrać pliki, które chce pobrać na swój komputer, a następnie nawiązać bezpośrednie połączenie z odpowiednimi hostami i rozpocząć pobieranie.
- ❏ Modyfikacja sieci - możliwość dodania i usunięcia wierzchołka, zachowując spójny i zrównoważony stan sieci.

2.2 Wymagania pozafunkcjonalne:

- ❏ Sieć pomiędzy komputerami powinna być możliwie lekka w połączenia (każdy host może pamiętać stałą liczbę sąsiadów), ale utrzymywać dobrą wydajność pomimo nieaktywnych użytkowników.
- ❏ Podłączenie do sieci powinno wymagać możliwie minimalnej ilości

danych, w szczególności powinien wystarczyć jeden adres podpiętego już użytkownika.

2.3 Poufność:

- ❏ Używanie klucza prywatnego do identyfikacji użytkowników.
- ❏ Szyfrowanie przesyłanych plików.

2.4 Integralność:

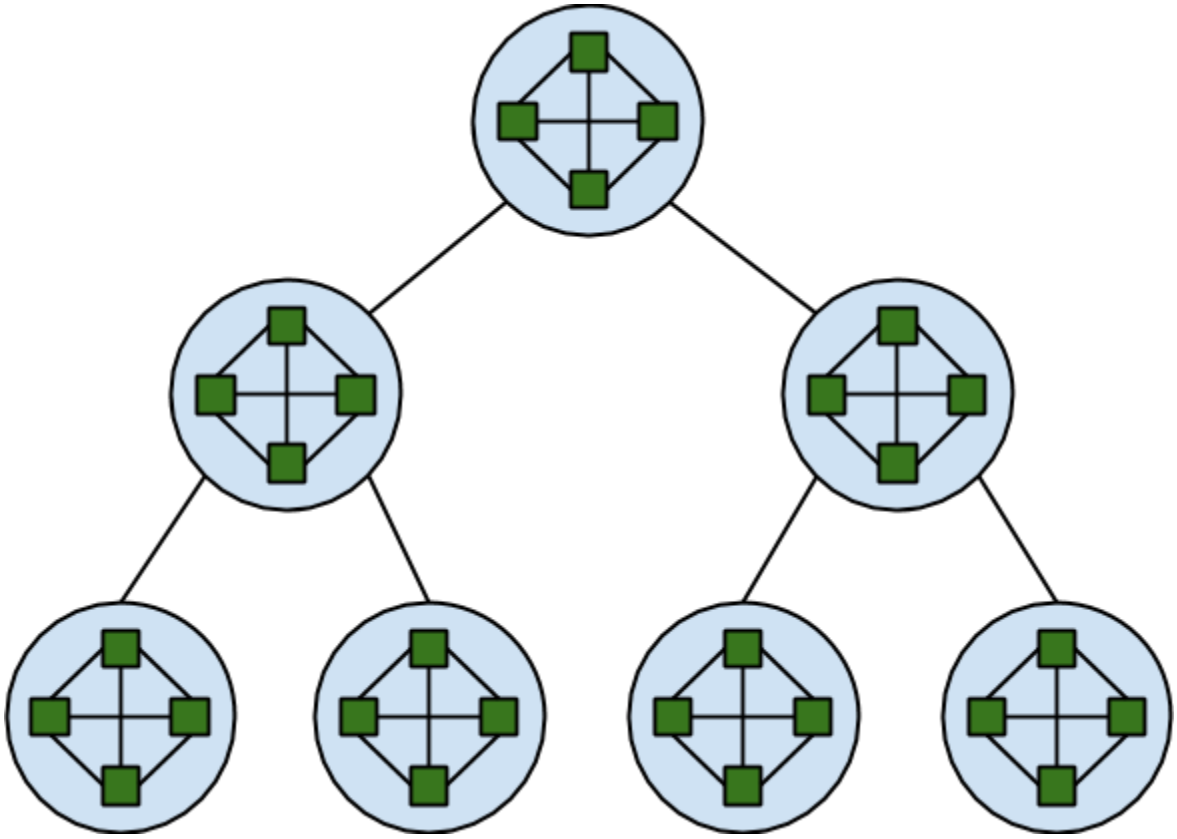
- ❏ Używanie bezpiecznych sposobów przesyłania danych, które dbają o to, by dane nie były zniekształcone (albo żeby rozpoznawać zniekształcenia).

