PSI 24Z - Dokumentacja końcowa

Maksym Bieńkowski (01178511@pw.edu.pl) | Jędrzej Grabski | Aleksander Drwal | Tomasz Kowalski

24.01.2024

Zmiany względem projektu wstępnego zostały umieszczone w oddzielnej sekcji pod jego treścią.

Projekt wstępny

1. Temat i treść zadania

Program obsługujący prosty protokół P2P (Peer-to-Peer)

Założenia

- Zasób to obiekt z danymi binarnymi identyfikowany pewną nazwą. Za takie same zasoby uważa się zasoby o takich samych nazwach.
- Rozmiar zasobu jest znaczny (tj. większy od jednorazowego transferu sieciowego).
- Początkowo dany zasób znajduje się w jednym hoście, a następnie może być propagowany do innych hostów w ramach inicjowanego przez użytkownika "ręcznie" transferu. Raz pobrany zasób zostaje zachowany jako kopia.
- Po pewnym czasie działania systemu ten sam zasób może znajdować się w kilku hostach sieci.
- Program ma informować o posiadanych lokalnie (tj. w danym węźle) zasobach i umożliwiać ich pobranie.

Funkcjonalność programu

- Dodawanie nowych zasobów przez użytkownika – wprowadzanie z lokalnego systemu plików.
- Pobieranie zasobów:
  - Użytkownik może pobrać konkretny zasób po nazwie ze zdalnego hosta (jeden zasóbna raz).
  - Użytkownik decyduje, z którego hosta dany zasób zostanie pobrany.
- Rozgłaszanie informacji o posiadanych lokalnie zasobach.

Dodatkowe założenia

- Zasób pobrany do lokalnego hosta jest kopią oryginału. Kopia jest traktowana tak samo jak oryginał (są nierozróżnialne), tj.:
  - Istnienie kopii jest rozgłaszane w taki sam sposób jak istnienie oryginału.
- Program powinien obsługiwać różne sytuacje wyjątkowe, np. przerwanie transmisji spowodowane błędem sieciowym.
- Lokalizacja zasobów odbywa się poprzez rozgłaszanie:
  - Wskazówka: użyć protokołu UDP, ustawić opcje gniazda `SO_BROADCAST`, wykorzystać adresy IP rozgłaszające (same bity "1" w części hosta).

Interfejs użytkownika

- Wystarczy prosty interfejs tekstowy.
- Interfejs powinien obsługiwać współbieżny transfer zasobów – tj. nie powinien się blokować w oczekiwaniu na przesłanie danego zasobu.

Wariant zadania W13/W22

- całość komunikacji (przesyłania zasobu) zrealizować na UDP, dodatkowo dla uproszczenia można przyjąć, że zasób mieści się w całości w jednym datagramie (datagram danych może być zgubiony – należy to uwzględnić)
- implementacja w C++

2. Interpretacja treści zadania

Główne funkcje programu:

1. **Udostępnianie zasobów lokalnych:**

- Na urządzeniu hosta istnieje folder roboczy programu, którego zawartość będzie udostępniana innym.
- Udostępnienie zasobu jest równoznaczne z umieszczeniem go w tym katalogu.

```
> share /path/to/resource
# równoznaczne ze skopiowaniem pliku /path/to/resource do /shared_folder
```

```
user@machine launch-program

Sharing resources from folder /shared_folder.
Shared resources:
file1.txt
file2.txt
```

2. **Rozgłaszanie udostępnianych zasobów:**

- Periodyczne wysyłanie listy dostępnych zasobów przy użyciu protokołu UDP w trybie broadcast.

3. **Aktualizacja dostępnych zasobów:**

- Po otrzymaniu wiadomości broadcastowej z listą udostępnianych przez hosta A zasobów, host B aktualizuje listę dostępnych u A zasobów.

```
> list-resources

file1.txt
hosts: [host1, host2]

# otrzymanie wiadomości o file2.txt od hosta 1
> list-resources

file1.txt
hosts: [host1, host2]

file2.txt
hosts: [host1]
```

