TIN - projekt wstępny

Małgorzata Górecka
Maciej Kapuściński
Weronika Paszko
Sebastian Pietras

03.04.2020

1 Zadanie

Napisać program obsługujący uproszczoną wersję protokołu NFS (Network File System). Zgodnie z wariantem zadania komunikacja powinna bazować na UDP. Celem projektu jest zaimplementowanie serwera, biblioteki klienckiej oraz testowego programu klienckiego.

Biblioteka powinna udostępniać następujące odpowiedniki funkcji systemowych:

- `mynfs_open`
- `mynfs_read`
- `mynfs_write`
- `mynfs_lseek`
- `mynfs_close`
- `mynfs_unlink`

2 Doprecyzowanie treści

- Serwer otrzymuje od klientów żądania wykonania operacji na plikach, które próbuje spełnić i odsyła rezultaty w jednym datagramie
- Serwer wykonuje jedno żądanie naraz, jednak komunikacja odbywa się współbieżnie
- Nasłuchiwanie żądań odbywa się na jednym porcie, dalsza komunikacja z klientami (po odebraniu żądania) jest przenoszona przez serwer na inny, dostępny port. Pozwala to na niezmienność portu do nasłuchiwania nadchodzących żądań
- W przypadku zaginięcia potwierdzenia przyjęcia żądania (wysyłane na nowym porcie), klient powtarza żądanie (na stary port), a nowy port po pewnym czasie staje się nieaktywny i może zostać użyty ponownie
- Do obsługi pliku klienci posługują się lokalnym identyfikatorem, zwracanym przez serwer przy otwarciu pliku. Serwer po swojej stronie będzie mapował identyfikatory na faktyczne deskryptory plików, używając struktury mapującej
- Maksymalna ilość danych do zapisu/odczytu to 4 kB
- Klient komunikuje się z serwerem wyłącznie za pomocą funkcji bibliotecznych
- Serwer wykonuje operacje na swoich plikach za pomocą funkcji systemowych
- Do uwierzytelniania posłuży lista zawierająca dopuszczalne adresy IP wraz z ich prawami dostępu, jednakowymi dla wszystkich plików serwera. Z tego powodu w funkcji `mynfs_open()` nie ma argumentu `mode`
- W razie zaginięcia datagramu istnieje mechanizm retransmisji pakietu
- Dopóki istnieją otwarte deskryptory plików (na przykład używane przez innego klienta) fizyczne usunięcie plików nie jest możliwe

3 Opis funkcjonalny

Po pozytywnym uwierzytelnieniu klient dostaje dostęp do usług zgodnie z przypisanymi prawami dostępu. Użytkownik może komunikować się z serwerem za pomocą tekstowych komend wpisywanych w terminalu, które pozwalają na wykonywanie zdalnych operacji na plikach sieciowych.

Klient ma możliwość:

- tworzyć nowe pliki, o ile na serwerze nie znajdują się pliki o tej samej nazwie
- działać na już istniejących plikach:
  - odczytywać
  - edytować
  - usuwać

4 Biblioteka kliencka

Biblioteka kliencka udostępnia następujące funkcje:

```
int16_t mynfs_open(char *host, char *path, uint8_t oflag);
```

Otwiera zdalny plik. Zwraca jego deskryptor.

```
int16_t mynfs_read(char *host, int16_t fd, void *buf,
                   int16_t count);
```

Odczytuje podaną ilość bajtów ze zdalnego pliku i zapisuje w podanym buforze. Zwraca rozmiar odczytanych danych.

```
int16_t mynfs_write(char *host, int16_t fd, void *buf,
                    int16_t count);
```

Zapisuje do zdalnego pliku podaną ilość bajtów z podanego bufora. Zwraca rozmiar zapisanych danych.

```
int32_t mynfs_lseek(char *host, int16_t fd, int32_t offset,
                    uint8_t whence);
```

Zmienia bieżącą pozycję w zdalnym pliku. Zwraca ustawioną pozycję liczoną od początku pliku.

```
int8_t mynfs_close(char *host, int16_t fd);
```