2.5 Dostępność:

- ❏ Każdy host może wymieniać dane z każdym innym, aktywnym i osiągalnym hostem.
- ❏ Konstrukcja sieci zapobiegająca odizolowaniu maszyny od reszty sieci w przypadku braku aktywności niektórych wierzchołków.

3. Realizacja wymagań

3.1 Struktura sieci



Sieci typu *mesh* posiadają bardzo wiele zalet takich jak stabilność, mała liczba przeskoków między wierzchołkami, odporność na nieaktywne punkty itd. Niestety, wymaga ona dużego nakładu pamięciowego oraz kosztów związanych z połączeniami umożliwiającymi bezpośredni przesył danych w dużych grupach. Z drugiej strony istnieją sieci przypominające linię czy owal. Te z kolei cechują się niskim nakładem, jednakże podatne są na nieaktywności oraz wiążą się z dużym dystansem między najodleglejszymi elementami. Środkiem w tych dywagacjach są struktury drzewiaste, które pozwalają na ograniczenie odległości przez logarytm z rozmiaru drzewa zachowując wszystkie zalety linii. Pozostaje problem stabilności, który możemy rozwiązać zamieniając pojedyncze komputery

w wierzchołkach drzewa na małych rozmiarów (przykładowo stałe niewiększe od stałej, równej 50 elementów) podsieci (zwane później *meshami*). Każdy taki mesh ma unikalny numerek. Numerowanie meshy jest analogiczne do numerowania w drzewie binarnym, tzn. korzeń ma numer 1, dla wierzchołka o numerze *k* jego dziećmi są wierzchołki 2k, 2k+1, a rodzicem [k/2]. Do rozszczepienia sieci wymagane by było wtedy, aby żaden z 50 elementów wierzchołka nie był dostępny, co w normalnych przypadkach ma miejsce bardzo rzadko (zakładając 20% na dostępność komputera - około 1:100000 prób). Obrazowy przykład wyglądu sieci dla ograniczenia równego 4 mamy na początku rozdziału.

Pozostaje pewien nieoczywisty problem, który objawia się dopiero przy bardzo dużych sieciach (dopiero przy setkach tysięcy byłoby to odczuwalne), mianowicie wyróżnienie wierzchołków. Pomimo faktu, że protokoły są lekkie i nie przesyłają niepotrzebnych danych, siłą rzeczy niektóre maszyny będą wykonywać większą pracę. Będą łączyły większe grupy komputerów, a zatem, aby zapytanie mogło się dostać do drugiej części drzewa, będzie musiało być przez niego odebrane, parsowane i przesłane dalej. Tutaj wkracza kolejna zaleta naszego schematu - jeśli za każdym razem, gdy z danego mesha chcemy przesłać żądanie będziemy losować przedstawiciela, to średnio zmniejszamy obłożenie elementów takiego wierzchołka o czynnik równy ograniczeniu rozmiaru mesha, zatem podwajając jego rozmiar, połowimy średnie obciążenie komputera. Do reprezentacji mesha służy klasa *model/AddressBlock*, przechowująca zbiór adresów hostów. Jest ona rozszerzeniem Javowej klasy *ArrayList* ze względu na prosty i szybki dostęp oraz modyfikacje elementów. Dodatkowo posiada metodę *getRandom()*. Na początku pinguje ona wszystkie maszyny w meshu z użyciem statycznej metody *common/CheckAvailability.available()* (z timeoutem ustawionym w *common/Constants.ping_timeout*), a następnie zwraca adres losowej spośród aktywnych maszyn. Ta maszyna jest wybierana jako tymczasowy przedstawiciel mesha do obsługi zapytania i tym samym odciąża inne wierzchołki w jej grupie.