4. **Pobieranie zasobu:**

- Transfer zasobu z wybranego hosta inicjowany przez użytkownika programu.
- Plik pobierany jest do katalogu `shared_folder` i wiadomość o dostępności wysyłana jest przy następnym rozgłoszeniu.

```
> download file1.txt host1

Downloading file1.txt from host1...

>

Completed download of file1.txt from host1.
Available at shared_folder/file1.txt
```

5. **Obsługa sytuacji wyjątkowych:**

- Ponawianie zapytań w przypadku utraty datagramów, stosowanie prostego rozwiązania ACK.
- Informowanie użytkownika o niepowodzeniach transferu.

```
> download file1.txt host1

Downloading file1.txt from host1...

Error while downloading file1.txt: could not connect to host1. Retrying...
Error while downloading file1.txt: could not connect to host1. Retrying...
Error while downloading file1.txt: could not connect to host1. Retrying...
Fatal: Error while downloading file1.txt: could not connect to host1. Retried 3 times.
# Usunięcie użytkownika host1 z pamięci do otrzymania od niego kolejnego broadcastu
```

6. **Sprawdzenie listy dostępnych zasobów**

- Po wpisaniu określonej komendy, wypisywane są adresy hostów wraz z udostępnianymi przez nich zasobami.

```
> list-resources

file1.txt
hosts: [host1, host2]
```

```
file2.txt
hosts: [host1, host3]
```

- Po wpisaniu innej komendy, wypisywane są adresy hostów udostępniających konkretny zasób.

```
> find file1.txt

file1.txt
hosts: [host1, host2]
```

3. Krótki opis funkcjonalny ("black-box")

Użytkownik

- Może udostępniać zasoby ze swojego systemu plików.
- Może wywołać pobieranie zasobu o podanej nazwie z określonego hosta.
- Może przeglądać udostępniane przez określonego hosta zasoby.
- Może przeglądać wszystkie dostępne w sieci zasoby.
- Jest informowany o wystąpieniu błędów i niepowodzeń.

Program

- Rozgłasza informacje o udostępnianych zasobach.
- Obsługuje prośby o przesłanie zasobów od innych hostów.
- Przesyła i odbiera zasoby.
- Zarządza transferami bez blokowania interfejsu użytkownika.
- Informuje użytkownika o wystąpieniu błędów i niepowodzeń.

5. Opis i analiza protokołów komunikacyjnych

Bazowy protokół - UDP

Struktura nagłówka pakietu

PROT_VERSION (8 bitów)

- wersja protokołu

MSG_TYPE (8 bitów)

- `RESOURCE_ANNOUNCE` - ogłoszenie lokalnie posiadanych zasobów
- `RESOURCE_REQUEST` - prośba o udostępnienie określonego zasobu od konkretnego hosta
- `RESOURCE_DATA` - przesłanie zasobu do węzła, który go zażądał

DATA_SIZE (32 bity)

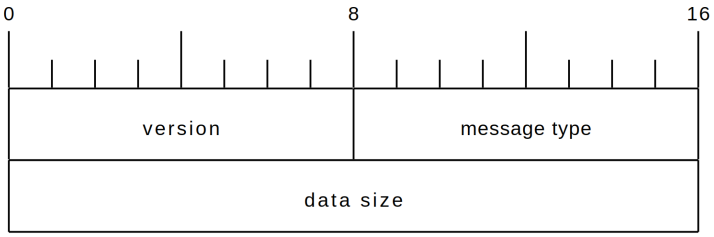
- długość danych spoza nagłówka w bajtach.

Za maksymalny rozmiar wysyłalnych danych uznajmy $65507 - 1 - 1 - 4 = 65501$ bajtów (zgodnie z założeniem, że dane mieszczą się w jednym datagramie UDP na podstawie przydzielonego wariantu zadania). Uznajemy, że zgubienie pakietu rozgłoszeniowego nie jest warte obsłużenia ze względu na ich periodyczne wysyłanie. Zgubienie pakietu `RESOURCE_REQUEST` obsługiwane jest mechanizmem ponownego odpytywania n razy do otrzymania zasobu, co rozwiązuje także przypadek zgubienia datagramu `RESOURCE_DATA`.

```
class PacketHeader {
public:
    enum class MsgType : uint8_t {
        RESOURCE_ANNOUNCE = 0,  // Ogłoszenie lokalnie posiadanych zasobów
        RESOURCE_REQUEST = 1,   // Prośba o udostępnienie określonego zasobu od konkretnego hosta
        RESOURCE_DATA = 2       // Przesłanie zasobu do węzła, który go zażądał
    };

    uint8_t PROT_VERSION; // Wersja protokołu (8 bitów)
    MsgType MSG_TYPE;     // Message type (8 bitów)
```

```
    uint32_t DATA_SIZE;    // Rozmiar danych do odebrania w bajtach (32 bitów)
};
```



