Techniki internetowe

Dokumentacja projektu "TinDox"

Jakub Mazurkiewicz	Damian Piotrowski	Anna Pyrka	Łukasz Reszka

Semestr 21Z

Spis treści

1. Cel projektu

Celem projektu jest opracowanie protokołu do wymiany plików przez sieć IPv4. Zakłada się także stworzenie wydajnego serwera oraz klientów na wybrane platformy.

2. Wymagania protokołu TDP

2.1. Opis

TDP to protokół komunikacyjny typu serwer–klient wykorzystujący protokół sterowania transmisją (TCP). Umożliwia dwukierunkowy transfer plików oraz przeglądanie katalogów znajdujących się na zdalnym dysku.

Komunikaty w protokole wymieniane są za pośrednictwem połączenia głównego (na rysunku oznaczonym jako "Polecenia i odpowiedzi"). Ich struktura jest inspirowana protokołem HTTP.

2.2. Słownik pojęć

1. TDP – TinDox Protocol.
2. Aktualny katalog – każdy zalogowany użytkownik posiada przypisany do siebie katalog bieżący, czyli ten, w którym się aktualnie znajduje.
3. Kod operacji – losowo wygenerowany 32-bitowy kod identyfikujący pewną złożoną operację (np. pobieranie pliku).

2.3. Autoryzacja

Po uzyskaniu połączenia z serwerem, klient powinien dokonać autoryzacji. Za pomocą odpowiedniego komunikatu przesyła on swój login oraz hasło, które są następnie weryfikowane przez serwer. W przypadku podania nieprawidłowych danych, użytkownik będzie miał możliwość poprawy, ale tylko maksymalnie trzy razy.

Na jedno konto może być zalogowany tylko jeden użytkownik. Jeżeli klient spróbuje wykorzystać aktualnie używane konto to otrzyma on błąd.

2.4. Poruszanie się po katalogach

Klient ma możliwość swobodnego przechodzenia po dostępnych katalogach. Może on korzystać ze ścieżek względnych lub bezwzględnych. Analogicznym działaniem w systemie Linux jest instrukcja `cd`.

2.5. Wypisywanie informacji o plikach i katalogach

Klient ma możliwość:

1. Wypisania plików znajdujących się w aktualnym katalogu (lub w innym jeżeli poda ścieżkę),
2. Wypisania drzewa katalogów i plików zaczynając od aktualnego katalogu (lub innego podanego jako parametr),
3. Wyświetlenia pełnej ścieżki do aktualnego katalogu.

2.6. Tworzenie katalogów

Możliwe jest tworzenie katalogów tylko wewnątrz bieżącego katalogu. Jeśli katalog o podanej nazwie już istnieje użytkownik otrzyma błąd.

W przypadku gdy dwie osoby spróbują stworzyć w tym samym czasie katalog o tej samej nazwie, jedna z nich otrzyma błąd mówiący że katalog już istnieje.

2.7. Tworzenie plików

- Scenariusz główny:
  1. Klient wysyła do serwera żądanie utworzenia pliku o wskazanej nazwie w folderze, w którym obecnie się znajduje,
  2. Serwer sprawdza, czy w danym katalogu istnieje już plik o danej nazwie,
  3. Serwer tworzy plik o podanej nazwie w odpowiednim katalogu,
  4. Serwer wysyła do klienta potwierdzenie utworzenia nowego pliku.
- Scenariusz alternatywny – próba utworzenia w "folderze klienta" pliku o tej samej nazwie co inny plik:
  1. Kroki 1 – 2 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę wykonania operacji.

2.8. Usuwanie plików i katalogów

- Scenariusz główny:
  1. Klient wysyła do serwera żądanie usunięcia wskazanego pliku lub katalogu,
  2. Serwer sprawdza, czy wskazany katalog (lub plik) istnieje,
  3. Serwer sprawdza, czy w katalogu do usunięcia ktoś się znajduje, czy plik do usunięcia jest aktualnie pobierany lub czy ktoś nie ładuje pliku do wskazanego katalogu,
  4. Jeśli nie, to serwer usuwa katalog (lub plik) i wysyła potwierdzenie usunięcia do klienta.
- Scenariusz alternatywny I – w katalogu do usunięcia ktoś się znajduje lub plik do usunięcia jest pobierany:
  1. Kroki 1 – 3 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę wykonania operacji.
- Scenariusz alternatywny II – próba usunięcia nieistniejącego pliku lub katalogu:
  1. Kroki 1 – 2 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę wykonania operacji.

2.9. Zmienianie nazwy plików i katalogów

- Scenariusz główny:
  1. Klient wysyła do serwera żądanie zmiany nazwy pliku lub katalogu,
  2. Serwer sprawdza, czy istnieje plik lub katalog o tej samej nazwie w danym katalogu,
  3. Jeśli jest to plik, serwer sprawdza, czy jest on aktualnie pobierany,
  4. Jeśli jest to katalog, serwer sprawdza, czy aktualnie ktoś się w nim znajduje,
  5. Serwer zmienia nazwę danego pliku lub katalogu i wysyła potwierdzenie do klienta.
- Scenariusz alternatywny I – istnieje już plik lub katalog o tej samej nazwie:
  1. Kroki 1 – 2 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę zmiany nazwy.
- Scenariusz alternatywny II – próba zmiany nazwy pliku który jest obecnie pobierany:
  1. Kroki 1 – 3 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę zmiany nazwy.
- Scenariusz alternatywny III – próba zmiany nazwy katalogu w którym ktoś się znajduje:
  1. Kroki 1 – 4 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę zmiany nazwy.

2.10. Kopiowanie lub przenoszenie plików między katalogami

- Scenariusz główny:
  1. Klient wysyła do serwera żądanie skopiowania pliku o danej nazwie do zadanego katalogu,
  2. Serwer sprawdza, czy plik który ma być skopiowany (lub przeniesiony) istnieje,
  3. Serwer sprawdza, czy istnieje katalog, do którego chcemy skopiować plik,
  4. Serwer sprawdza, czy w danym katalogu istnieje już plik o danej nazwie,
  5. Jeśli nie, serwer kopiuje plik do podanego katalogu.
- Scenariusz alternatywny I – plik który chcemy skopiować (lub przenieść) nie istnieje:
  1. Kroki 1 – 2 scenariusza głównego,
  2. Serwer wysyła do klienta odmowę skopiowania pliku.