3.2 Klient

Program do wysyłania żądań korzysta z jednego z trzech klientów. Przy zapytaniu każdego typu tworzony jest obiekt odpowiedniego klienta obługujący to połączenie:

- *dołączenie do sieci* - gdy użytkownik chce podłączyć się do sieci, podaje adres jednego z komputerów w niej (port *serwera add* oraz IP, opisane w protokole 4.1), a ten mu odpowiada wystarczającymi danymi, aby być w stanie zlokalizować się w nowej sieci.
- *zapytanie o pliki* - użytkownik wprowadza nazwę pliku, który chciałby pobrać, a aplikacja przerabia to na zapytanie zgodne z protokołem. Później rekurencyjnie nawiązuje połączenie z serwerami innych użytkowników do określonego zagłębienia i zbiera od nich dane (w przystępnym formacie) na temat posiadania tego pliku (patrz 4.2).
- *pobieranie danych* - gdy użytkownik zdecyduje się pobrać pliki i zatwierdzi chęć nawiązania połączenia, to klient nawiązuje bezpośrednie połączenie z odpowiednim serwerem i odbiera zaszyfrowane dane, które sam odszyfrowuje (patrz 4.3).

3.3 Serwer

Każda aplikacja posiada moduł wielowątkowego serwera, który w tle odbiera i przekazuje zapytania z sieci. Z powodu różnorodności zapytań stworzyliśmy oddzielne instancje, które odbierają wiadomości na różnych portach:

- *dołączenie do sieci* - gdy nowy użytkownik próbuje się połączyć z daną maszyną, ten serwer jest odpowiedzialny za znalezienie odpowiedniej lokacji dla nowego użytkownika. Zwracane nowemu

użytkownikowi są dane wystarczające do identyfikacji ze znalezionym wierzchołkiem w sieci oraz adresy odpowiednich sąsiadów (patrz 4.1).

- *pytanie o pliki* - obsługuje zgodne z protokołem (4.2) zapytania o posiadanie danego pliku. Sprawdza, czy dany plik znajduje się w lokalnym folderze oraz przesyła zapytanie do komputerów dalej w sieci, zbierając od nich odpowiedzi. Suma tych wyników poszukiwań jest zwracana do nadawcy pytania.
- *przesyłanie danych* - odbiera prośbę o pobranie pliku od klienta innej instancji aplikacji, szyfruje i przesyła plik do kienta.

3.4 Klasa *NetworkManager*

Każda instancja aplikacji posiada swój obiekt opakowujący i zarządzający logiczną strukturą sieci i swoimi serwerami. Przechowuje on wycinek sieci jako 4 obiekty klasy *AddressBlock* (swój mesh, meshe dzieci i rodzica), swój adres IP oraz trzy numery portów (obiekt *ServerAddress*), na których pracują serwery. NetworkManager służy do pozyskiwania informacji o aktualnym stanie sieci.

3.5 Dołączanie do sieci i utrzymywanie jej formy

Za każdym razem, gdy podłącza się nowy użytkownik do sieci, musimy znaleźć mu pozycję (aka numer mesha), którą powinien zająć. Rozwiązujemy to w następujących krokach (dokładny opis protokołu znajduje się w 4.1):

1. komputer wysyła prośbę o podłączenie do sieci do jednego z komputerów w tej sieci,
2. odbiorca prośby przekazuje zapytanie w górę sieci tak długo jak potrafi,
3. najwyższy wierzchołek wysyła zapytanie w dół sieci, które znajduje najpłytszy mesha, który nie jest pełny (lub tworzy pustą instancję),

4. zapytanie jest przeniesione do dowolnego elementu tego niepełnego mesha lub, w drugim przypadku, do rodzica pustego mesha,
5. ten wierzchołek jest odpowiedzialny za powielenie swojej listy adresów i przekazanie jej nowemu (+ dopisanie go do swojej listy oraz powiadomienie innych wierzchołków w meshu o nowym) lub, w przypadku tworzenia nowego bloku, ustawieniu swojego mesha jako rodzica nowoutworzonego wierzchołka,
6. wysyłany jest komunikat do sąsiadów w sieci o dodaniu nowego wierzchołka, aby mogli go sobie dodać do listy znajomych,
7. komputer pytający już pełnoprawnie należy do sieci i może wykonywać swoje zadania jak każdy inny.