Rysunek 1: Struktura nagłówka protokołu

Dane poza nagłówkiem

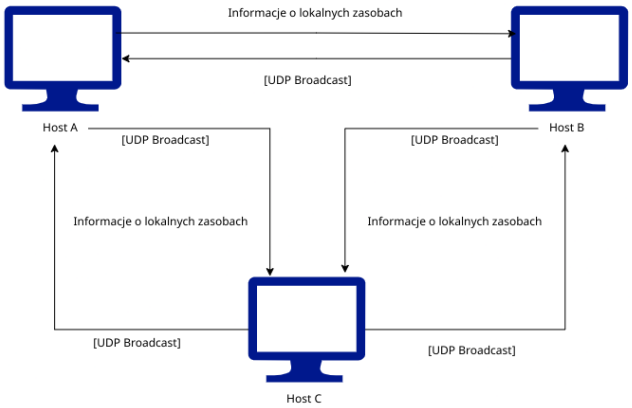
- W przypadku pakietu `RESOURCE_ANNOUNCE`, lista zakodowanych w ASCII nazw plików, separowanych przez bajt `\0`
- W przypadku pakietu `RESOURCE_REQUEST`, nazwa żądanego pakietu zakończona bajtem `\0`
- W przypadku pakietu `RESOURCE_DATA`, binarne dane zasobu o długości `DATA_SIZE`

```
class Message {
public:
    PacketHeader header;            // Nagłówek pakietu
    std::vector<uint8_t> data;      // Dane poza nagłówkiem (zmienny rozmiar)
};
```

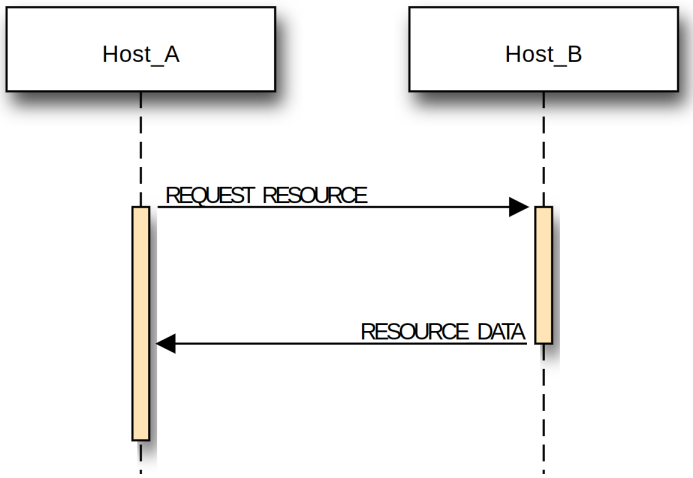
Przypisanie portu

- Zakładamy, że na potrzeby protokołu wszystkim hostom przypisany jest stały port.