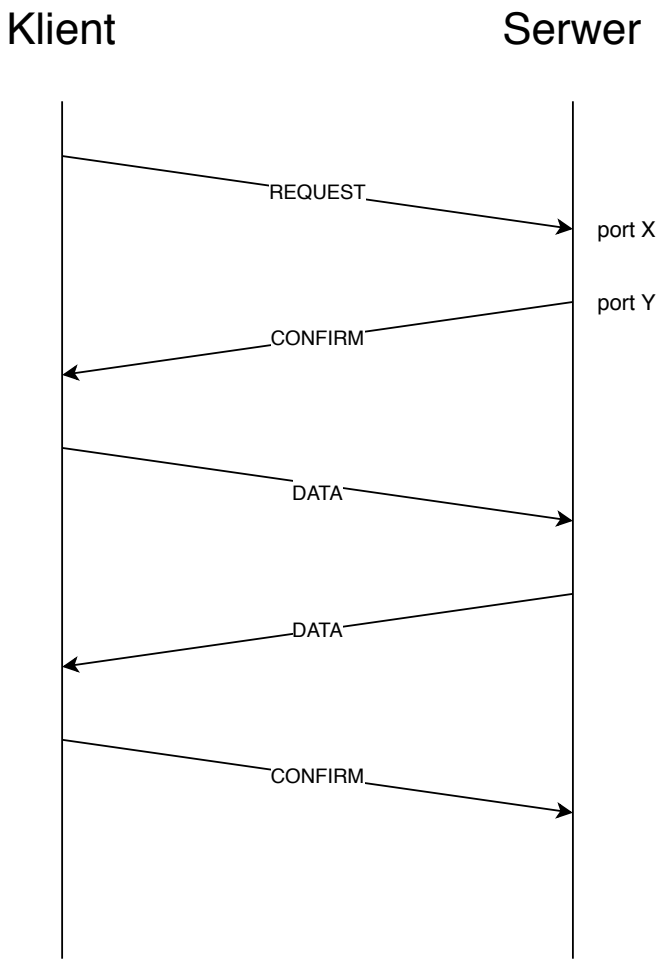
Zamyka zdalny plik. Zwraca 0 przy sukcesie, -1 przy błędzie.

```
int8_t mynfs_unlink(char *host, char *path);
```

Oznacza plik jako "do usunięcia". System usuwa plik dopiero wtedy, kiedy nie ma deskryptorów z nim powiązanych. Zwraca 0 przy sukcesie, -1 przy błędzie.

5 Protokół

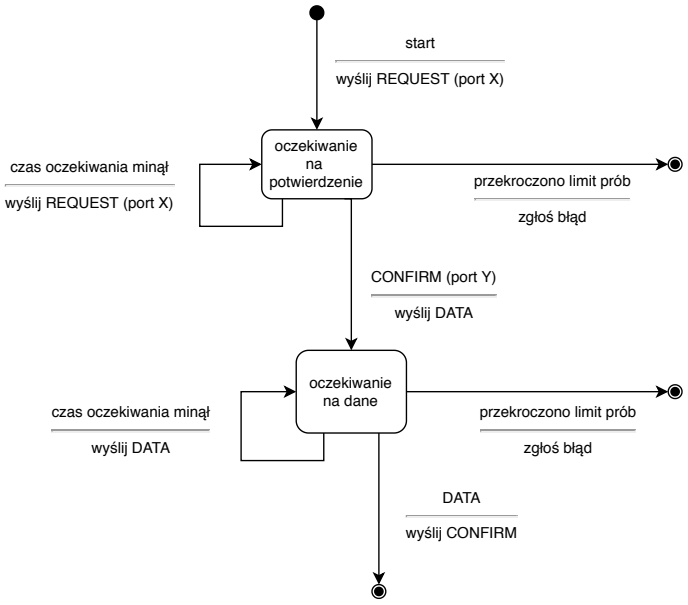
Rysunek 1: Zależności czasowe przy wymianie komunikatów



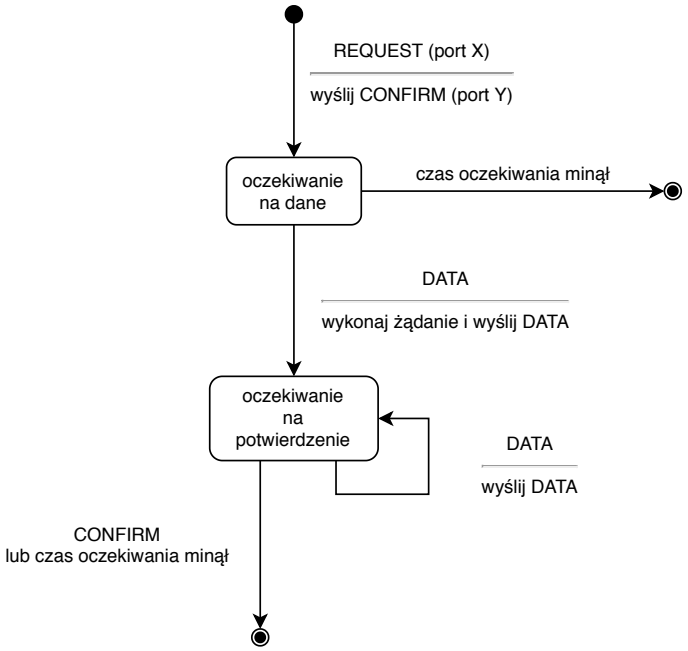
Opisy przy przejściach na rysunkach 2 i 3 należy interpretować w następujący sposób:

- informacja nad kreską mówi o tym, co wyzwala przejście
- informacja pod kreską mówi o tym, jaka akcja jest podejmowana w związku z przejściem

Rysunek 2: Diagram stanów komunikacji klienta



Rysunek 3: Diagram stanów komunikacji serwera



Składnia wiadomości:

```
MSG          = REQUEST_MSG | CONFIRM_MSG | DATA_MSG

REQUEST_MSG  = <1 bajt: MSG_TYPE=REQUEST>

CONFIRM_MSG  = <1 bajt: MSG_TYPE=CONFIRM> <1 bajt: ERROR>

DATA_MSG     = <1 bajt: MSG_TYPE=DATA> <1 bajt: ERROR> DATA

DATA         = REQUEST_DATA | REPLY_DATA

REQUEST_DATA = <1 bajt: FUN_ID=OPEN>   open_request_data |
               <1 bajt: FUN_ID=READ>   read_request_data |
               <1 bajt: FUN_ID=WRITE>  write_request_data |
               <1 bajt: FUN_ID=LSEEK>  lseek_request_data |
               <1 bajt: FUN_ID=CLOSE>  close_request_data |
               <1 bajt: FUN_ID=UNLINK> unlink_request_data
REPLY_DATA  =  <1 bajt: FUN_ID=OPEN>   open_reply_data |
               <1 bajt: FUN_ID=READ>   read_reply_data |
               <1 bajt: FUN_ID=WRITE>  write_reply_data |
               <1 bajt: FUN_ID=LSEEK>  lseek_reply_data |
               <1 bajt: FUN_ID=CLOSE>  close_reply_data |
               <1 bajt: FUN_ID=UNLINK> unlink_reply_data

struct open_request_data {
        uint8_t oflag; // flagi do open
        int16_t path_count; // długość ścieżki
        char* path; // ścieżka, maks. 4096 bajtów
}

struct read_request_data {
        int16_t fd; // deskryptor pliku
        int16_t count; // liczba bajtów do odczytania
}

struct write_request_data {
        int16_t fd; // deskryptor pliku
        int16_t count; // liczba bajtów do zapisania
        void* buf; // dane do zapisania, maks. 4096 bajtów
}
```

```
struct lseek_request_data {
        int16_t fd; // deskryptor pliku
        int32_t offset; // pozycja
        uint8_t whence; // tryb - odkąd liczyć pozycję
}

struct close_request_data {
        int16_t fd; // deskryptor pliku
}

struct unlink_request_data {
        int16_t path_count;  // długość ścieżki
        char* path; // ścieżka, maks. 4096 bajtów
}

struct open_reply_data {
        int16_t fd; // deskryptor pliku
}

struct read_reply_data {
        int16_t count; // liczba odczytanych bajtów
        void* buf; // odczytane dane, maks. 4096 bajtów
}

struct write_reply_data {
        int16_t count; // liczba zapisanych bajtów
}

struct lseek_reply_data {
        int32_t offset; // ustawiona pozycja
}

struct close_reply_data {
        int8_t success; // dodatnie - sukces, ujemne - błąd
}

struct unlink_reply_data {
        int8_t success;  // dodatnie - sukces, ujemne - błąd
}
```

Jeśli po stronie serwera wystąpi błąd to kod błędu będzie przekazany w nagłówku odpowiedzi. Po odebraniu komunikatu z błędem biblioteka kliencka będzie zapisywać kody błędów w globalnej zmiennej.

6 Organizacja

Nasz projekt będzie składał się z trzech części: programu serwera, biblioteki klienckiej i programu klienckiego do przetestowania działania biblioteki klienckiej.

Program serwera będzie podzielony na trzy różne moduły: moduł główny, moduł obsługi żądania oraz moduł obsługi plików.

- Główny moduł będzie odbierał komunikaty REQUEST od klientów. Dla każdego poprawnie odebranego komunikatu będzie tworzył nowy wątek, mający za zadanie zrealizować żądanie klienta.
- Moduł obsługi żądań będzie tworzył gniazda używające innych portów niż główny i kontynuował komunikację z klientem za pomocą tych gniazd. Będzie przekazywał polecenia klienta do modułu obsługi plików i wysyłał klientowi efekty działań.
- Moduł obsługi plików będzie dostawał od modułu obsługi żądań polecenia do wykonania na plikach. Następnie będzie je wykonywał i przekazywał efekty z powrotem, aby zostały wysłane do klientów.

Program kliencki będzie składał się z modułu interakcji z użytkownikiem i modułu komunikacji z serwerem.

7 Implementacja

Projekt będzie wykonany w języku C++ z użyciem gniazd BSD.