Metoda ta daje niskie prawdopodobieństwo na utworzenie niezbalansowanego drzewa, które powodowałoby opóźnienia.

3.6 Usuwanie nieaktywnych komputerów

Zakładamy, że duża ilość komputerów po pewnym czasie może stać się nieaktywna. Powodowałoby to zwiększenie szans na rozszczepienie sieci oraz zwiększenie średnich odległości między komputerami, które faktycznie chciałyby się porozumieć w jakimś celu. Dlatego potrzebujemy pewnej metody odsiewania komputerów, które prawdopodobnie już nigdy się nie odezwą na nasze wołanie.

Każdy z komputerów przy próbie połączenia losuje adresata z docelowego mesha. Aby losowanie nie było zbyt długie, to musi wpierw sprawdzić, które z komputerów po drugiej stronie są w ogóle dostępne. Za każdym razem, gdy komputer zgłosi dostępność, odświeżamy statystykę związaną z ostatnim razem, gdy została nawiązana łączność pomiędzy komputerami. Jeśli ta statystyka przekroczy pewną granicę (np. tydzień), to powinno być wysłane zapytanie do sieci, czy ktokolwiek sprzeciwia się usunięciu danego komputera. Aplikacje, które nawiązały łączność w niedawnym okresie z daną maszyną zawetują prośbę. Jeśli natomiast pytanie przeszłoby bez sprzeciwu, to mesh danego komputera oraz sąsiedzi w drzewie powinni od wnioskodawcy dostać wiadomość, że

pozbywamy się nieaktywnego użytkownika. Aplikacja otrzymująca taką wiadomość powinna usunąć dany komputer z wszystkich możliwych list adresów i zwolnić jego miejsce dla nowych.

Metoda ta nie jest skomplikowana, wymagałaby jedynie dodania statystyk przy sąsiedztwach w sieci oraz 2 dodatkowych typów zapytań - zawołania o *veto* oraz prośba o usunięcie z list sąsiedztwa. Ta zmiana powodowałaby większą stabilność i mniejsze oczekiwania przy dłuższym czasie działania sieci.
(proponowane rozwiązanie to koncept, nie zostało zaimplementowane)

3.7 Stałe

Stałe programu znajdują się w klasie *Common/Constants*:

1. *mesh_size* - ogranicznie na rozmiar meshy
2. *query_ttl* - zagłębienie rekurencji w zapytaniu o pliki
3. *ping_timeout* - opóźnienie, po jakim komputer może zostać uznany za nieaktywny

4. Opis protokołów

Wszystkie przedstawione niżej protokoły warstwy aplikacji korzystają z protokołu TCP. Dodatkowo protokół zapytania o plik można rozszerzyć na bezpieczne sockety określone w klasach *javax.net.ssl.SSLSocket* oraz *javax.net.ssl.SSLServerSocket*, tworzone za pomocą instancji klasy *javax.net.ssl.SSLContext*, bazujących na tzw. self-signed certificates. Rozwiązanie to nie zostało zaimplementowane w obecnej wersji, może jednak zapobiegać wykonywaniu złożonego protokołu zapytania o plik (patrz 4.2) wysłanego przez nieautoryzowanego hosta. Odpowiednie do tego celu metody znajdują się w klasie *network/SSLUtils*. Są to kolejno (domyślna wartość parametru oznacza przypadek, gdy ma on wartość *null*):

- public static SSLContext *getSSLContext*(String *keyStorePath*, char[] *keyStorePass*) throws Exception

  Zwraca obiekt klasy *SSLContext*, służący do utworzenia bezpiecznych socketów. Parametr keyStorePath wskazuje ścieżkę do pliku zawierającego wymagane certyfikaty, domyślnie jest to "keystore.jks". Argument keyStorePass zawiera hasło do tego pliku (domyślnie "changeit").

