Projekt *TFTP* na Sieci Komputerowe

Artur Zubilewicz

1. **Opis i uruchomienie.**

   Celem projektu było zaimplementowanie własnego klienta i serwera *TFTP (Trivial File Transfer Protocol)* na podstawie dokumentów *RFC 1350*, *RFC 2347*, *RFC 2348*, *RFC 2349* i *RFC 7440*.

   Należało napisać serwer, który wysyłał pliki do klienta, a więc obsługiwane miało być polecenie *RRQ (Read Request)*. Następnie klient obliczał hash *md5* otrzymanego pliku i zwracał ją na output.

   Aby uruchomić pobieranie należy najpierw uruchomić *server.py* podając mu dwa argumenty pozycyjne *directory* (folder, z którego mają być brane pliki do pobierania przez klienta) i *port* (na którym ma nasłuchiwać przychodzących pakietów UDP). Można również włączyć symulację szumu poprzez dodanie flagi *--noise*, wówczas wysłanie pakietu powiedzie się z prawdopodobieństwem 90%.

   Następnie należy uruchomić *client.py* podając mu dwa argumenty pozycyjne *server* (nazwa serwera, z którego ma być pobrany plik, np. jego adres publiczny albo localhost dla obecnej maszyny) i *filename* (nazwa pliku do pobrania). Można również dostarczyć argumenty modyfikujące jego działanie, jak np.:

   - *--blocksize (RFC 2348)* - rozmiar pojedynczego bloku danych pliku przesyłanego w pakiecie TFTP; liczba całkowita dodatnia,
   - *--timeout (RFC 2349)* - czas, który musi minąć zanim klient i serwer przestaną czekać na pakiet TFTP i wyślą ponownie swój ostatni pakiet; liczba rzeczywista dodatnia,
   - *--windowsize (RFC 7440)* - liczba pakietów, które mogą być wysłane zanim nastąpi czekanie na ACK (acknowledgement) od klienta z informacją, że otrzymał pakiety; wartość całkowita dodatnia,
   - *--retries (opcja własna)*, pozwala ustalić liczbę timeoutów zanim klient jak i serwer poddadzą się przy kolejnych nieudanych czekaniach i zakończą działanie; liczba całkowita dodatnia.

2. **Implementacja.**

   Kod symulatora został napisany w języku *Python 3.6*. Implementacja składa się z dwóch plików:

   - *client.py*
   - *server.py*

   Kod klienta składa się z metod, które pozwalają wysyłać pakiety ACK, $ERROR$ oraz RRQ w zgodzie z RFC 2347, to jest z możliwością dodawania opcji.

   Kod serwera składa się z metod, które pozwalają wysyłać pakiety $DATA$, $ERROR$ i $OACK$, również zgodnie z RFC 2347.

3. **Testy.**

   Kod przetestowałem na swojej lokalnej maszynie poprzez utworzenie kilku plików o różnych rozmiarach i przesłanie ich za pomocą serwera i porównanie wynikowego hasha $md5$ z oryginalnym.

   Pliki starałem się utworzyć tak, aby jak najniekorzystniej wpłynęły na działanie programów, tj. pozwalały zapętlać się numerom bloków $DATA$, sama długość pliku była długości podzielnej przez długość bloku, itp.

   Dla przykładu, utworzyłem trzy pliki, pierwszy posiadający domyślny test, tj. *echo "Zażółć gęślą jaźń!" > test.txt* oraz dwa 'duże' pliki złożone z kopii znaku 'a': *printf 'a%.0s' {1..33554432} > BIG32MiB.txt* i *printf 'a%.0s' {1..67108864} > BIG64MiB.txt* (gdzie $67108864 = 2 \cdot 33554432 = 2 \cdot 512 \cdot 256 \cdot 256$).

   Uruchomienie serwera i klienta z domyślnymi ustawieniami jak również z przykładowymi flagami *--blocksize 123 --timeout 2.0 --windowsize 100 --retries 5* zwracało te same wyniki, które pokrywały się z obliczonymi wcześniej wartościami haszy $md5$ za pomocą programu $md5sum$.

   Testowałem również kod w przypadku 'szumu', tj. w sytuacji, gdy niektóre pakiety znikają na drodze między klientem i serwerem. Symulację takiego zachowania przeprowadziłem poprzes losowanie liczby z przedziału [0..1] i jeżeli wylosowana liczba była mniejsza niż 0.5 to nie wysyłałem pakietu.