- Scenariusz alternatywny II – próba skopiowania pliku w miejsce gdzie istnieje już inny o tej samej nazwie:
    1. Kroki 1 – 4 scenariusza głównego,
    2. Serwer skopiuje plik, ale zmieni jego nazwę dopisując np. frazę "`(copy N)`", gdzie `N` to numer kopii.
- Scenariusz alternatywny III – próba przeniesienia pliku w miejsce gdzie istnieje już inny o tej samej nazwie:
    1. Kroki 1 – 3 scenariusza głównego,
    2. Serwer wysyła do klienta odmowę przeniesienia pliku.
- Scenariusz alternatywny IV – próba przeniesienia pliku, który jest aktualnie pobierany
    1. Krok 1 scenariusza głównego,
    2. Serwer wysyła do klienta odmowę przeniesienia pliku.

2.11. Pobieranie plików

Klient może pobrać dany plik z serwera z aktualnego katalogu pod warunkiem, że ma do tego odpowiednie uprawnienia:

- Scenariusz główny:
    1. Klient wysyła pytanie o zgodę na pobranie pliku. Zawiera się w niej nazwa przesyłanego pliku oraz opcjonalnie number bajtu, od którego klient chce rozpocząć pobieranie,
    2. Serwer, po zweryfikowaniu uprawnień użytkownika, wysyła zgodę na pobieranie wraz z dodatkową informacją – ilością bajtów, które klient będzie musiał przeczytać,
    3. Klient wysyła komunikat inicjujący pobieranie,
    4. Serwer przesyła dane strumieniowo do klienta, który musi odczytać dokładnie tyle bajtów ile zostało określone w odpowiedzi,
    5. Klient czyta bajty ze strumienia i dopisuje je do pliku o nazwie "`[NAZWA-ŁADOWANEGO-PLIKU].partial`" reprezentującego częściowo pobrane dane,
    6. Po przeczytaniu wszystkich bajtów aplikacja klienta zmienia nazwę pliku częściowego na docelową.
- Scenariusz alternatywny I - odmowa pobierania:
    1. Krok 1 scenariusza głównego,
    2. Klient otrzymuje odmowę pobierania pliku (np. z powodu braku uprawnień) – użytkownik dostaje informację o niepowodzeniu operacji.
- **Scenariusz alternatywny II – przerwanie połączenia z siecią w trakcie pobierania pliku**:

1. Kroki 1 – 4 scenariusza głównego,
2. Przerwanie połączenia ze strony klienta lub serwera,
3. Klient zapisuje do specjalnego pliku informację o przerwanej operacji. Zawiera się w niej ścieżka do pobieranego pliku oraz numer pierwszego bajtu, który nie został odebrany,
4. Klient i serwer oczekują na odzyskanie łączności z siecią,
5. Klient nawiązuje ponownie połączenie z serwerem, loguje się na swoje konto,
6. Klient sprawdza czy żadna operacja nie została wcześniej przerwana. Tak jest, zatem kieruje do użytkownika zapytanie o ponowienie operacji,
7. Jeżeli użytkownik chce ponowić operację, to aplikacja automatycznie zmienia bieżący katalog na ten, w którym znajdował się pobierany plik,
8. Wykonywany jest ponownie krok 1 scenariusza głównego. Klient tym razem wysyła w żądaniu numer bajtu, od którego chce rozpocząć ponowne pobieranie,
9. Kroki 2 – 6 scenariusza głównego.

2.12. Ładowanie plików

Klient może załadować plik do aktualnego katalogu:

- Scenariusz główny:
  1. Klient wysyła pytanie o zgodę na załadowanie pliku (`ul`). Zawiera się w nim nazwa przesyłanego pliku oraz jego rozmiar,
  2. Serwer weryfikuje uprawnienia użytkownika i ilość dostępnego miejsca na dysku,
  3. Serwer wydaje zgodę na ładowanie. W odpowiedzi zawiera również ilość pierwszych bajtów pliku, które powinny zostać pominięte (istotne w przypadku ponownej próby ładowania pliku),
  4. Klient, w przypadku uzyskania pozytywnej odpowiedzi, od razu przesyła strumieniowo dane do serwera, który musi odczytać dokładnie tyle bajtów ile było określone w żądaniu (minus ewentualne pomijane pierwsze bajty),
  5. Serwer odbiera fragmenty pliku od klienta i dopisuje je do pliku o nazwie "`[NAZWA-UŻYTKOWNIKA].partial`" reprezentującego częściowo załadowane dane. Plik ten umieszczony jest w katalogu z konfiguracją serwera (`.tds`), co pozwala ukryć go przed innymi użytkownikami,
  6. Serwer po wczytaniu całego pliku zmienia nazwę pliku na docelową oraz przenosi go do docelowego katalogu.
- Scenariusz alternatywny I - odmowa ładowania:
  1. Krok 1 scenariusza głównego,

2. Klient otrzymuje odmowę ładowania pliku – użytkownik dostaje odpowiedź o niepowodzeniu operacji. Serwer pozostaje w trybie przyjmowania poleceń.

- **Scenariusz alternatywny II – przerwanie połączenia z siecią w trakcie ładowania pliku**:

  1. Kroki 1 – 5 scenariusza głównego,
  2. Przerwanie połączenia ze strony klienta lub serwera,
  3. Serwer zapisuje parametry przerwanej operacji do pliku o nazwie "`[NAZWA-UŻYTKOWNIKA].upload`" znajdującego się w katalogu z konfiguracją serwera ("`.tds`"). Parametry przerwanej operacji to docelowy rozmiar pliku, ilość przeczytanych do tej pory bajtów, nazwa pliku oraz jego miejsce docelowe na serwerze,
  4. W tym samym czasie klient zapisuje do pliku informację o przerwaniu operacji oraz o pliku, który był ładowany,
  5. Klient i serwer oczekują na odzyskanie łączności z siecią,
  6. Klient nawiązuje ponownie połączenie z serwerem, loguje się na swoje konto,
  7. Klient sprawdza czy żadna operacja nie została wcześniej przerwana. Tak jest, zatem kieruje do użytkownika zapytanie o ponowienie operacji,
  8. Użytkownik ponawia ładowanie pliku – aplikacja klienta wysyła komunikat rozpoczynający ładowanie pliku z dodatkowym parametrem, mówiącym o ponowieniu operacji,
  9. Serwer, po zweryfikowaniu że taka operacja faktycznie miała wcześniej miejsce, wyraża zgodę na ponowienie,
  10. Kroki 4 – 6 scenariusza głównego.