- public static SSLSocket *getClientSocket*(String *host*, int *port*, String *keyStorePath*, char[] *keyStorePass*) throws Exception

  Zwraca niepodłączony, bezpieczny socket służący do komunikacji jako klient. Argumenty *host* oraz *port* wskazują docelowy adres IP i port, pozostałe dwa są wykorzystywane w celu otrzymania obiektu *SSLContext* za pomocą metody *getSSLContext* (mogą być domyślne - *null*).

- public static SSLServerSocket *getServerSocket*(int *port*, String *keyStorePath*, char[] *keyStorePass*) throws Exception

  Zwraca bezpieczny socket służący do komunikacji jako serwer, przypisany do portu określonego przez argument *port*. Pozostałe argumenty są wykorzystywane w celu otrzymania obiektu *SSLContext* za pomocą metody *getSSLContext* (mogą być domyślne - *null*).

- public static void *addCert*(String *name*, char[] *keyStorePass*) throws Exception

  Metoda pomocnicza, służąca do dodawania certyfikatu z pliku *name*.pem do bazy zaufanych hostów JVM, aby możliwe było nawiązanie połączenia SSL. Ścieżka bazy wykrywana jest

automatycznie, najczęściej jest to %java.home%/jre/lib/security/cacerts. Alias certyfikatu musi mieć taką samą nazwę jak plik, tj. *name*. Argument *keyStorePass* zawiera hasło do pliku *cacerts*, domyślnie jest to "changeit". Na niektórych systemach może wystąpić konieczność uruchomienia tej metody z poziomu administratora, aby uzyskać dostęp do powyższego pliku.

4.1 Protokół podłączania do sieci. (autor: Michał Dyrek)

W obrębie tego protokołu znajduje się 5 różnych typów wiadomości:

Phase0 (punkty 1. i 2. w 3.4)**:** *0;ServAddress*
gdzie "*0*" to numer fazy, a *ServAddress* to *ServerAddress* komputera, czyli IP oraz numery 3 portów komputera który prosi o dodanie do sieci. Implementacja klasy *ServerAddress* znajduje się w *model/ServerAddress.java*

Jest to zapytanie, które wysyła komputer chcący się dołączyć do sieci do komputera już znajdującego się w sieci. Ten z kolei przesyła (rekurencyjnie) to zapytanie do jak najwyższego wierzchołka w sieci do jakiego jest w stanie. Gdy komputer nie jest już w stanie wyżej posłać tej wiadomości, rozpoczyna się kolejna faza.

Phase1 (punkt 3. w 3.4)**:** *1*
Wiadomość w tej fazie zawiera tylko "1".

Jest to faza, w której wierzchołek wyszukuje najpłytszy względem niego niepełny mesh. Wiadomość jest wysyłana w dół drzewa, a serwery dzieci odpowiadają (rekurencyjnie) numerem najpłytszego mesha pod nimi, który jest niepełny lub swoim, jeśli one są niepełne. Odpowiedzią jest pojedyncza liczba całkowita, oznaczająca numer mesha.

Phase2 (punkt 4. w 3.4)**:** *2;destId;ServAddress*
gdzie *destId* to numer mesha, w którym nowy wierzchołek powinien się znajdować.

W tej fazie przenosimy się rekurencyjnie do podmiotu odpowiedzialnego za wprowadzenie nowego wierzchołka. Sprawdzane jest, czy numer obecnego mesha jest równy docelowemu, a jeśli nie, to przesuwamy się do tego dziecka, w którego stronę się on znajduje, ponawiając zapytanie o tej samej treści. Istnieją dwie możliwości: nowy wierzchołek będzie dołączony do istniejącego mesha - wtedy końcem jest istniejący wierzchołek z docelowego mesha. Wpp. skończymy w wierzchołku bezpośrednio nad nieistniejącym meshem i nastepuje jego stworzenie. W każdym z tych przypadków, jeśli dotrzemy do celu, rozpoczyna się faza 3.

Phase3 (punkt 5. w 3.4)**:** *3;id;Neighbours*
gdzie id to numer mesha, w którym nowy wierzchołek się znajduje, a *Neighbours* to listy sąsiadów, które ma on sobie skopiować i wpisać. Technicznie jest to *AddressBlock[]*, którego implementację można znaleźć w *model/AddressBlock.java*

Jest to paczka wysłana przez wierzchołek wprowadzający na serwer nowego komputera. On ją sobie otwiera i kopiuje zawartość. Po przesłaniu danych wierzchołek wprowadzający rozpoczyna fazę 4.

