Jagiełło Maciej, Kobuszewski Michał, Kucharski Maciej, Szlachetka Arkadiusz

# Techniki internetowe - projekt wstępny

1. Temat: Oprogramowanie prostego sieciowego serwera plików.

Założenia: program realizowany jest w technologii klient-serwer z wykorzystaniem protokołu TCP. Serwer udostępnia zestaw podstawowych operacji na plikach. Klient odwołuje się do serwera poprzez bibliotekę funkcji “ukrywającej” przed programistą aplikacji szczegóły komunikacji. Oprogramowanie klienta powinno stanowić zestaw testów.

2. Interpretacja tematu:

Z wykorzystaniem protokołu TCP napisać serwer na którym będzie możliwosć tworzenia i przechowywania plików przez użytkowników wykorzystujących aplikację kliencką (klienta) która będzie udostępniała takie funkcje jak właśnie tworzenie i modyfikowanie.

Przetrzeń na serwerze będzie wspólna a dostęp do niej będzie synchronizowany przy użyciu funkcji fs_lock (niżej opisanej dokładniej).

W celu komunikacji klienta z serwerem będzie tworzona biblioteka obudowywująca podstawowe funkcje serwera, co zapewnioni modularność rozwiązania. W celu sprawdzenia funkcjonalności projektu, po stronie klienta będzie zdefiniowany zestaw testów, który pokaże wszystkie braki i niedociągnięcia stworzonego rozwiązania.

Sprecyzowanie działania poszczególnych funkcji udostępnianych przez bibliotekę:

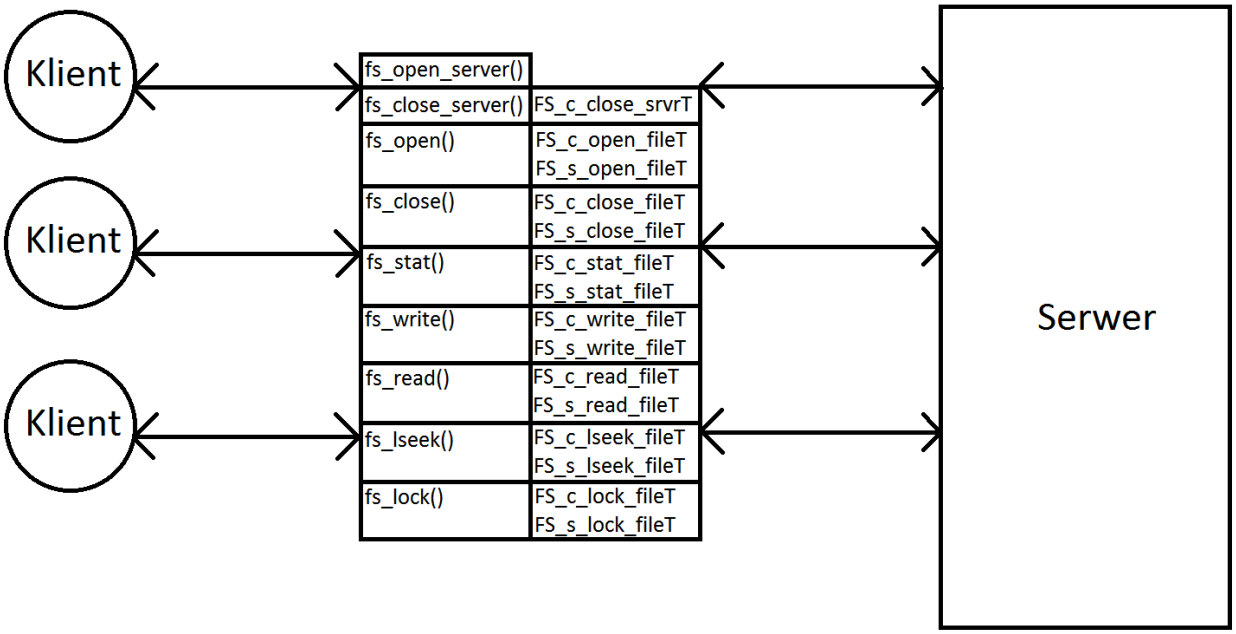
**int fs_open_server(char *adres_serwera)** - tworzy połączenie z serwerem, zwraca uchwyt serwera, który jest jednocześnie desktryptorem gniazda utworzonego do komunikacji, w przypadku niepowodzenia zwróci pewną wartość ujemną, zależną od typu zastniałego błędu.
**int fs_close_server(int srvhndl)** - zamyka połączenie z serwerem, w wypadku sukcesu zwraca 0, w przypadku błędu jak wyżej.
**int fs_open(int srvhndl, char *name, int flags)** - otwarcie konkretnego pliku - ścieżka do pliku podawana przez klienta będzie podlegać translacji na serwerze, które będzie polegało na przekształceniu na ścieżkę bezwględną. Z uwagi na brak uwierzytelniania podłączanych klientów zakłada się brak kontroli praw dostępu, zatem każdy klient będzie w stanie otworzyć każdy plik z dowolną flagą. W odniesieniu do sposobu przekazywania deskryptorów plików, będą przekazywane unikalne wartości nadawne przez aplikacje serwera i będą podlegać mapowaniu na lokalną wartość wykorzystywaną przez danego klienta, który będzie osobnym procesem.
**int fs_write(int srvhndl , int fd , void * buf , size_t len)** - zapisanie do pliku o deskrytporze *fd* danych z bufora. Aby funkcja zadziałała poprawnie, na pliku musi być założona dokładnie jedna blokada, i to do zapisu, należąca do klienta wykonującego operację.
**int fs_read(int srvhndl , int fd , void * buf , size_t len)** - wczytanie z pliku o deskryptorze *fd* danych do bufora. Aby funkcja zadziałała poprawnie, na pliku musi występować blokada do czytania lub pisania, należąca do klienta wykonującego operację.
**int fs_lseek(int srvhndl, int fd , long offset , int whence)** - przesuwa wskaźnik pliku o