3. Struktura protokołu TDP

3.1. Struktura komunikatów

3.1.1 Komunikaty wysyłane przez klienta

Każdy komunikat składa się z jednej linii, w której znajduje się nazwa polecenia oraz wielu linii w postaci "`nazwa_pola:wartość`". Wymagania:

1. Kolejne komunikaty oddzielone są pustą linią (znak o kodzie `0x0A`),
2. Każda linia może mieć maksymalnie długość 2048 bajtów. Po przeczytaniu tej ilości danych serwer zwróci błąd i rozpocznie analizę kolejnego polecenia,
3. Każde polecenie może mieć maksymalnie 16 pól,
4. Nazwa pola może składać się tylko z małych liter alfabetu łacińskiego (znaki o kodach od `0x61` do `0x7A`),
5. Między:

- Początkiem linii a nazwą pola,
- Nazwą pola a dwukropkiem,
- Dwukropkiem a wartością pola,
- Wartością pola a końcem linii,

Mogą znaleźć się białe znaki o kodach `0x09` (tabulacja) oraz `0x20` (spacja).

6. Wartość pola może być typu:
   - `boolean` – pole jest typu logicznego, gdy jego wartość jest równa dokładnie `true` lub `false`,
   - `integer` – pole jest typu całkowitego, gdy jego wartość składa się wyłączne z cyfr,
   - `string` – ostatecznie pole jest typu `string`. Jeżeli pierwszym i ostatnim znakiem pola jest apostrof (kod `0x27`) to znaki te nie są brane pod uwagę przez interpreter (przykładowo wartość "`'value'`" jest tym samym co "`value`").

3.1.2 Odpowiedzi serwera

Struktura odpowiedzi:

1. Zawsze w pierwszej linii znajduje się kod zwrotny wykonanej operacji oraz nazwa wykonanego polecenie oddzielone spacją,
2. W przypadku odpowiedzi pozytywnej w kolejnych liniach zawarte są jej szczegóły. Struktura odpowiedzi zależy wykonanej komendy,
3. W przypadku odpowiedzi negatywnej w jednej linii zawarta jest krótka informacja o błędzie. Jest to informacja dla programisty, aplikacja klienta nie powinna pokazywać jej użytkownikowi,
4. Odpowiedź jest zakończona pustą linią (znak o kodzie `0x0A`).

3.1.3 Kody zwrotne serwera

Kod 100 (`ok`) oznacza, że żądana operacja powiodła się. Kody `3xx` informują o błędach interpretera lub wykonawcy komend, które zostały spowodowane np. nieprawidłowo sformułowanym przez klienta komunikatem. Kody `4xx` informują o błędach przy wykonywaniu poleceń, np. użytkownik próbował wykonać zablokowaną operację.
Kody błędów interpretera i wykonawcy komend:

- `300 too_long_line` – błąd ten oznacza, że przeczytana linia jest zbyt długa dla interpretera,
- `301 too_many_fields` – błąd oznaczający, że klient podał zbyt wiele pól,
- `302 bad_field` – błąd oznaczający, że format pola jest nieprawidłowy (np. pole nie ma wartości),
- `303 bad_command` – błąd oznaczający, że wskazana komenda nie istnieje.

Błędy te występują tylko w przypadku odbioru nieprawidłowego pakietu. Po ich wystąpieniu interpreter poleceń dokonuje resetu swojego stanu – dzięki temu jest w stanie przyjąć od razu kolejny pakiet.
Kody błędów wykonywanych komend:

- `401 unknown` – nieznany błąd spowodowany np. poważnym błędem systemu operacyjnego serwera,
- `402 not_logged_in` – błąd zwracany, gdy klient próbuje skorzystać z komendy wymagającej autoryzacji bez bycia zalogowanym (np. użycie `ls` przed komunikatem `auth`),
- `403 invalid_field_value` – pole komunikatu ma nieprawidłową wartość lub jest nieprawidłowego typu,
- `404 not_found` – żądany obiekt nie został znaleziony (np. plik przy próbie pobierania, miejsce docelowe przy przenoszeniu),
- `405 no_upload_to_resume` – klient nie może wznowić ładowania pliku, gdyż:
  - Nie rozpoczął ładowania,
  - Rozpoczął ładowanie innego pliku przed wznowieniem ładowania poprzedniego,
- `406 not_enough_perms` – użytkownik nie posiada uprawnień do wykonania danej operacji,
- `407 user_already_logged` – błąd zwracany, gdy użytkownik próbuje użyć komunikatu `auth` będąc zalogowanym lub gdy inny użytkownik próbuje zalogować się na aktualnie używane konto,
- `408 invalid_credentials` – użytkownik próbuje użyć komunikatu `auth` z nieprawidłowymi danymi (np. zły login lub hasło),
- `409 file_already_exists` – użytkownik próbuje stworzyć obiekt, który już istnieje w systemie plików, np. próbuje utworzyć istniejący katalog lub próbuje załadować plik o powtarzającej się nazwie,
- `410 invalid_file_type` – wskazany plik nie może zostać wykorzystany podczas pewnej operacji. Błąd ten zgłaszany jest np. przez komendy `cp` i `mv` kiedy klient próbuje skopiować katalog,
- `411 dls_without_dl` – klient próbował użyć instrukcji `dls` bez wcześniejszego wywołania `dl`,
- `412 in_use` – użytkownik próbuje usunąć lub zmienić nazwę pliku (lub katalogu), który jest aktualnie wykorzystywany przez innego użytkownika,
- `413 wrong_upload_path` – użytkownik próbuje ponowić ładowanie pliku, ale nie znajduje się w katalogu w którym operacja ta była przerwana,
- `414 not_a_directory` – użytkownik próbował podać ścieżkę do pliku zamiast do katalogu w polu `path` komend takich jak np. `cd` czy `mv`,
- `415 too_large_file` – użytkownik próbował załadować zbyt duży plik na serwer.

