TIN - dokumentacja końcowa

Małgorzata Górecka
Maciej Kapuściński
Weronika Paszko
Sebastian Pietras

02.06.2020

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);
```

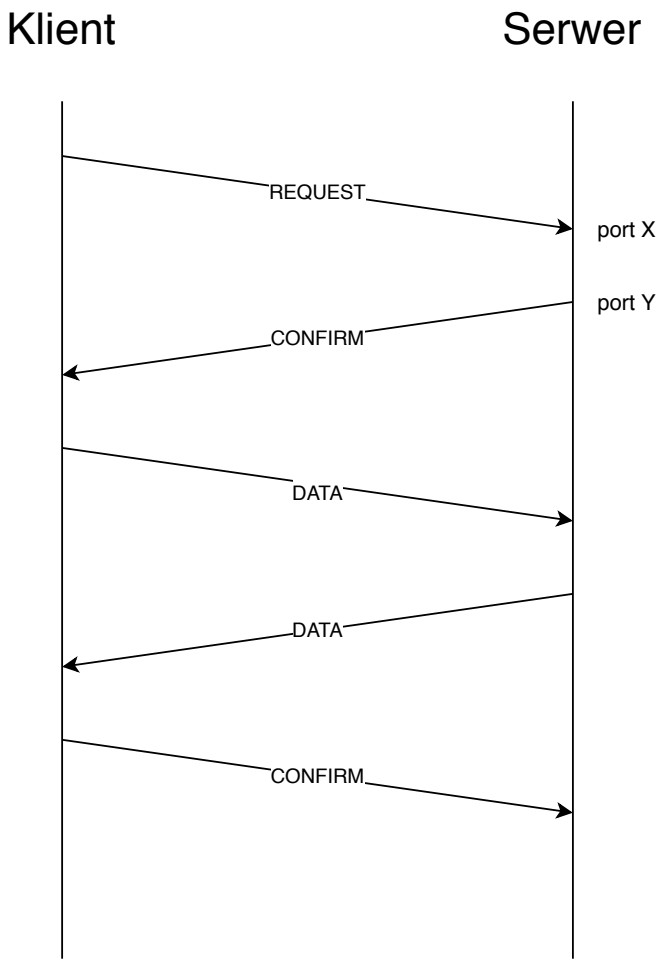
Zamyka zdalny plik. Zwraca 0 przy sukcesie, -1 przy błędzie.

```
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

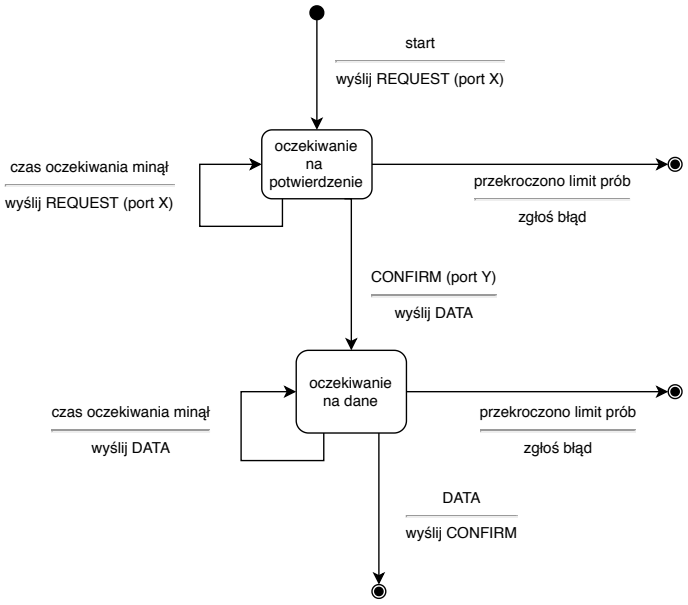
Rysunek 1: Zależności czasowe przy wymianie komunikatów