zadaną wartość w określonym kierunku.
**int fs_close(int srvhndl, int fd)** - zamknięcie danego pliku. Zamknięcie pliku oznacza również zdjęcie wszystkich blokad założonych przez danego klienta.
**int fs_stat(int srvhndl , int fd, struct stat* buff)** - pobranie informacji o pliku. Nie wszystkie pola struktury mają sens, zatem poniżej są wymienione tylko te używane przez nas:

```
mode_t    st_mode;    /* protection */
off_t     st_size;    /* total size, in bytes */
blksize_t st_blksize; /* blocksize for file system I/O */
blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
time_t    st_atime;   /* time of last access */
time_t    st_mtime;   /* time of last modification */
time_t    st_ctime;   /* time of last status change */
```

**int fs_lock(int srvhndl , int fd , int mode)** - założenie blokady na pliku. W przypadku powodzenia zwróci 0. Niepowodzenie wystąpi w następujących sytuacjach:
-próba założenia blokady READ jeśli jest założona blokada WRITE,
-próba założenia blokady WRITE jeśli jest już założona inna blokada;
i będzie sygnalizowane poprzez zwrócenie ujemnej wartości. Informacja o założonych blokadach będzie przechowywana w strukturze opisującej dany plik.

3.Krótki opis funkcjonalny
Poglądowy rysunek komunikacji pomiędzy modułami:




Operacje udostępniane klientowi to: połączenie z serwerem, operacje na plikach: tworzenie i modyfikacja, zamknięcie połączenia z serwerem, podejrzenie statystyk pliku. Połączenie będzie tworzone na czas całej sesji. Serwer, poza udostępnieniem klientom powyższych operacji, rozwiązuje kwestie związane ze współbieżnym dostępem do plików. Dane pomiędzy klientem a serwerem (przy operacjach odczytu i zapisu) będą przesyłane w

4KB blokach.

4. Opis i analiza poprawności stosowanych protokołów komunikacyjnych

**int fs_open_server(char *adres_serwera)**

Od klienta będzie wysyłany jedynie adres serwera, natomiast serwer nie przesyła żadnych dodatkowych danych.

Obsługa błędów będzie miała miejsce jedynie po stronie klienta. Lista błędów jest zbieżna z tymi generowanymi przez funkcje *socket()* oraz *connect()*.

**int fs_close_server(int srvhndl)**

Od klienta jest przesyłana struktura

```
struct FS_c_close_srvrT {
      FS_cmdT command;
}
```

Obsługa błędów podobnie jak przy *fs_open_server().*

**int fs_open(int srvhndl, char *name, int flags)**

Klient

```
struct FS_c_open_fileT
{
      FS_cmdT command;
      char* name;
      int flags;
}
```

Serwer

```
struct FS_s_open_fileT {
      FS_resT command; //na jaką komende
jest to odpowiedź
      int fd; //deskryptor pliku lub wartość
ujemna w przypadku niepowodzenia
}
```

Możliwe wartości *fd*:
*fd* > 0 - sukces, jest to deskryptor pliku
*fd* = -200999 - żądany plik nie istnieje
*fd* = -200998 - na pliku nie jest założona odpowiednia blokada
Pozostałe błędy będą sygnalizowane zwróceniem ujemnej wartości kodu odpowiedniej stałej z *errno*.

**int fs_close(int srvhndl, int fd)**

Klient

```
struct
FS_c_close_fileT {
      FS_cmdT command;
      int fd;
}
```

Serwer

```
struct FS_s_close_fileT {
      FS_resT command; //na jaką komende
jest to odpowiedź
      FS_statT status;
}
```

Możliwe wartości pola status:

0 w przypadku powodzenia. W przypadku niepowodzenia ujemna wartość kodu odpowiedniej stałej z *errno* (odpowiadającej wartości ustawianej przez “zwykłą” funkcję *close()*).

**int fs_stat(int srvhndl , int fd, struct stat* buff)**

Klient

```
struct FS_c_stat_fileT {
      FS_cmdT command;
      int fd;
}
```

Serwer

```
struct FS_s_stat_fileT {
      FS_resT command;
      FS_statT status;
      struct stat buf; //bufor na dane o
pliku
}
```

Możliwe wartości pola status:
0 w przypadku powodzenia. W przypadku niepowodzenia ujemna wartość kodu odpowiedniej stałej z *errno* (odpowiadającej wartości ustawianej przez “zwykłą” funkcję *close()*).