Phase4 (punkt 6. w 3.4)**:** *4;destId;ServAddress*
gdzie *destId* i *ServAddress* to numer mesha nowego wierzchołka oraz jego dane.

Wysłana jest taka wiadomość do wszystkich sąsiadów nowego wierzchołka, aby mogli go sobie zorientować w ich listach sąsiadów.

Po tej procedurze nowy wierzchołek jest już w pełni zdolny do współuczestniczenia w sieci na równi z innymi.
Klasy poszczególnych zapytań znajdują się w *queries/PhaseXQuery.java*, gdzie X to numer fazy.

4.2 Protokół zapytania o plik. (autor: Patryk Urbański)

Zapytanie tekstowe w formacie:

id:filename:ttl

gdzie:

- *id* - identyfikator mesha, do którego należy wierzchołek wysyłający zapytanie
- *filename* - pełna lub częściowa nazwa szukanego pliku, zwracane są wszystkie pliki, zawierające w nazwie podany argument jako spójny podciąg
- *ttl* - liczba naturalna określająca jak daleko przekazywane jest zapytanie w głąb sieci

Odpowiedzią jest wieloliniowy string, w którym każda linia jest postaci:

address:port:path

gdzie:

- *address* - adres komputera w formacie IPv4, z którym należy się połączyć w celu pobrania pliku
- *port* - port serwera przesyłu danych znajdującego się na maszynie pod adresem *address*
- *path* - względna ścieżka do zlokalizowanego pliku (względem udostępnianego folderu), którą należy podać w protokole pobierania pliku (patrz 4.3)

Klient przyjmuje w konstruktorze zapytanie oraz managera (patrz 3.4), z którego wydobywa adresy sąsiadów, których należy spytać o dany plik. Następnie za pomocą metody *getReply()* przesyła zapytanie do losowo

wybranych reprezentantów każdego sąsiadującego mesha i oczekuje na odpowiedzi, które po otrzymaniu zwraca.

Serwer, otrzymując zapytanie, działa w dwojaki sposób. Jeśli ttl = 0, to serwer tylko zwraca listę dopasowanych plików, znajdujących się w jego lokalnym folderze i zamyka połączenie. Jeśli ttl > 0, to serwer ponadto rekurencyjnie wysyła zapytanie (w którym podmienia *id* na swoje *myid*) według następującej reguły:

- do wszystkich sąsiadów wewnątrz mesha wyślij zapytanie *myid:filename*:0 (zażądaj natychmiastowej odpowiedzi),
- do wszystkich sąsiednich meshy, z wyjątkiem tego, od którego pochodzi zapytanie wyślij: *myid:filename*:*ttl*-1.

Implementacja powyższego protokołu znajduje się w klasach: *network/FileSearchClient.java, network/FileSearchServer.java, network/FileSearchServerThread.java.*

4.3 Protokół pobierania pliku. (autor: Patryk Urbański)

W celu pobrania wybranego pliku należy wykorzystać odpowiedź uzyskaną z protokołu zapytania o plik (patrz 4.2): *address:port:path*. Należy połączyć się z hostem *address* na porcie *port*, podając w zapytaniu *path*. Służy do tego klasa *network/FileTransferClient.java*, której konstruktor przyjmuje wymagane dane. Następnie za pomocą metody *download()* ustanawiane jest połączenie i pobranie pliku, który zapisywany jest w udostępnianym folderze. Przesyłane pliki szyfrowane są algorytmem DES. W aplikacji pobieranie następuje w osobnym wątku. Implementacja serwera znajduje się w klasach *network/FileTransferServer.java* oraz *network/FileTransferServerThread.java* i sprowadza się do wyszukania żądanego pliku w folderze i wysłania go.

