# Zadanie programistyczne nr 3
# z Sieci komputerowych

## 1 Opis zadania

Napisz program `transport`, który będzie łączyć się z określonym serwerem i wysyłając — zgodnie z opisanym niżej protokołem — pakiety UDP, pobierze od serwera plik o określonej wielkości.

Program powinien przyjmować cztery argumenty: *adres_IP*, *port*, *nazwa_pliku* i *rozmiar* (w bajtach). Celem jest pobranie od specjalnego serwera UDP nasłuchującego na porcie *port* komputera o adresie *adres_IP* pierwszych *rozmiar* bajtów pliku i zapisanie ich w pliku *nazwa_pliku*. W celu komunikacji z serwerem UDP należy wysyłać mu datagramy UDP zawierające następujący napis:

`GET` *start długość*`\n`

Wartość *start* powinna być liczbą całkowitą z zakresu $[0, 10\,000\,000]$, zaś *długość* z zakresu $[1, 1\,000]$. Znak `\n` jest uniksowym końcem wiersza, zaś odstępy są pojedynczymi spacjami. Jedyną zawartością datagramu musi być powyższy napis: serwer zignoruje datagramy, które nie spełniają tej specyfikacji. W odpowiedzi serwer wyśle datagram, na którego początku będzie znajdować się napis:

`DATA` *start długość*`\n`

Wartości *start* i *długość* są takie, jakich zażądał klient. Po tym napisie znajduje się *długość* bajtów pliku: od bajtu o numerze *start* do bajtu o numerze *start*+*długość*-1. Uwaga: bajty pliku numerowane są od zera.

### 1.1 Przykład

Jeśli program zostanie uruchomiony w następujący sposób:

```
$> ./transport 127.0.0.1 40001 output 1100
```

to może wysłać w do portu 40001 serwera uruchomionego pod adresem `127.0.0.1` dwa datagramy o treściach:

```
GET 0 700\n
```

oraz

```
GET 700 400\n
```

następnie zaczekać na odpowiedzi serwera i zapisać odpowiednie 1100 bajtów do pliku `output`.

## 1.2 Serwer UDP

Statycznie skonsolidowaną, 64-bitową wersję kodu serwera UDP o nazwie `transport-server` można pobrać ze strony wykładu i uruchomić na swoim komputerze podając jako argument port, na którym serwer ma nasłuchiwać. Można też uruchomić kod w maszynie wirtualnej. Dostępny na stronie wykładu obraz maszyny wirtualnej *Virbian* zawiera już ten program w katalogu `/usr/local/bin/`.

## 1.3 Dodatkowe wymagania

Pamiętaj, że pobierane dane są danymi binarnymi i mogą zawierać bajt równy zero. Należy je zatem zapisywać do pliku funkcją `write()`, `fwrite()` lub podobną. Twój program powinien zapisywać dane wyłącznie na końcu pliku wyjściowego. W szczególności niedopuszczalne jest uzyskiwanie dostępu do różnych miejsc pliku wyjściowego za pomocą funkcji `lseek()` lub podobnej. Twój program powinien zużywać co najwyżej 5 MB pamięci operacyjnej (łącznie z binarnym kodem programu; patrz część *Sposób oceniania programów*).

Program powinien obsługiwać błędne dane wejściowe, zgłaszając odpowiedni komunikat. Program nie powinien wypisywać niepotrzebnych komunikatów diagnostycznych, ale może wypisywać postęp w pobieraniu kolejnych części pliku.

## 1.4 Zawodna komunikacja

Należy pamiętać, że datagramy UDP mogą zostać zagubione, zduplikowane lub nadejść w innej kolejności niż były wysyłane. Wykorzystywany serwer UDP sprzyja nauczeniu się tych reguł, działając w następujący sposób:

1. Odpowiedź na dane żądanie zostaje wysłana z prawdopodobieństwem ok. 1/2.
2. Każda wysyłana odpowiedź zostanie zduplikowana z prawdopodobieństwem ok. 1/5. W tej definicji duplikat też jest traktowany jako odpowiedź, więc może pojawić się też duplikat duplikatu (z prawdopodobieństwem ok. 1/25) itd.
3. Każdy wysyłany datagram jest wysyłany z losowym opóźnieniem wynoszącym od 0,5 do 1,5 sekundy.
4. Serwer utrzymuje kolejkę co najwyżej 1000 datagramów, które ma wysłać.