3.2. Uprawnienia użytkowników

- `write`, `w` – uprawnienie dające dostęp do komend modyfikujących strukturę systemu plików, np. `mkdir`, `rm`,
- `copy`, `c` – uprawnienie dające dostęp do komendy kopiującej `cp`,
- `copy`, `m` – uprawnienie dające dostęp do komendy przenoszącej `mv`,
- `download`, `d` – uprawnienie dające dostęp do komend odpowiadających za pobieranie plików: `dl`, `dls`,
- `upload`, `u` – uprawnienie dające dostęp do komendy odpowiadającej za ładowanie plików (`ul`).

Podstawowe komendy takie jak `cd`, `ls` czy `exit` nie wymagają uprawnień. Użytkownik zaraz po utworzeniu ma tylko uprawnienie do pobierania plików (`d`).

3.3. Polecenia

3.3.1 Polecenie `auth`

Polecenie `auth` musi być pierwszym poleceniem wykonanym przez połączonego klienta – bez poprawnej autoryzacji nie ma możliwości wykonywania jakichkolwiek innych komend. Dostępne pola:

- `login` – pole typu `string` zawierające nazwę użytkownika,
- `passwd` – pole typu `string` zawierające hasło.

3.3.2 Polecenie `logout`

Polecenie `logout` pozwala na wylogowanie się z aktualnego konta i przejście na inne.

3.3.3 Polecenie `exit`

Polecenie `exit` pozwala na bezpieczne rozłączenie się z serwerem. Jest to jedyne polecenie poza `auth`, które może być użyte bez bycia zalogowanym. Dodatkowo polecenie to posiada alias `bye`.

3.3.4 Polecenie `cd`

Polecenie `cd` służy do zmiany aktualnego katalogu użytkownika. Dostępne pola:

- `path` – pole typu `string` zawierające względną lub bezwzględną ścieżkę.

Polecenie to zwraca nową ścieżkę, o ile jego wykonanie się powiodło.

3.3.5 Polecenie `name`

Polecenie `name` zwraca nazwę użytkownika, na którego konto zalogowany jest klient.

3.3.6 Polecenie `perms`

Polecenie `perms` zwraca uprawnienia użytkownika, na którego konto zalogowany jest klient, w postaci ciągu znaków odpowiadających pierwszym literom nazw uprawnień.

3.3.7 Polecenie `pwd`

Polecenie `pwd` zwraca katalog, w którym aktualnie znajduje się użytkownik. Serwer w odpowiedzi w jednej linii zwróci ten katalog.

3.3.8 Polecenie `ls`

Polecenie `ls` zwraca listę plików dostępnych w danym katalogu. Dostępne pola:

- `path` – opcjonalne pole typu `string` zawierające ścieżkę do katalogu, którego pliki powinny zostać wypisane. Jeżeli pole to nie zostało określone to wykorzystany będzie aktualny katalog użytkownika,
- `size` – opcjonalne pole typu `boolean`. Określa ono, czy użytkownik chce otrzymać w odpowiedzi rozmiary plików. W przypadku katalogów zwrócony zostanie znak `-` (kod ASCII `0x2D`),
- `mod` – opcjonalne pole typu `boolean`. Jego zadaniem jest określenie, czy użytkownik chce otrzymać w odpowiedzi datę ostatniej modyfikacji plików w postaci "`DD.MM.YYYY HH:MM:SS`". Jeżeli serwer nie będzie w stanie pobrać tej daty to zwrócony zostanie znak `-` (kod `0x2D`),

W odpowiedzi serwera w kolejnych liniach podane zostaną nazwy plików w cudzysłowach oraz dodatkowe parametry, w takiej kolejności jaka jest podana wyżej. Przykładowa odpowiedź serwera, gdy wszystkie pola opcjonalne mają wartość `true`:

```
100 ls
"test.txt" 4096 "25.10.2021 09:24:14"
"test2.txt" 8192 "30.12.2020 13:23:10"
"directory" - "20.06.2019 20:55:09"
```

3.3.9 Polecenie `tree`

Polecenie `tree` zwraca drzewo plików zaczynając od katalogu wskazanym w polu `path` typu `string`. Jeżeli pole to nie zostanie podane, to drzewo zostanie wypisane od aktualnego katalogu klienta. O zagnieżdżeniu pliku świadczy ilość spacji poprzedzających jego nazwę, tj. jedna spacja to jeden stopień w dół. Przykładowa odpowiedź serwera na to polecenie:

```
100 tree
"Documents"
     "CV.docx"
     "Essay.pdf"
"Programming"
     "HelloWorld"
          "HelloWorld.txt"
```

"Empty" E

Litera `E` po nazwie oznacza, że jest to pusty katalog. Jej zadaniem jest umożliwienie użytkownikowi odróżnienia plików od pustych katalogów.

3.3.10 Polecenie `mkdir`

Polecenie `mkdir` tworzy nowy katalog w aktualnym katalogu. Dostępne pola:

- `name` – pole typu `string` zawierające nazwę nowego katalogu.

3.3.11 Polecenie `rm`

Polecenie `rm` usuwa plik z aktualnego katalogu. Dostępne pola:

- `name` – pole typu `string` określające plik do usunięcia.

Polecenie w jednej linii odpowiedzi zwraca ilość usuniętych plików.

3.3.12 Polecenie `rename`

Polecenie `rename` zmienia nazwę określonego pliku znajdującego się w aktualnym katalogu. Dostępne pola:

- `oname` – pole typu `string` określające nazwę pliku, którego nazwa będzie zmieniona,
- `nname` – pole typu `string` zawierające nową nazwę.