5. Dokumentacja użytkowa projektu

5.1 Wymaganie systemowe.

Do uruchomienia i poprawnego działania aplikacji wymagana jest Java w wersji co najmniej 1.7 (do pobrania stąd) oraz narzędzie Maven (do pobrania stąd).

5.2 Instalacja i uruchomienie aplikacji.

Kod źródłowy projektu można pobrać z serwisu github.com np. poleceniem *git clone https://github.com/butozerca/tcs-fileshare.git*. Następnie w katalogu głównym projektu tcs-fileshare wykonujemy kolejno:

mvn clean package
java -jar ./target/tcs-fileshare-0.0.1-SNAPSHOT-jar-with-dependencies.jar

5.3 Konfiguracja aplikacji.

Po pierwszym uruchomieniu należy odpowiednio skonfigurować aplikację. Najważniejszą rzeczą jest ustawienie swojego adresu IP oraz portów, na których nasłuchiwać mają serwery. Wykonujemy to korzystając z przycisku *Set IP*. W pojawiającym się okienku wpisujemy kolejno nasz adres IP, który widoczny będzie dla innych użytkowników, a następnie porty serwerów, kolejno serweru zapytań o plik, transferu plików i dodawania nowych wierzchołków. Domyślnie są to wartości: 0.0.0.0/20000/21000/22000. **Ważne:** modyfikacja tych wartości w późniejszym czasie może prowadzić do niezdefiniowanego zachowania aplikacji.

Uwaga: jeśli chcemy wykorzystywać aplikację poza siecią lokalną konieczne jest zagwarantowanie widoczności maszyny poza tą siecią np. przez posiadanie zewnętrznego IP lub odpowiednie forwardowanie portów na routerze.

Analogicznie za pomocą przycisków *Change Name* i *Change Path* możemy ustawić odpowiednio nazwę użytkownika oraz (relatywną) ścieżkę do udostępnianego folderu. W obecnej wersji nazwa użytkownika nie ma żadnego wpływu na działanie aplikacji. Domyślna ścieżka to *fileshare/shared/*. Obie wartości mogą być zmieniane w dowolnym momencie.

Początkowo aplikacja nie zna żadnych innych wierzchołków i tworzy strukturę sieci, sama będąc korzeniem. Jeśli chcemy od razu dołączyć się do innej sieci, należy użyć przycisku *Join a network*, wpisać adres IP znanego wierzchołka w innej sieci, jego port do obsługi dodawania nowych wierzchołków i kliknąć OK.

Ważne: po konfiguracji należy zapisać aktualne ustawienia za pomocą przycisku *Save*.

5.4 Przykład sesji - wyszukiwanie i pobieranie.

W tej sekcji pokażemy jak połączyć ze sobą dwie maszyny z sieci lokalnej oraz jak wyszukiwać i przesyłać pliki pomiędzy tymi maszynami. Załóżmy następujące adresy lokalne:

host A: 192.168.1.2

host B: 192.168.1.3

Konfigurujemy aplikację:

host A → Set IP → 192.168.1.2/25000/25001/25002 → OK

host B → Set IP → 192.168.1.3/25000/25001/25002 → OK

Host A chce dołączyć do sieci utworzonej przez host B:

host A → Join a network → 192.168.1.3/25002 → OK

Host B dodaje do swojego udostępnianego folderu pliki: *abc.txt*, *bcd.txt*, *cde.txt*. Teraz host A chce wyszukać wszystkie pliki na maszynie B zawierające w nazwie *bc*:

host A → Search → bc → OK

W wyniku zapytania host A otrzymuje listę znalezionych plików: *abc.txt*, *bcd.txt*. Chcąc pobrać oba pliki zaznacza oba checkboxy obok wyników i kilka OK. W osobnym wątku rozpoczyna się pobieranie plików do folderu hosta A.

6. Funkcjonalności do zaimplementowania w przyszłych wersjach

- usuwanie nieaktywnych przez dłuższy czas wierzchołków,
- podłączenie socketów SSL do protokołu zapytania o plik i instalacja certyfikatów podczas dodawania nowych wierzchołków,
- autoryzacja i obsługa wielu użytkowników jednej aplikacji na tej samej maszynie.