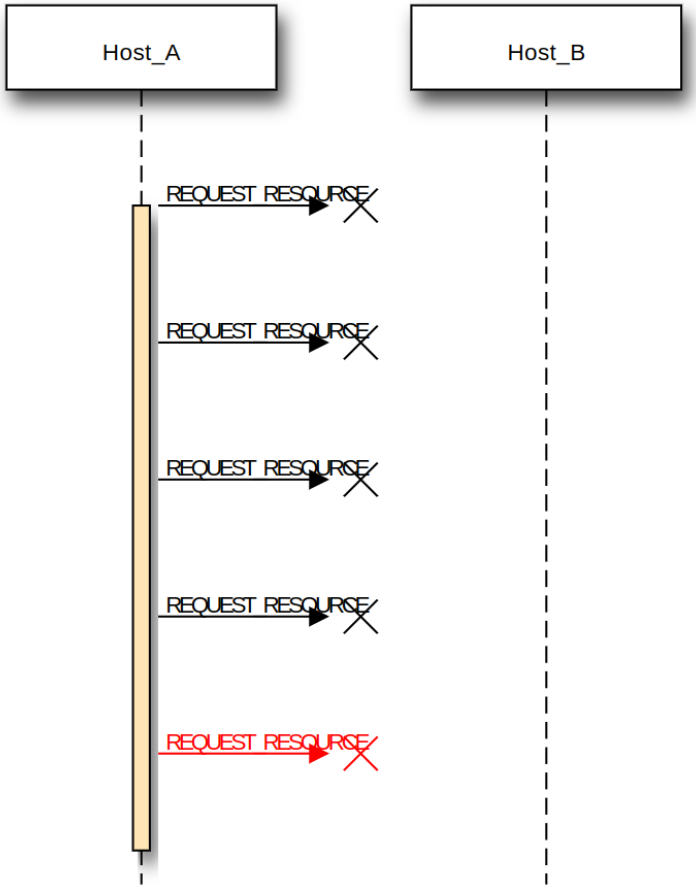
Diagramy komunikacji



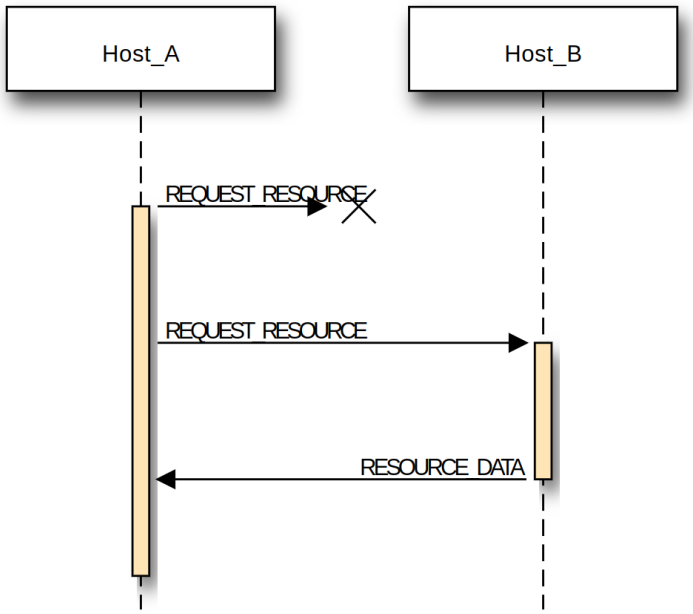
Rysunek 2: Rozgłaszanie zasobów



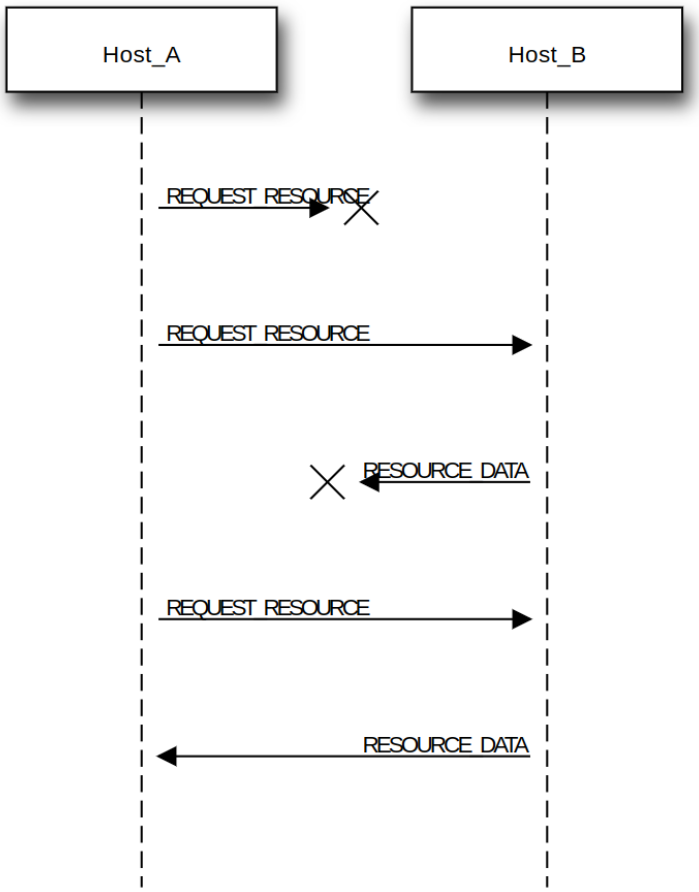
Rysunek 3: Bezproblemowy przesył danych



Rysunek 4: Niemożnośc osiągnięcia hosta



Rysunek 5: Utrata prośby o zasób



Rysunek 6: Utrata prośby i zasobu

6. Planowany podział na moduły

1. Moduł zarządzania zasobami:

- Obsługa dodawania, usuwania i przeglądu zasobów lokalnych

2. Moduł sieciowy:

- Rozgłaszanie zasobów
- Obsługa zapytań i transferu zasobów
- Zarządzanie retransmisją utraconych datagramów i obsługą błędów

3. Moduł interfejsu użytkownika:

- Prosty tekstowy interfejs umożliwiający wykonywanie operacji przez użytkownika, na wzór wcześniej pokazanych zrzutów z terminala

7. Zarys koncepcji implementacji

- Implementacja w języku C++

Biblioteki

- **Boost.Asio**: Obsługa gniazd sieciowych i operacji asynchronicznych.
- **STL**: Przechowywanie i zarządzanie danymi lokalnymi.

Narzędzia

- **CMake**: System budowy projektu.
- **GCC/Clang**: Kompilator.
- **Clang-tidy, clang-format**: Linter i formater.
- **Google test**: Testy jednostkowe

Ogólne podejście do implementacji

W celu zapewnienia nieblokujących operacji IO zamierzamy zastosować wielowątkową architekturę programu:

- wątek odpowiedzialny za interakcję przez CLI z użytkownikiem
- wątek odpowiedzialny za cykliczne wysyłanie rozgłoszeń
- dynamicznie powoływane wątki *workers* odpowiedzialne za pobieranie i przesyłanie zasobów

Zamierzamy zastosować obiektowe podejście przy implementacji programu w celu testowalnego i wyraźnego podziału odpowiedzialności między komponentami. Testowanie poprawności implementacji przeprowadzone będzie przy pomocy testów ręczych i jednostkowych.