**int fs_write(int srvhndl , int fd , void * buf , size_t len)**

Klient

```
struct FS_c_write_fileT
{
      FS_cmdT command;
      int fd;
      size_t len; //wielkość
danych do zapisania
      void* data; //dane do
zapisania w pliku
}
```

Serwer

```
struct FS_s_write_fileT
{
      FS_resT command;
      FS_statT status;
      size_t written_len; //ilość
faktycznie zapisanych danych
}
```

Pole *written_len* jest istotne w sytuacjach, gdy klient spróbuje wysłać jednorazowo zbyt dużą porcję danych.

Możliwe błędy w logice (zawarte w polu *status*):
-200999 - nie ma takiego pliku
-200998 - na pliku nie jest założona odpowiednia blokada
Pozostałe błędy będą sygnalizowane zwróceniem ujemnej wartości kodu odpowiedniej stałej z *errno*.

**int fs_read(int srvhndl , int fd , void * buf , size_t len)**

Klient

```
struct
FS_c_read_fileT {
      FS_cmdT command;
      int fd;
      size_t len;
}
```

Serwer

```
struct FS_s_read_fileT {
      FS_resT command;
      FS_statT status;
      size_t read_len; //ilość faktycznie
odczytanych danych
      void * data; //bufor na dane do odczytu
}
```

Pole *read_len* jest istotne w sytuacjach, gdy klient spróbuje odczytać jednorazowo zbyt dużą porcję danych.
Możliwe błędy w logice (zawarte w polu *status*):
-200999 - nie ma takiego pliku
-200998 - na pliku nie jest założona odpowiednia blokada
Pozostałe błędy będą sygnalizowane zwróceniem ujemnej wartości kodu odpowiedniej stałej z *errno*.

**int fs_lseek(int srvhndl, int fd , long offset , int whence)**

Klient

```
struct FS_c_lseek_fileT {
      FS_cmdT command;
      int fd;
      long offset;
      int whence;
}
```

Serwer

```
struct FS_s_lseek_fileT {
      FS_resT command;
      FS_statT status;
}
```

Sygnalizacja błędów taka sama jak w funkcjach *fs_read()* i *fs_write()*.

**int fs_lock(int srvhndl , int fd , int mode)**

Klient

```
struct FS_c_lock_fileT {
      FS_cmdT command;
      int fd;
      int lock_type; //alias dla
mode
}
```

Serwer

```
struct FS_s_lock_fileT {
      FS_res command;
      FS_statT status;
}
```

Możliwe wartości pola *status*:
-200999 - nie ma takiego pliku.
-200997 - brak możliwości założenia blokady ze względu na obecność innych blokad.

enum FS_cmdT{ close_srv , file_open, file_close , file_read, file_write, file_stat, file_lock, file_lseek}
enum FS_resT{ close_srv , file_open, file_close , file_read, file_write, file_stat, file_lock, file_lseek}
typedef FS_statT int;

## 5. Planowany podział na moduły

Projekt jest zasadniczo podzielony na trzy części:
1. prosty program klienta, który wykorzystuje funkcje zaimplementowane w bibliotece klienckiej,
2. biblioteka kliencka, która jest odpowiedzalna za stworzenie i utrzymanie połączenia pomiędzy klientem a serwerem oraz udostępnia wcześniej wyspecyfikowane funkcje,
3. program serwera, który odpowiedzialny jest za wykonanie wszystkich operacji zleconych przez bibliotekę kliencką.

Dla programu klienta nie przewiduje się podziału na moduły, a cała jego implementacja będzie się mieścić w pliku *fs_client.cpp*.

Biblioteka kliencka będzie zbiorem wszystkich funkcji udostępnionych klientowi do wykonywania na serwerze, a implementacja mieścić się będzie w plikach *fs_libclient.h* oraz *fs_libclient.cpp* natomiast dla aplikacji klienta będzie widoczy jako *fs_libclient.o*.

Serwer będzie składał się z dwóch podmodułów: sieciowego (*fs_server_web.h* i *fs_server_web.cpp*) oraz podmodułu operacji na plikach (*fs_server_file.h* i *fs_server_file.cpp*). Podmoduł sieciowy będzie odpowiedzialny za nasłuchiwanie podłączenia nowych klientów, tworzeniem nowych procesów dla tych klientów oraz obsługą pakietów przesyłanych do i z biblioteki klienckiej. Będzie on wywoływał odpowiednie funkcję podmodułu obsługi plików na serwerze.

Modułem łączącym aplikacje serwera i klienta jest plik *fs_server.h* który przechowuje definicję wszystkich struktur używanych w komunikacji sieciowej.

## 6. Zarys koncepcji implementacji

Język programowania: C++, kompilator GNU g++ 4.8.2, zestaw bibliotek boost 1.55.0, narzędzie do budowania Scons, narzędzie do debugowania gdb.