## 1.5 Uwagi implementacyjne

Konieczne jest sprawdzanie, czy adres źródłowy i port źródłowy datagramu jest prawidłowy. Możesz założyć, że jeśli Twój program otrzymuje dane od adresu IP i portu, do którego dane uprzednio wysłał, to są one zgodne ze specyfikacją i nie trzeba tego sprawdzać.[1]

Twój program będzie testowany pod kątem poprawności i wydajności. W najprostszym wariancie można zaprogramować go jako algorytm typu *stop-and-wait*. Do odczekiwania i sprawdzania, czy gniazdo zawiera datagram, można wykorzystać funkcję `select()`. Tak zaimplementowany został program `transport-client-slow` (statycznie skonsolidowaną wersję 64-bitową można pobrać ze strony wykładu). Za poprawną implementację takiego podejścia można dostać 4 punkty.

[1] W prawdziwych zastosowaniach byłby to bardzo zły pomysł.

Podejście typu *stop-and-wait* jest bardzo nieefektywne. Aby je poprawić, możesz zaimplementować algorytm przesuwnego okna, utrzymujący odpowiednie liczniki czasu dla wszystkich otrzymanych segmentów. (Ograniczenie na pamięć ma na celu wymuszenie, żeby okno nie mieściło wszystkich datagramów: prefiks danych, który udało nam się pobrać należy zapisać do pliku). Taki algorytm wykorzystuje program `transport-client-fast` (również do pobrania ze strony wykładu). Za taką implementację można dostać maksymalną liczbę punktów.

Twój program nie musi pobierać danych zgodnie z opisanymi wyżej schematami. Program będzie uruchamiany na dwóch połączonych ze sobą maszynach wirtualnych. Parametry wpływające na jego efektywność (czas odczekiwania i rozmiar okna) można dobrać eksperymentalnie. Napisanie programu, który będzie dynamicznie dostosowywał się do parametrów łącza nie jest wymagane.

# 2 Uwagi techniczne

**Pliki** Sposób utworzenia napisu oznaczanego dalej jako *`imie_nazwisko`*: Swoje (pierwsze) imię oraz nazwisko proszę zapisać wyłącznie małymi literami zastępując litery ze znakami diakrytycznymi przez ich łacińskie odpowiedniki. Pomiędzy imię i nazwisko należy wstawić znak podkreślenia.

Prowadzącemu ćwiczeniopracownię należy dostarczyć plik *`imie_nazwisko`*`.tar.xz` z archiwum (w formacie `tar`, spakowane programem `xz`) zawierającym pojedynczy katalog o nazwie *`imie_nazwisko`* z następującymi plikami.

- Kod źródłowy w C lub C++, czyli pliki `*.c` i `*.h` lub pliki `*.cpp` i `*.h`. Każdy plik `*.c` i `*.cpp` na początku powinien zawierać w komentarzu imię, nazwisko i numer indeksu autora.
- Plik `Makefile` pozwalający na kompilację programu po uruchomieniu `make`.
- Ewentualnie plik `README.txt` lub `README.md`

W katalogu tym **nie** powinno być żadnych innych plików, w szczególności skompilowanego programu, obiektów `*.o`, czy plików źródłowych nie należących do projektu.

**Kompilacja** Kompilacja i uruchamianie przeprowadzone zostaną w 64-bitowym środowisku Linux. Kompilacja w przypadku C ma wykorzystywać standard C99 lub C17 z ewentualnymi rozszerzeniami GNU (opcja kompilatora `-std=c99`, `-std=gnu99`, `-std=c17` lub `-std=gnu17`), zaś w przypadku C++ — standard C++11, C++14 lub C++17 z ewentualnymi rozszerzeniami GNU (opcje kompilatora `-std=c++11`, `-std=gnu++11`, `-std=c++14`, `-std=gnu++14`, `-std=c++17` lub `-std=gnu++17`). Kompilacja powinna korzystać z opcji `-Wall` i `-Wextra`. Podczas kompilacji nie powinny pojawiać się ostrzeżenia.