Zmiany względem dokumentacji wstępnej

Struktura wiadomości

Z nagłówka zostało usunięte pole `DATA_SIZE`. Jako, że korzystamy z protokołu UDP i wszystkie wiadomości mieszczą się w jednym pakiecie, długość wiadomości obliczana jest na podstawie liczby odebranych bajtów i długości nagłówka: `MSG_SIZE = ALLTOGETHER_SIZE - HEADER_SIZE`. Pozwoliło nam to zmniejszyć rozmiar nagłówka do dwóch bajtów, dzięki czemu maksymalny rozmiar wysyłanego pliku to `65507 - HEADER_SIZE - MAX_FNAME_SIZE - 1`, gdzie 1 odpowiedzialne jest za separator między nazwą zasobu i jego treścią. Maksymalny rozmiar zasobu to ostatecznie `65507 - 2 - 1 = 255 = 65249` bajtów. Moglibyśmy pokusić się o pozwolenie na większy zasób w zależności nazwy (65504 dla jednoznakowej nazwy zasobu), jest to jednak podatne na błędy, więc z tego zrezygnowaliśmy.

Konsekwencje nieudanego pobrania zasobu

W dokumentacji początkowej uznaliśmy, że nieotrzymanie zasobu w odpowiedzi na prośbę skutkowało będzie usunięciem wszystkich zasobów użytkownika z bazy zasobów trzymanej w pamięci programu. Zamiast tego postanowiliśmy usunąć wyłącznie ten plik - nie zakładamy w ten sposób, że użytkownik stracił połączenie, a jedynie, że plik, który rozgłaszał, został usunięty/stracony.

Model wielowątkowy, struktura aplikacji

Założenia dotyczące radzenia sobie z utratą pakietów, modelu architektury wielowątkowej i podziału na moduły okazały się słuszne. W strukturze plików został wydzielony jeszcze pakiet `serialization` zawierający pomocnicze funkcje - nie chcieliśmy przeładować plików związanych z wiadomościami funkcjami, które nie są ściśle związane z serializacją ich struktury.