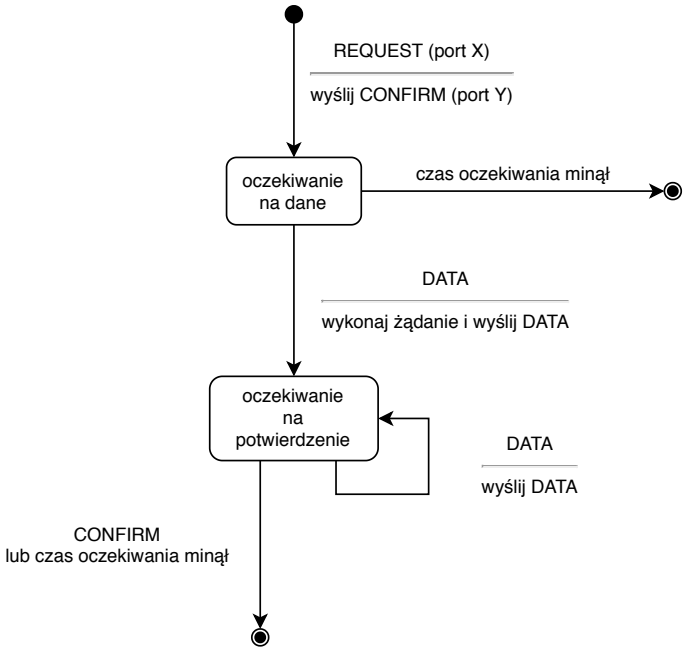
Opisy przy przejściach na rysunkach 2 i 3 należy interpretować w następujący sposób:

- 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 Interfejs użytkownika

Użytkownik może komunikować się z serwerem za pomocą tekstowych komend wpisywanych w terminalu, które pozwalają na wykonywanie zdalnych operacji na plikach sieciowych. Po uruchomieniu programu użytkownik podaje adres i port serwera, z którym chce się połączyć. Następnie zostaną wyświetlone polecenia, z których może skorzystać.

Dostępne polecenia:

- Open - otwieranie pliku na serwerze. Plik może zostać otwarty w trzech trybach: tylko do odczytu, tylko do zapisu oraz do odczytu i zapisu jednocześnie. Na początku użytkownik podaje ścieżkę do pliku. Użytkownik może podać flagi otwarcia pliku. Dostępne flagi zostaną wyświetlone użytkownikowi. Po udanym wykonaniu się funkcji wyświetlony zostanie komunikat o powodzeniu operacji oraz deskryptor powiązany z otwartym plikiem.
- Read - odczyt danych z pliku. Użytkownik podaje liczbę bajtów do odczytania. Po udanej operacji otrzymuje komunikat, ile bajtów z pliku udało się odczytać oraz treść, która została odczytana. Serwer nie zawsze jest w stanie przeczytać tyle bajtów, ile oczekuje użytkownik.
- Write - zapis danych do pliku. Użytkownik wprowadza dane tekstowe do zapisania. Po wysłaniu danych dostanie informację, ile bajtów danych zostało zapisanych.
- Lseek - zmiana pozycji w pliku do odczytu/zapisu. Klient wybiera nową pozycję oraz odkąd jest liczona. Po pozytywnym zakończeniu operacji przekazywana jest nowa pozycja w pliku liczona od zera.
- Close - zamknięcie deskryptora pliku. Klient wybiera, który deskryptor ma zostać zamknięty. Po udanej operacji dany deskryptor jest niedostępny dla użytkownika.
- Unlink - usunięcie nazwy z systemu plików. Użytkownik wybiera plik, którego nazwa ma zostać usunięta z systemu plików. Plik zostanie fizycznie usunięty po zamknięciu ostatniego otwartego deskryptora.
- Exit - zamknięcie programu.

Podczas operacji wymagających deskryptora użytkownik będzie mógł wybrać go z listy dostępnych.

Po każdym udanym wykonaniu się funkcji na plikach wyświetlany jest komunikat o powodzeniu operacji. Jeśli jednak z dowolnych przyczyn operacja nie mogła być wykonana wyświetlony zostanie komunikat z kodem i treścią błędu.