3.3.13 Polecenie `cp`

Polecenie `cp` służy do kopiowania pliku z aktualnego katalogu w inne dostępne na serwerze miejsce. Dostępne pola:

- `name` – pole typu `string` określające nazwę pliku, który będzie kopiowany,
- `path` – ścieżka do katalogu, w którym znajdzie się kopia pliku.

3.3.14 Polecenie `mv`

Polecenie `mv` służy do przenoszenia pliku z aktualnego katalogu w inne dostępne na serwerze miejsce. Dostępne pola:

- `name` – pole typu `string` określające nazwę pliku, który będzie przenoszony,
- `path` – ścieżka do katalogu, w którym znajdzie się plik.

3.3.15 Polecenie `dl`

Wykonanie polecenia `dl` podzielone jest na dwa etapy:

1. Najpierw klient wysyła do serwera komunikat o nazwie `dl` z polem `name` (nazwa pobieranego pliku z aktualnego katalogu). Kod odpowiedzi będzie decydował o tym czy klient uzyskał zgodę na pobieranie czy nie. W odpowiedzi w jednej linii zawarta też będzie ilość bajtów, które powinien odczytać klient. Przykładowa odpowiedź serwera:
   ```
   100 dl
   3742456
   ```
   Serwer zapisze w swoich strukturach informację o pliku, który będzie pobierany. Będzie ona ważna aż do wywołania komendy `dls`. Do momentu wykonania tej komendy plik nie będzie mógł zostać usunięty przez innych użytkowników.

2. Drugim komunikatem wysyłanym przez klienta jest `dls` ("download start") rozpoczynający pobieranie. Serwer nie wyśle odpowiedzi na to zapytanie – od razu zacznie wysyłać bajty pliku wskazanego w ostatnio wykonanej instrukcji `dl`.

Komunikat rozpoczynający pobieranie `dl` może przyjmować także dodatkowy parametr typu `integer` o nazwie `offset`. Może być on przydatny w sytuacji, gdy połączenie z serwerem zostało przerwane i klient chce rozpocząć pobieranie od pewnego momentu. Po wykonaniu polecenia `dl` serwer z tym parametrem serwer zwróci ilość bajtów do przeczytania, czyli rozmiar pliku minus przekazany parametr `offset`.

3.3.16 Polecenie `ul`

W celu załadowania pliku do aktualnego katalogu klient wysyła do serwera komunikat `ul`. Zawiera on następujące pola:

- `name` – pole typu `string` określające nazwę pliku,
- `size` – pole typu `integer` określające rozmiar pliku,
- `retry` – pole typu `boolean` określające czy jest to wznawiane ładowanie.

W zależności od kodu operacji klient powinien:

- Kod pozytywny (100) – klient w odpowiedzi otrzyma liczbę, która oznacza ilość pierwszych bajtów z pliku do pominięcia. Jest to przydane przy próbie ponowienia ładowania. Serwer także od razu przejdzie w tryb ładowania danych – wszystkie dane które wyśle klient są od tego momentu dopisywane do pliku. Trwa to dopóki serwer nie przeczyta $S - O$ bajtów, gdzie S to rozmiar pliku, który zadeklarował klient, a O to liczba wysłana przez serwer w odpowiedzi,
- Kod negatywny – klient nie może rozpocząć ładowania pliku, serwer pozostaje w stanie interpretacji poleceń protokołu.

Przy ponawianiu operacji istotne jest, aby pola `name` i `size` były takie same jak przy pierwszym ładowaniu. Ponadto klient musi znajdować się w tym samym katalogu, do którego wcześniej ładował plik.

4. Serwer

4.1. Słownik pojęć

Pojęcia używane w tej sekcji:

1. TDS – TinDox Server.
2. Instancja serwera – katalog zawierający katalog podrzędny o nazwie `.tds`, w którym znajdują pliki konfiguracyjne potrzebne do uruchomienia serwera. Instancja jest również korzeniem systemu plików widocznego przez klienta.
3. IO - wejście i wyjście.
4. Klient (*Client*) – każdy komputer podłączony w danej chwili do serwera.
5. Użytkownik (*User*) – każdy potencjalny klient. Lista użytkowników zapisana jest w pliku `users`.

4.2. Wymagania funkcjonalne

1. Możliwość szybkiego utworzenia instancji serwera w dowolnym miejscu na dysku – umożliwia to instrukcja "`tds init`".
2. Tworzenie, modyfikacja i usuwanie użytkowników uprawnionych do korzystania z dysku sieciowego – umożliwiają to instrukcje "`tds user ...`".
3. Realizacja komunikacji z klientami z wykorzystaniem gniazd BSD – w tym celu powstał moduł "`tds::ip`".
4. Wydajne IO dzięki wykorzystaniu zasobów systemowych:
   - Wykorzystanie efektywnej liczby wątków do obsługi wielu klientów jednocześnie – w celu realizacji wydajnej wielowątkowości powstała klasa "`tds::server::ClientService`",
   - Wykorzystanie linuksowego mechanizmu `epoll` do sprawnej obsługi klientów – w tym celu powstała klasa "`tds::linux::EpollDevice`" oraz inne klasy pomocnicze.
5. Bezbłędne zamykanie połączeń z klientami oraz całego serwera – odpowiadają za to klasy z modułu "`tds::server`".
6. Prawidłowe reagowanie na sygnały systemowe (np. użycie `Ctrl+C` w terminalu powinno prawidłowo zakończyć działanie serwera) – w tym celu powstała klasa "`tds::linux::SignalDevice`".
7. Monitorowanie połączeń przychodzących oraz błędów serwera, wypisywanie logów – do realizacji tej funkcjonalności została użyta biblioteka "`spdlog`".