Wykorzystane biblioteki

Ostatecznie posłużyliśmy się obsługą wątków zawartą w bibliotece standardowej, w związku z czym biblioteka `Boost` okazała się zbędna.

Interfejs uzytkownika

Interfejs użytkownika uległ kilku kosmetycznym zmianom w porównaniu do dokumentacji wstępnej, funkcjonalność pozostała taka sama.

Inne

Zamiast `uint8_t` do przekazywania i serializacji wiadomości został użyty nowszy `std::byte`.

Warte wspomnienia jest, że nazwy plików nie mogą zawierać znaków spoza zbioru ASCII ze względu na sposób deserializacji danych.

Opis najważniejszych rozwiązań funkcjonalnych wraz z uzasadnieniem

W architekturze programu można wyróżnić 5 główne komponenty:

CLI - interfejs linii poleceń

Pozwala na interakcję użytkownika ze wszystkimi funkcjami programu:

- pobieranie pliku
- wyświetlanie dostępnych plików i hostów je posiadających
- znalezienie hosta posiadającego dany plik

CLI jest uruchamiane w głównym wątku programu i zakończenie go kończy działanie programu.

Rozgłaszacz broadcast

Komponent uruchomiony w osobnym wątku, który cyklicznie rozgłasza informacje o dostępnych plikach.

Serwer

Komponent uruchomiony w osobnym wątku, który nasłuchuje na przychodzące pakiety i wykonuje wymagane nie blokując zadania np. zapisuje plik na dysku oraz zleca zadania innym komponentom.

Moduł pobierania plików

Komponent przechowujący strukturę danych zawierającą wątki pobierające pliki.

Do zlecenia pobrania pliku przypisywany jest nowy wątek (identyfikowany po nazwie pliku), który do 5 razy próbuje pobrać plik od innego klienta. Wątek jest i usuwany z hash mapy i ewentualnie zabijany po udanej lub nieuadanej próbie pobrania pliku. Tak zaprojektowane wielowątkowe rozwiązanie pozwala na pobieranie wielu plików jednocześnie bez blokowania innych operacji.

Sekcja krytyczna zabezpieczona jest mutexem, który zapewnia, że dostęp do hash mapy i tworzenie nowych zleceń pobrania pliku jest bezpieczne w środowisku wielowątkowym. Do komunikacji z wątkami odpowiedzialnymi za pobieranie zasobów (w przypadku otrzymania zasobu lub polecenia wyjścia z CLI) użyte są warunki stopu w postaci `std::atomic_int`.

Moduł zarządzania plikami

Komponent przechowujący listę plików dostępnych do pobrania od każdego połączonego klienta. Jest to hash mapa z kluczami będącymi IP innych użytkowników i listami plików dostępnych od nich do pobrania jako wartościami.

Moduł ten pozwala na aktualizaowanie listy plików dostępnych do pobrania oraz rozgłoszania.

Sekcja krytyczna zabezpieczona jest mutexem, który zapewnia stabilne działanie komponentu nawet w gdy jednocześnie wiele użytkowników powoduje zmiany w liście plików dostępnych do pobrania.

Opis interfejsu użytkownika

Interfejs CLI pozwala użytkownikom na zarządzanie wyświetlanie zasobów dostępnych w sieci peer-to-peer, wyszukiwanie plików w sieci, pobieranie plików od innych hostów. Poniżej opisano szczegółowo funkcje i dostępne komendy:

---

1. Uruchomienie aplikacji

Po uruchomieniu aplikacji użytkownik widzi powitanie oraz listę dostępnych komend:

```
Welcome to the P2P File Sharing CLI!
Available commands:
  list-resources
  find <filename>
  download <host-ip> <filename>
  help
  exit
```

2. Dostępne komendy

1. **`list-resources`**
   Wyświetla listę zasobów dostępnych w sieci P2P, które można pobrać od innych użytkowników.

   Przykład użycia:

   ```
   > list-resources
   Listing resources available for download:

   file1.txt
   [192.168.1.2, 192.168.1.3]

   file2.jpg
   [192.181.2.3, 192.172.5.2, 192.151.2.5]
   ```

2. **`find <filename>`**
   Wyszukuje określony plik w sieci i podaje listę hostów, którzy go udostępniają.