7 Implementacja

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

Podczas pracy wykorzystywaliśmy:

- system kontroli wersji – git
- zintegrowane środowisko programistyczne języków C/C++ - CLion
- narzędzie do automatycznego zarządzania procesem kompilacji – CMake
- biblioteki Catch i FFF użyte do testów jednostkowych
- bibliotekę thread do zaimplementowania wielowątkowości
- symulacja bezpośrednich połączeń w Internecie - Hamachi

7.1 Rozwiązania funkcjonalne

Serwer nie udostępnia klientowi faktycznych deskryptorów systemowych. Zamiast tego przesyła generowane deskryptory, które po stronie serwera są mapowane na deskryptory systemowe. Po zamknięciu pliku deskryptor trafia do listy deskryptorów do ponownego użycia. Podczas generacji nowego deskryptora serwer pobiera najmniejszy deskryptor z listy lub, jeśli lista jest pusta, tworzy nowy deskryptor o najmniejszym numerze, który nie był do tej pory użyty.

W strukturze mapującej deskryptory aplikacyjne na systemowe znajduje się również adres IP klienta, który otworzył dany deskryptor. Dzięki temu można sprawdzać czy deskryptor, który podał klient, należy do niego.

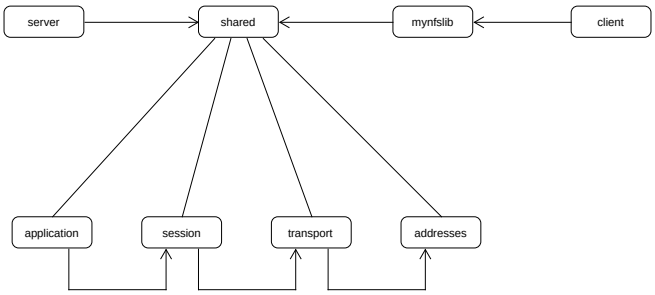
W celu zabezpieczenia maszyny serwera wszystkie pliki klientów znajdują się w specjalnie wydzielonej przestrzeni katalogowej. Klient nie może dostać się do plików spoza tej przestrzeni.

Serwer podczas uruchomienia odczytuje plik `hosts.txt`, w którym znajdują się adresy IP klientów uprawnionych do korzystania z usług serwera wraz z ich prawami dostępu (np. `192.168.0.4 RW`).

7.2 Podział na moduły i komunikacja między nimi

Projekt składa się następujących modułów: serwera (server), aplikacji klienckiej (client), biblioteki klienckiej (mynfslib) i bibliotek współdzielonych - addresses, application, session oraz transport.

Rysunek 4: Struktura zależności modułów



Z bibliotek współdzielonych korzystają serwer oraz biblioteka kliencka. Natomiast aplikacja kliencka korzysta z biblioteki klienckiej. Biblioteka application korzysta z session, session korzysta z transport, a transport z addresses. Wszystkie biblioteki współdzielone znajdują się w katalogu `shared`.

Krótki opis modułów:

- Addresses - wrappery na adresy (adres IP i port), umożliwia wywołanie resolvera DNS do uzyskania adresu IP
- Transport - odpowiada za obsługę gniazd
- Session - serializacja i deserializacja komunikatów
- Application - klasy reprezentujące żądania i odpowiedzi
- mynfslib - biblioteka kliencka służąca do komunikacji z serwerem
- Client - aplikacja kliencka umożliwiająca użytkownikowi wykonywanie operacji na plikach na serwerze za pomocą poleceń tekstowych
- Server - aplikacja serwera przetwarzająca żądania klientów

7.3 Organizacja serwera

Serwer jest podzielony na kilka różnych modułów pracujących w osobnych wątkach, z których każdy znajduje się w osobnej klasie:

- Główny moduł ServerEndpoint tworzy wszystkie pozostałe obiekty i wątki. Po utworzeniu wątków Executora oraz TerminalListenera, moduł wchodzi w pętlę, którą powtarza do momentu odebrania sygnału do zakończenia pracy od TerminalListenera. W pętli odbiera REQUEST od klientów i dla każdego poprawnie odebranego żądania tworzy i odłącza nowy wątek ServerSubEndpoint, mający za zadanie wypełnić to żądanie. Po sygnale do zakończenia pracy, główny moduł przestaje odbierać nowe żądania, czeka aż zakończy się wykonanie dotychczasowych żądań, sygnalizuje zakończenie do Executora i dołącza pozostałe wątki.
- Moduł obsługi żądania, w klasie ServerSubEndpoint, dostaje adres klienta od ServerEndpoint i komunikuje się z klientem używając osobnego gniazda. Po poprawnym odebraniu od klienta polecenia, tworzy odpowiedni Handler do wykonania go, a następnie przekazuje ten Handler do Executora i czeka na zakończenie obsługi. Gdy Executor zasygnalizuje obsłużenie żądania, moduł wysyła klientowi efekt polecenia. Po otrzymaniu potwierdzenia ServerSubEndpoint kończy swoją pracę.
- Moduł obsługi plików, w klasie Executor, pobiera najstarszy Handler z współdzielonej kolejki, obsługuje go, a następnie sygnalizuje zakończenie wykonania wątkowi obsługującemu żądanie. Powtarza to dopóki główny moduł nie zasygnalizuje zakończenia działania.
- Moduł sygnalizacji zakończenia, w klasie TerminalListener, oczekuje na sygnał do zakończenia z wejścia standardowego, a po otrzymaniu go sygnalizuje polecenie zakończenia modułowi głównemu.

8 Budowanie

Plik `build.sh` umożliwia zbudowanie aplikacji klienckiej, biblioteki klienckiej i serwera poprzez wywołanie:

```
./build.sh { client, mynfslib, server }
```

Współdzielone biblioteki zostaną dołączone automatycznie. Do budowania jest użyty CMake. Wykorzystuje on `CMakeLists.txt`, w których przechowywana jest informacja, z jakich plików składa się moduł i jakich bibliotek potrzebuje.

9 Opis sytuacji wyjątkowych i reakcji na błędy

Biblioteka kliencka wyłapuje wyjątki, które mogą wystąpić podczas komunikacji z serwerem i na ich podstawie ustala typy błędów. Wyróżniamy następujące rodzaje błędów:

- Address error (1xxx) - nieprawidłowe adresy
- Bad argument error (2xxx) - niepoprawne parametry żądań
- Network error (3xxx) - wadliwe połączenie
- Reply error (4xxx) - niemożność wykonania żądania przez serwer
- Authentication error (5xxx) - brak praw dostępu

W zależności od sytuacji obsługa żądania po stronie serwera może:

- być od razu przerwana (np. pierwsza wiadomość od klienta ma nieodpowiedni typ),
- spowodować wysłanie informacji o błędzie od serwera do klienta (np. klient nie jest uprawniony lub wysłał request nieznanego typu),
- powtórzyć wysyłanie komunikatu (np. czas oczekiwania na odpowiedź minął).

Maksymalnie wykonuje się 5 prób retransmisji. Jeśli po tej liczbie prób klient wciąż nie uzyska odpowiedzi, to u klienta w terminalu wyświetli się „Host unreachable".

Dokładne zachowanie klienta i serwera podczas komunikacji opisane jest diagramami stanów z rozdziale **Protokół**.

10 Testowanie

Do każdego z modułów zaimplementowane zostały testy jednostkowe. Są to testy sprawdzające pojedyncze funkcjonalności i reakcje systemu na pojawiające się błędy i wyjątki.

Testy komunikacji pomiędzy serwerem a klientem zostały przeprowadzone w dwóch etapach:

1. Testy lokalne - na jednym komputerze uruchomione zostały instancje programów klienta i serwera. Klient łączy się z serwerem pod adresem `localhost`. W ten sposób jesteśmy pewni, że wszelkie błędy nie wynikają z wadliwej sieci.

2. Testy internetowe - został utworzony VPN za pomocą Hamachi, dzięki czemu klient mógł połączyć się z serwerem, który nie znajdował się fizycznie w obrębie sieci lokalnej, a połączenia musiały podróżować przez wiele węzłów.

Podczas każdego etapu testowaliśmy poprawność działania wszystkich komend, a także reakcje programów na sytuacje wyjątkowe i błędy.

Przykładowe scenariusze testowe:

- nieprawidłowe komendy aplikacji klienckiej
- niepoprawny adres serwera
- brak praw dostępu klienta do serwera
- niepoprawne ścieżki do plików
- niepoprawne argumenty żądań
- brak połączenia