4.3. Wymagania niefunkcjonalne

1. Konfigurowalność – serwer będzie wykorzystywał plik konfiguracyjny w formacie TOML. Edytując go, użytkownik będzie mógł dostosować serwer do możliwości swojego komputera, co oznacza między innymi:
   - Możliwość wyboru maksymalnej ilości wątków używanych przez serwer (domyślnie będzie to wartość zwracana przez funkcję z języka `C++` – `std::thread::hardware_concurrency()`),
   - Możliwość wyboru maksymalnej ilości klientów w sesji,
   - Możliwość zmiany portu, na którym będzie nasłuchiwał serwer,
   - Możliwość wyboru maksymalnego rozmiaru pliku, który może zostać załadowany na serwer.
2. Wydajność – serwer powinien sprawnie obsługiwać wiele żądań jednocześnie przy wykorzystaniu optymalnej ilości zasobów systemu operacyjnego.
3. Bezpieczeństwo – serwer powinien być odporny na złośliwe zapytania. Oznacza to, że w aplikacji nie wystąpią podatności takie jak np. RCE.

4.4. Polecenia linii komend

4.4.1 Polecenie `help`

Polecenie `help` służy do wyświetlenia pomocy, czyli między innymi listy dostępnych komend.

4.4.2 Polecenie `init`

Polecenie `init` służy do utworzenia instancji serwera, czyli katalogu "`.tds`", pliku "`.tds/config`" zawierającego modyfikowalne parametry serwera oraz pliku "`.tds/users`" zawierającego listę użytkowników wraz z ich hasłami i uprawnieniami. Może być wywołane bez parametrów, wówczas instancja zostanie utworzona w bieżącym katalogu, lub z jednym argumentem, który jest ścieżką do istniejącego, docelowego folderu. Przykłady użycia:

- `tds init` – utworzenie instancji w bieżącym katalogu,
- `tds init .` – to samo co wyżej,
- `tds init $HOME` – utworzenie instancji w katalogu domowym użytkownika.

Serwer zaraz po utworzeniu ma tylko jednego dostępnego użytkownika o nazwie `admin`, który posiada wszystkie dostępne uprawnienia.

4.4.3 Polecenie `run`

Polecenie `run` służy do uruchamiania serwera. Może być wywołane bez parametrów, wówczas serwer zostanie uruchomiony w bieżącym katalogu, o ile jest to prawidłowa instancja. Dodatkowo polecenie `init` definiuje trzy flagi:

- `--path [ścieżka]` – wybór innej instancji serwera niż bieżąca,

- `--port [port]` – wybór innego portu (domyślnie port jest ładowany z pliku `config`),
- `--debug` – wyświetlenie dodatkowych informacji o przebiegu pracy serwera.

Przykłady użycia:

- `tds run` – uruchomienie instancji serwera w bieżącym katalogu,
- `tds run --path .` – to samo co wyżej,
- `tds run --port 80` – uruchomienie instancji serwera na porcie 80,
- `tds run --path $HOME --port 3000` – uruchomienie instancji serwera w katalogu domowym użytkownika na porcie 3000.

4.4.4 Polecenie `user`

Polecenie `user` służy do tworzenia nowych użytkowników, zmiany ich haseł, usuwania ich oraz do dodawania i odbierania uprawnień. Przykłady:

- `tds user add` – rozpoczęcie dialogu, w trakcie którego zostaną podane login i hasło oraz powstanie nowy użytkownik,
- `tds user passwd admin` – zmiana hasła użytkownika `admin` poprzez dialog,
- `tds user perms admin +u -d` – zmiana uprawnień użytkownika `admin` – od teraz może on ładować pliki na serwer, ale nie może ich pobierać. Lista liter odpowiadających uprawnieniom może być wyświetlona z wykorzystaniem instrukcji `tds help`,
- `tds user remove admin` – usunięcie użytkownika `admin`.

4.4.5 Polecenie `version`

Polecenie `version` służy do wyświetlenia aktualnej wersji serwera.

4.5. Opis techniczny

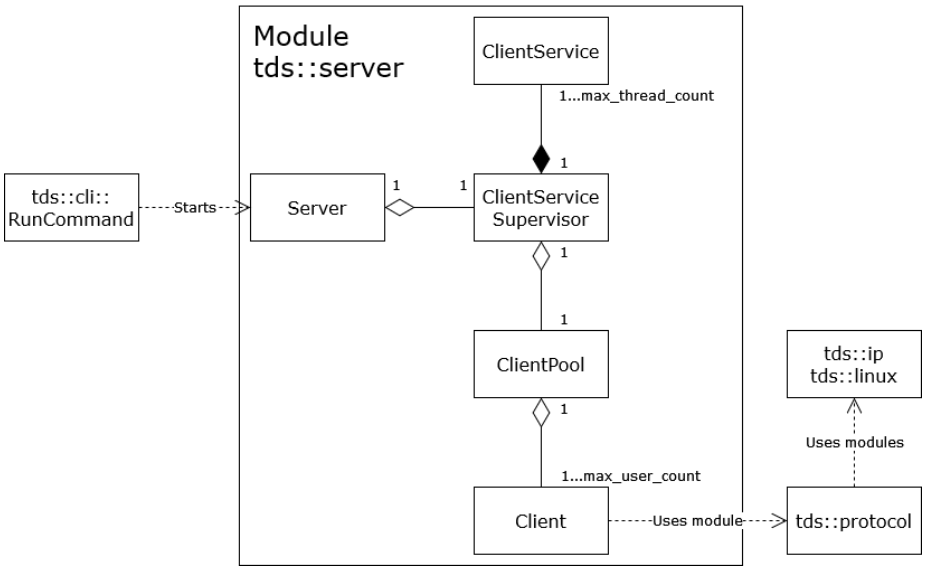
4.5.1 Moduły

Projekt składa się z następujących modułów:

- `cli` – moduł odpowiadający za obsługę linii komend.
- `cli::user_command` – podmoduł odpowiadający za obsługę polecenia `tds user [...]`.
- `command` – moduł zawierający klasy służące do wykonywania poleceń (wzorzec "Polecenie"). Jest on bezpośrednio używany tylko przez moduły `cli` (implementacja obsługi poleceń linii komend) oraz `protocol` (implementacja poleceń klienta).