   Przykład użycia:

   ```
   > find example.txt
   The file 'example.txt' is available from the following hosts:
   1) 192.168.1.5
   2) 192.168.1.7
   ```

3. **`download <host-ip> <filename>`**
   Pobiera plik od określonego hosta w sieci P2P.

   Przykład użycia:

   ```
   > download 192.168.1.5 example.txt
   Downloading 'example.txt' from 192.168.1.5...
   ```

4. **`help`**
   Wyświetla listę wszystkich dostępnych komend.

   Przykład użycia:

   ```
   > help
   Available commands:
     list-resources
     find <filename>
     download <host-ip> <filename>
     help
   ```

```
exit
```

5. `exit`
   Zamyka aplikację.

   Przykład użycia:

   ```
   > exit
   Exiting CLI.
   ```

3. Obsługa błędów

Interfejs CLI zapewnia obsługę błędów i podpowiedzi, gdy użytkownik wprowadzi nieprawidłowe polecenie lub pomyli składnię.

Przykłady obsługi błędów:

- W przypadku braku wymaganych argumentów:

  ```
  > find
  Usage: find <filename>
  ```

- W przypadku nieznanej komendy:

  ```
  > unknown-command
  Unknown command: unknown-command
  ```

4. Wyjście z aplikacji

Aby zamknąć aplikację, użytkownik może wpisać komendę `exit`. Aplikacja wyświetla komunikat i kończy działanie.

Postać wszystkich plików konfiguracyjnych, logów, itp

W projekcie tworzenie obrazów jest skonfigurowane w Dockerfile

- instalowanie pakietów:
  - narzędzia, pakiety itd.
- przygotowanie środowiska:
  - katalog roboczy
- ustawianie *just* jako komendy startowej

Do kompilacji i budowania skonfigurowany jest plik CMakeLists.txt

- wymagania minimalne
- standard C++
- katalogi z kodem źródłowym
- pliki źródłowe
- biblioteki
- testy

Żeby uruchamiać kontenery naraz skonfigurowany jest plik docker-compose.yaml

- daje możliwość uruchamiania kontenerów jednocześnie
- skonfigurowane są w nim trzy targety: client1/2/3 używane do testów programu
  - budowane są na podstawie tego samego obrazu
- podłączane są do tej samej sieci *Docker network*, żeby mogły się komunikować

Logi realizujemy przy użyciu biblioteki spdlog

- używany jest do wypisywania komunikatów o działaniach programu, wysłanych i odebranych wiadomościach itp.
- wypisuje informacje, ostrzeżenia, błędy
- użyty jest Logger (*Log.h*)
  - w zależności od użycia:
    - loguje na konsolę
    - zapisuje logi do pliku
- ma ustawiony poziom logowania na *trace*, w domyślnym ustawieniu logi trafiają do pliku

Testowanie

Podczas testowania aplikacji pojawiły się różne błędy

Użytkownik widzący swoje własne pliki

- BroadcastUDP prowadził do tego, że użytkownik miał wyświetlane pliki znajdujace się w jego własny lokalnym folderze zasobów
- Rowiązanie: Host ignoruje własne pakiety

Niepoprawna synchronizacja wątków

- Przez niepoprawną synchronizację i niezabijanie wątków dochodziło do błędów segmentacji (segfault)
- Rozwiązanie: Żeby tego uniknąć zastowana została opisana wcześniej flaga `std::atomic_bool` dla pobocznych wątków

Błędne formatowanie CLI

- Dochodziło do błędnego formatowania CLI:
  - pobieranie pliku o arbitralnej nazwie, który nie był udostępniany przez hosta skutkowało wiadomościami o błędzie w cli hosta, od którego próbowano pobrać plik
  - plik o tej nazwie tworzony był w katalogu pobierającego
- Powodowany przez asynchroniczne otrzymanie komunikatów o powodzeniu bądź niepowodzeniu pobierania
- Rozwiązanie: Zastosowanie sekwencji sterujących ASCII i std::flush, uniemożliwienie wysyłania nieposiadanego pliku

Udało się nam rozwiązać wszystkie dotychczas napotkane błędy.

