# TFTP

Piotr Mikołajczyk

## 1. OPIS ROZWIĄZANIA

### 1.1 Wstęp

Implementacja protokołu *TFTP* znajduje się w trzech plikach *.py:*

- *client.py* - zawiera klasę *clientTFTP,* która realizuje funkcjonalność strony klienta
- *server.py* - zawiera klasę *serverTFTP,* która realizuje funkcjonalność strony serwera
- *common.py* - zawiera zmienne konfiguracyjne dla obu stron oraz metody wykorzystywane do wysyłania/odbierania pakietów

Funkcjonalności obu stron są ograniczone jedynie do scenariusza, w którym klient czyta plik serwowany przez serwer. Jedyną opcją negocjowaną jest długość okna.

Domyślnie, w każda strona ma włączone logowanie. W celu wyłączenia należy wykomentować linię:

*logging.basicConfig(level=logging.DEBUG)*

### 1.2 Strona serwera

Serwer można uruchomić z 0-2 argumentami, które kolejno oznaczają: port, na którym serwer ma nasłuchiwać oraz ścieżkę z do katalogu, z którego ma serwować pliki. Domyślna wartość portu ustawiona jest w pliku *common.py*, natomiast ścieżka jest domyślnie ustawiona na bieżącą.

Funkcje *__init__()* oraz *start()* przygotowują do pracy socket. Główna pętla programu znajduje się w metodzie *monitor()*. Serwer oczekuje w niej na przychodzące requesty. Jeśli taki nadejdzie, tworzony jest obiekt klasy wewnętrznej *connection_handler*, który przejmuje odpowiedzialność za komunikację z nowym klientem.

#### 1.2.1 *connection_handler*

Obiekty tej klasy uruchamiane są w osobnym wątku. Po zainicjalizowaniu potrzebnych pól oraz zajęcia losowo wygenerowanego portu (metody *__init__(), run()*), następuje parsowanie requesta (*parse_request*()). Akceptujemy jedynie *RRQ* z dowolnie wybraną przez klienta szerokością okna.

W funkcji *read_request()* dokonuje się cała dalsza komunikacja z klientem. W słowniku *history* mapujemy przeczytane z pliku i jeszcze niepotwierdzone przez klienta bloki z danymi. Wybrana polityka wysyłania pliku zakłada działanie w ściśle określony sposób. W pętli, która kończy się jedynie w przypadku zerwania połączenia, otrzymania błędu od klienta lub pomyślnego zakończenia komunikacji, najpierw wysyłamy pewną ilość pakietów z danymi (określoną przez wynegocjowaną wartość *windowsize*). Następnie zawieszamy się w oczekiwaniu na potwierdzenie od klienta.

1. W przypadku timeoutu, ponawiamy wysyłanie całego ostatniego okna pakietów.

2. Jeśli otrzymamy potwierdzenie ostatnio wysyłanego pakietu, kontynuujemy wysyłanie, jako że dotąd wszystko udało się przesłać pomyślnie.
3. Jeśli otrzymamy potwierdzenie pakietu z ostatniego okna (nie ostatniego), to zaczniemy kolejne okno od pierwszego niepotwierdzonego kawałka.
4. Jeśli otrzymamy potwierdzenie pakietu z przyszłości/przeszłości to go ignorujemy.

### 1.2 Strona klienta

Klienta uruchamiamy z dwoma argumentami – pierwszy odpowiada nazwie serwera, z którym chcemy się połączyć, drugi jest nazwą pliku, który chcemy odczytać.

Inicjalizacja pól oraz używanego socketu odbywa się w metodzie *__init__()*. Tu można zmienić preferowaną długość okna.

Metoda *establish_connection()* odpowiada za nawiązanie połączenia z serwerem, negocjację długości okna. Jeśli klient chce czytać z długością niewspieraną przez serwer, następuje zerwanie połączenia oraz wyświetlenie odpowiedniego komunikatu zachęcającego do zmiany długości okna.

Główna komunikacja z serwerem znajduje się w metodzie *read_request().* Dopóki nie potwierdzimy niepełny (o długości < 512B) pakiet, lub napotkamy na wiadomość z błędem lub nie stracimy połączenia, działamy w pętli. Najpierw staramy się odebrać określoną liczbę pakietów (tyle ile liczy wynegocjowana długość okna). Timeout powoduje przerwanie oczekiwania. Następnie patrzymy na każdy z odebranych pakietów w kolejności rosnących numerów. W zależności od potrzeb, albo hashujemy MD5, albo wyświetlamy odebrany kawałek pliku. Wysyłamy potwierdzenie pakietu będący wartością mex(wszystkie odebrane dotychczas) - 1.

### 1.4 *common.py*

W tym pliku zaimplementowane są proste metody konwersji liczb całkowitych do postaci 2-bajtowej w zapisie heksadecymalnym (w drugą stronę również), testowanie czy dany pakiet jest pakietem błędu, wysyłanie pakietów z odpowiednimi opcode'ami oraz nasłuchiwanie w pętli z możliwością retransmisji jednego lub wielu z ostatnich pakietów.

Zmienna *default_port* określa domyślny port, na którym serwer powinien nasłuchiwać, a klient łączyć się w celu nawiązania komunikacji.

Zmienne *timeout* oraz *attempts* regulują politykę oczekiwania na pakiety.

## 2. OPIS TESTÓW

Komunikacja testowana była na *Satori*, z serwerem *atftp* oraz na *localhost* na plikach o zróżnicowanych rozmiarach (od kilkunastu kilobajtów do kilkuset megabajtów). Dodatkowo, w testach na lokalnej maszynie, testowane były losowe opóźnienia transmisji oraz gubienia pakietów. Poza S*atori*, testy obejmowały różne długości okna: od 1 do 48.