- `config` – moduł odpowiadający za obsługę plików konfiguracyjnych serwera. Jest on używany bezpośrednio przez moduły `cli` (tworzenie domyślnych plików konfiguracyjnych) oraz `server` (wczytywanie konfiguracji lub pobieranie domyślnej konfiguracji w przypadku braku pliku "`config`").
- `ip` – moduł udostępniający klasy i funkcje związane ze stosem sieciowym, np. abstrakcję adresów IP, obsługę gniazd. Stanowi on rozbudowanie modułu "`linux`".
- `linux` – moduł udostępniający klasy i funkcje związane z systemem Linux, np. obsługę mechanizmu systemowego `epoll`, sygnałów systemowych i potoków.
- `protocol` – moduł odpowiadający za obsługę protokołu TinDox. Zawiera między innymi interpreter poleceń, obsługę błędów protokołu czy realizację modelu "Receiver–Sender". Korzysta on bezpośrednio z modułów "`linux`" oraz "`ip`" – jest wyższą warstwą abstrakcji.
- `protocol::execution` – podmoduł zawierający klasy odpowiadające za wykonywanie i odpowiadanie na polecenia protokołu TinDox. Jest używany przez wykonawcę komend z modułu "`protocol`".
- `server` – moduł zawierający klasy najwyższego poziomu odpowiadające za uruchamianie i działanie serwera. Ponadto kontroluje on działania klientów oraz odpowiada za obsługę błędów.
- `user` – moduł odpowiadający za parametry użytkowników serwera (np. uprawnienia).

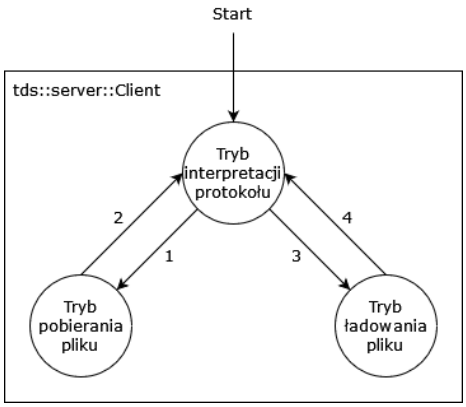
4.5.2 Schemat komunikacji z głównym modułem – `tds::server`



Rysunek 1: Schemat komunikacji z głównym modułem

4.5.3 Tryby pracy klienta

Dla każdego podłączonego klienta klasa `tds::server::ClientPool` tworzy instancję klasy `tds::server::Client`, która zawiera podstawowe obiekty potrzebne do przeprowadzenia komunikacji z klientem, czyli między innymi gniazdo, interpreter protokołu czy menadżera pobierania plików. Zasadniczo klasa ta jest automatem skończonym o następujących stanach:



Rysunek 2: Schemat przechodzenia między trybami klienta

Opis trybów pracy:

1. Tryb interpretacji protokołu – w tym trybie wszystkie dane wejściowe traktowane są jako fragmenty poleceń. Są one przekazywane do interpretera poleceń (instancja klasy "`tds::protocol::ProtocolInterpreter`"), który w momencie wczytania pełnego polecenia próbuje utworzyć obiekt klasy "`tds::protocol::Request`" i przekazuje go do wykonawcy komend ("`tds::protocol::DefaultCommandExecutor`").

2. Tryb pobierania pliku – w tym trybie dane wejściowe są zablokowane. Instancja obsługująca klienta wykorzystując menadżera pobierania (obiekt klasy "`DownloadManager`") wysyła dane w najwydajniejszy możliwy sposób bezpośrednio do klienta.

3. Tryb ładowania pliku – w tym trybie wszystkie dane wejściowe są traktowane jako fragmenty pliku. Instancja obsługująca klienta wykorzystując menadżera ładowania (obiekt klasy "`UploadManager`") tworzy plik i monitoruje status operacji.

Przejścia w schemacie:

1. Przejście w tryb pobierania pliku – jest to możliwe tylko w przypadku wykonania polecenia "`dls`" przez klienta.

2. Przejście z trybu pobierania do trybu interpretacji – następuje automatycznie po przeczytaniu przez klienta wszystkich bajtów pobieranego pliku.

3. Przejście w tryb ładowania pliku – jest to możliwe tylko w przypadku pozytywnego wykonania polecenia "`ul`" przez klienta.

4. Przejście z trybu ładowania do trybu interpretacji – następuje automatycznie po wczytaniu przez serwer wszystkich bajtów pliku.

4.6. Narzędzia i biblioteki

4.6.1 Narzędzia

Element	Narzędzia	Wersja
Język programowania	`C++`	ISO/IEC 14882:2020
Kompilator	`g++` `clang`	11.1.0 13.0.0
System budowania	`CMake`	3.18.4
Automatyzacja testów	`CTest`	3.18.4
Platforma docelowa	`Linux x86-64`	4.0 – 5.16

4.6.2 Biblioteki

Nazwa	Wersja	Opis
`Catch2`	3.0.0	Tworzenie testów jednostkowych
`tomlplusplus`	2.5.0	Obsługa formatu TOML
`{fmt}`	8.0	Formatowanie tekstów (Odpowiednik `std::format` z C++20)
`spdlog`	1.9	Tworzenie logów (Używane przez klasy z modułu `tds::server`)

5. Klient mobilny

5.1. Opis klienta

Połączenie z serwerem za pomocą klienta w aplikacji mobilnej zaimplementowane z użyciem języka kotlin w środowisku Android Studio. Za ich pomocą powstanie prosta aplikacja mobilna z interakcyjnym interfejsem graficznym reprezentująca nasz zdalny system plików. Aplikacja będzie budowana na system Android.

5.2. Narzędzia i biblioteki

Element	Narzędzia
Język programowania	`Kotlin`
Kompilator	`kotlinc`
IDE	`Android Studio`
Platforma docelowa	`Android`

6. Klient okienkowy

6.1. Opis klienta

Połączenie z serwerem realizowane przez klienta okienkowego zaimplementowanego z użyciem języka Java i klasy klasy z pakietu java.nio implementującej nieblokujące IO. Pakiet ten jest częścią biblioteki standardowej Javy od wersji 1.4 i dostarcza narzędzi służących do przeprowadzania operacji wejścia/wyjścia zarówno w sposób blokujący, jak i nieblokujący. Po uruchomieniu klient podejmie próbę połączenia z serwerem. W przypadku udanego połączenia aplikacja umożliwi interakcję ze zdalnym systemem plików.