# 3 Sposób oceniania programów

Poniższe uwagi służą ujednoliceniu oceniania w poszczególnych grupach. Napisane są jako polecenia dla prowadzących, ale studenci powinni **koniecznie się** z nimi zapoznać, gdyż będziemy się ściśle trzymać poniższych wytycznych. Programy będą testowane na zajęciach w obecności autora programu. Na początku program uruchamiany jest w różnych warunkach i otrzymuje za te uruchomienia od 0 do 10 punktów. Następnie obliczane są ewentualne punkty karne. Oceniamy z dokładnością do 0,5 punktu. Jeśli ostateczna liczba punktów wyjdzie ujemna wstawiamy zero. (Ostatnia uwaga nie dotyczy przypadków plagiatów lub niesamodzielnych programów).

**Testowanie: punkty dodatnie** Rozpocząć od kompilacji programu. W przypadku programu niekompilującego się stawiamy 0 punktów, nawet jeśli program będzie ładnie wyglądał.

Do testów uruchomić na dwie maszyny wirtualne połączone siecią na jednym komputerze. Na jednej z nich uruchomić instancję serwera `transport-server` (pobraną ze strony wykładu), słuchającą na wybranym porcie, a na drugiej maszynie uruchamiać kod studenta.

Przy uruchamianiu programu należy wykorzystywać polecenie `/usr/bin/time -v`. Umożliwi to pomiar czasu i zajętość pamięci (pola `Elapsed time` i `Maximum resident set size`).

**3 pkt.** Uruchomić program do pobrania ok. 15 000 bajtów, gdzie liczba bajtów nie jest wielokrotnością 1 000. Na takich samych danych uruchomić program `transport-client-slow`; niech $t$ będzie czasem jego działania. Program studenta otrzymuje punkty, jeśli jego czas działania jest nie większy niż $4 \cdot t + 5$ sek, zajętość pamięci nie większa niż 5 MB, a pliki generowane przez oba programy są identyczne.

**1 pkt.** Uruchomić program do pobrania ok. 15 000 bajtów. Zatrzymać go w trakcie wykonywania; sprawdzić Wiresharkiem jaki jest jego port źródłowy. Następnie poleceniem `nc` wysłać do tego portu źródłowego datagram zawierający śmieci. Wznowić działanie programu i sprawdzić, czy generowany jest poprawny plik.

**3 pkt.** Jak w pierwszym punkcie, ale pobieramy ok. 1 000 000 bajtów i porównujemy czas z czasem działania programu `transport-client-fast`; niech $t$ będzie czasem jego działania. Czas działania programu studenta nie powinien być większy niż $4 \cdot t + 5$ sek.

**3 pkt.** Jak w poprzednim punkcie, ale pobieramy ok. 9 000 000 bajtów.

**Punkty karne** Punkty karne przewidziane są za następujące usterki.

**-2 pkt.** Za każdy rozpoczęty megabajt ponad limit 5 MB na zajętość pamięciową programu.

**-5 pkt.** Zapis do pliku wyjściowego w innym miejscu niż na jego końcu.

**-1 pkt.** Brak sprawdzania poprawności danych na wejściu.

**do -3 pkt.** Zła / nieczytelna struktura programu: brak modularności i podziału na funkcjonalne części, niekonsekwentne wcięcia, powtórzenia kodu.

**-2 pkt.** Aktywne czekanie zamiast zasypiania do momentu otrzymania pakietu.

**-1 pkt.** Brak sprawdzania poprawności wywołania funkcji systemowych, takich jak `recvfrom()`, `write()` czy `bind()`.

**-1 pkt.** Nietrzymanie się specyfikacji wejścia i wyjścia. Przykładowo: wyświetlanie nadmiarowych informacji diagnostycznych, inna niż w specyfikacji obsługa parametrów.

**-1 pkt.** Zły plik `Makefile` lub jego brak: program powinien się kompilować poleceniem `make`: poszczególne pliki `*.c` i `*.cpp` powinny kompilować się do obiektów tymczasowych `*.o` a następnie powinny być konsolidowane do wykonywalnego programu. Polecenie `make clean` powinno czyścić katalog z tymczasowych obiektów (plików `*.o`), zaś polecenie `make distclean` powinno usuwać te obiekty i wykonywalny program pozostawiając tylko pliki źródłowe.

**-3 pkt.** Kara za wysłanie programu po terminie; opóźnienie nie może być większe niż 1 tydzień.

*Marcin Bieńkowski*