Scenariusze testów manualnych - zamierzone użycie

Poniżej przedstawiono scenariusze testowe, oraz otrzymane rezultaty.

Test 1: Wyświetlenie listy dostępnych zasobów (`list-resources`) Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `list-resources`.

Oczekiwana odpowiedź (sukces):

```
Listing resources available for download:

file1.txt
[192.168.1.2, 192.168.1.3]

file2.jpg
[192.181.2.3, 192.172.5.2, 192.151.2.5]
```

Test 2: Wyszukanie pliku w sieci (`find <filename>`) Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `find example.txt`.

Oczekiwana odpowiedź (sukces):

```
The file 'example.txt' is available from the following hosts:
1) 192.168.1.5
2) 192.168.1.7
```

Test 3: Pobranie pliku z hosta (`download <host-ip> <filename>`) Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `download 192.168.1.5 example.txt`.

Oczekiwana odpowiedź (sukces):

```
Downloading 'example.txt' from 192.168.1.5
```

Test 5: Wyświetlenie listy komend (`help`) Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `help`.

Oczekiwana odpowiedź (sukces):

```
Available commands:
  list-resources
  find <filename>
  download <host-ip> <filename>
  change-resource-folder <new-folder-path>
  help
  exit
```

Test 6: Zamknięcie aplikacji (`exit`) Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `exit`.

Oczekiwana odpowiedź (sukces):

```
Exiting CLI.
```

Scenariusze testów manualnych - niepowodzenia

Poniżej przedstawiono scenariusze testowe dla przypadków, w których działanie aplikacji kończy się niepowodzeniem.

Test 1: Wpisanie nieznanej komendy Kroki testowe:

1. Uruchom aplikację.
2. Wpisz nieznaną komendę, np. `unknown-command`.

Oczekiwana odpowiedź:

```
Unknown command: unknown-command
```

Test 2: Wyświetlenie zasobów gdy nie ma dostępnych Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `list-resources`, gdy nie ma dostępnych danych

Odpowiedź aplikacji:

```
Listing resources available for download:
```

Test 3: Wyszukiwanie pliku, który nie istnieje w sieci Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `find non-existing-file.txt`.

Oczekiwana odpowiedź:

```
No hosts have the file 'non-existing-file.txt'
```

Test 4: Pobieranie pliku od hosta, który nie istnieje Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `download 192.168.1.100 example.txt`, gdzie `192.168.1.100` to nieistniejący host.

Odpowiedź aplikacji:

```
> download 192.168.1.100 example.txt
Downloading 'example.txt' from 192.168.1.100
> Download failed: example.txt
```

Test 5: Pobieranie pliku, który nie jest udostępniany przez hosta Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `download 192.168.1.5 non-existing-file.txt`, gdzie plik nie istnieje na hoście.

Odpowiedź aplikacji:

```
> download 172.22.0.3 non-existing-file.txt
Downloading 'non-existing-file.txt' from 172.22.0.3
> Download failed: non-existing-file.txt
```

Test 7: Wyszukiwanie pliku bez podania nazwy Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `find`.

Oczekiwana odpowiedź:

```
Usage: find <filename>
```

Test 8: Pobieranie pliku bez podania adresu hosta Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `download example.txt` (brak adresu hosta).

Oczekiwana odpowiedź:

```
Usage: download <host-ip> <filename>
```

Test 9: Pobieranie pliku bez podania nazwy pliku Kroki testowe:

1. Uruchom aplikację.
2. Wpisz komendę `download 192.168.1.5`.

Oczekiwana odpowiedź:

```
Usage: download <host-ip> <filename>
```

Narzędzia

Zgodnie z dokumentacją wstępną, użyliśmy CMake do zbudowania projektu. Użyliśmy także `clang_tidy` i `clang_format` ze współdzieloną konfiguracją trzymaną w repozytorium. Testowanie projektu przeprowadzaliśmy początkowo na maszynie lokalnej, następnie w sieci dockerowej. Do zagregowania często używanych komend - budowania, formatowania, lintowania, uruchamiania testów oraz wynikowych plików binarnych użyliśmy narzędzia just, które jest nowoczesnym zamiennikiem `make`. Justfile zawiera także instrukcje do uruchomienia aplikacji z predefiniowanymi parametrami dla określonych użytkowników sieci projektowej.