6.2. Podział na klasy

- `Connection` – zawiera podstawowe elementy połączenia, czyli SelectionKey, SocketChannel, oba ByteBuffery -wysyłający i odbierający.
- `Client` – obejmuje główną logikę aplikacji.
- `ResponseAnalyzer` – odpowiada za interpretacje wiadomości odebranych od serwera.
- `ClientGUI` – interfejs użytkownika.

6.3. Narzędzia

Element	Narzędzia
Język programowania	`Java`
Kompilator	`javac`

7. Klient konsolowy

7.1. Opis klienta

Połączenie z serwerem realizowane poprzez klienta zaimplementowanego przy użyciu biblioteki `FTXUI` i języka `C++`. FTXUI *(Functional Terminal (X) User interface)* to biblioteka pozwalająca stworzyć zgrabne TUI. Wykorzystując jej elementy powstało narzędzie do wykonywania operacji plikowych na zdalnym systemie plików.

Klient konsolowy przeznaczony jest na system `Ubuntu`. Jest kompilowany z wykorzystaniem `g++` oraz budowany z wykorzystaniem `CMake`.

Klient jest uruchamiany poprzez wywołanie pliku wykonywalnego **./cli** i wymaga zdefiniowanego katalogu *TinDox*, który jest automatycznie tworzony.

Narzędzie zostało przetestowane jedynie z poziomu funkcjonalności dostępnych dla użytkownika (*UAT*). W ramach zrealizowanych scenariuszy testowych skupiono się na procesach podstawowych oraz przypadkach granicznych (brak łączności z serwerem w trakcie realizacji funkcjonalności, obsługa spacji w nazwach plików i katalogów).

7.2. Narzędzia i biblioteki

Element	Narzędzia	Wersja
Język programowania	`C++`	ISO/IEC 14882:2020
Kompilator	`g++`	11.2.0
System budowania	`CMake`	3.22.0
TUI	`FTXUI`	2.0.0
Platforma docelowa	`Linux x86-64`	4.0.0

8. Testowanie

8.1. Testy jednostkowe

8.1.1 Testy serwera

Testy jednostkowe serwera znajdują się w katalogu "`server/tests/unit`". Weryfikują one działanie podstawowych modułów systemu takich jak na przykład "`tds::linux`", który odpowiada za między innymi prawidłowe reagowanie na sygnały systemowe, czy "`tds::ip`", który odpowiada za nawiązywanie i obsługę połączeń. Ponadto testy te implementują *fuzzing* do sprawdzania działania interpretera poleceń na nieprawidłowe dane wejściowe.

Są to testy automatyczne, można je uruchomić z wykorzystaniem programu `ctest`: "`ctest -R UNIT`".

8.2. Testy integracyjne

8.2.1 Testy serwera

Testy integracyjne znajdują się w katalogu "`server/tests/integration`". Zostały napisane w języku skryptowym `Bash`. Ich zadaniem jest sprawdzanie reakcji programu na polecenia linii komend. Ponadto sprawdzana jest prawidłowość wykonania niektórych poleceń klienta z wykorzystaniem programu `netcat`.

Są to testy automatyczne, można je uruchomić z wykorzystaniem programu `ctest`: "`ctest -R INTEGRATION`".

8.3. Testy penetracyjne

Testy penetracyjne serwera są wykonywane manualnie. Zakładamy, że osoba atakująca serwer będzie próbowała wykonować następujące akcje:

1. Atakujący próbuje dostać się do katalogów zabronionych, czyli takich do których dostęp powinien mieć tylko właściciel serwera, z wykorzystaniem polecenia "`cd`". Serwer powinien w takim przypadku zawsze skierować użytkownika do katalogu głównego.

2. Atakujący wysyła losowe ciągi bajtów w celu doprowadzenia do błędu interpretera. W takiej sytuacji moduł odpowiadający za obsługę klienta powinien zadbać o to, aby interpreter nie znalazł się w stanie niedozwolonym (np. nie powinien wpaść w pętlę nieskończoną).

3. Atakujący wysyła wiele prawidłowych poleceń na raz. W takiej sytuacji serwer powinien wysłać wiele odpowiedzi w jednym momencie oraz nie powinien dopuścić do nadmiernej alokacji pamięci na przyjęte dane.

8.4. Testy empiryczne

Testy manualne wykonywane w ramach testów empirycznych:

1. Zweryfikowanie reakcji serwera na nieistniejące lub nieprawidłowo sformułowane polecenia z wykorzystaniem programu `netcat`,

2. Zweryfikowanie wszystkich poleceń serwera z wykorzystaniem programu `netcat`,

3. Wykonanie wszystkich poleceń z wykorzystaniem każdego klienta z osobna,
4. Przetestowanie poprawności obsługi wszystkich scenariuszy alternatywnych dla poleceń,
5. Sprawdzenie pobierania plików o różnych rozmiarach (od kilku kilobajtów do dwóch gigabajtów),
6. Sprawdzenie ładowania plików o różnych rozmiarach,
7. Przerywanie połączenia w trakcie ładowania lub pobierania, oraz wznawianie tych operacji.

9. Zespół

9.1. Środowisko deweloperskie

Element	Narzędzia	Wersja

System operacyjny	`Ubuntu`
	`Manjaro Linux`
	`Windows`
	`Arch Linux`
		20.04, 21.10
		21.1.6
		10.0
		—

Pomocniczy język skryptowy[1]	`Bash`	5.1.8
Kontrola wersji	`git`	2.32.0

Repozytorium
ITS[2]
Tablica kanban
	`GitHub`
	`GitHub Issues`
	`GitHub Projects`
		—

CI/CD	`Github Actions`	—
Dokumentacja	`Overleaf`	—

9.2. Podział pracy

Projekt	Wykonawca
Serwer	Jakub Mazurkiewicz
Klient mobilny	Damian Piotrowski
Klient okienkowy	Anna Pyrka
Klient konsolowy	Łukasz Reszka

[1] Języki skryptowe będą używane do np. symulowania złożonych przypadków testowych, automatyzacji testów.
[2] Issue tracking system.