Dokumentacja projektowa

Techniki Internetowe

Program obsługujący protokół NFS wraz z blokowaniem dostępu do plików w trybie jeden pisarz wielu czytelników (W12) oraz z wykorzystaniem wielu niezależnych systemów plików (W22).

Data przekazania: 25.01.2022

Skład zespołu:

- Ignacy Gołoś
- Jan Dzięgiel
- Michał Brus
- Radosław Radziukiewicz

Zadanie

Naszym zadaniem jest napisanie prostej wersji protokołu NFS (Network File System). Będziemy implementować serwer, bibliotekę kliencką oraz programy testowe sprawdzające funkcje operujące na plikach zdalnych. W naszym wariancie umożliwiamy dostęp do plików w trybie “jeden pisarz albo wielu czytelników” oraz możliwość wykorzystywania wielu różnych hierarchii plików z różnych serwerów po stronie klienta (tj. metoda mount).

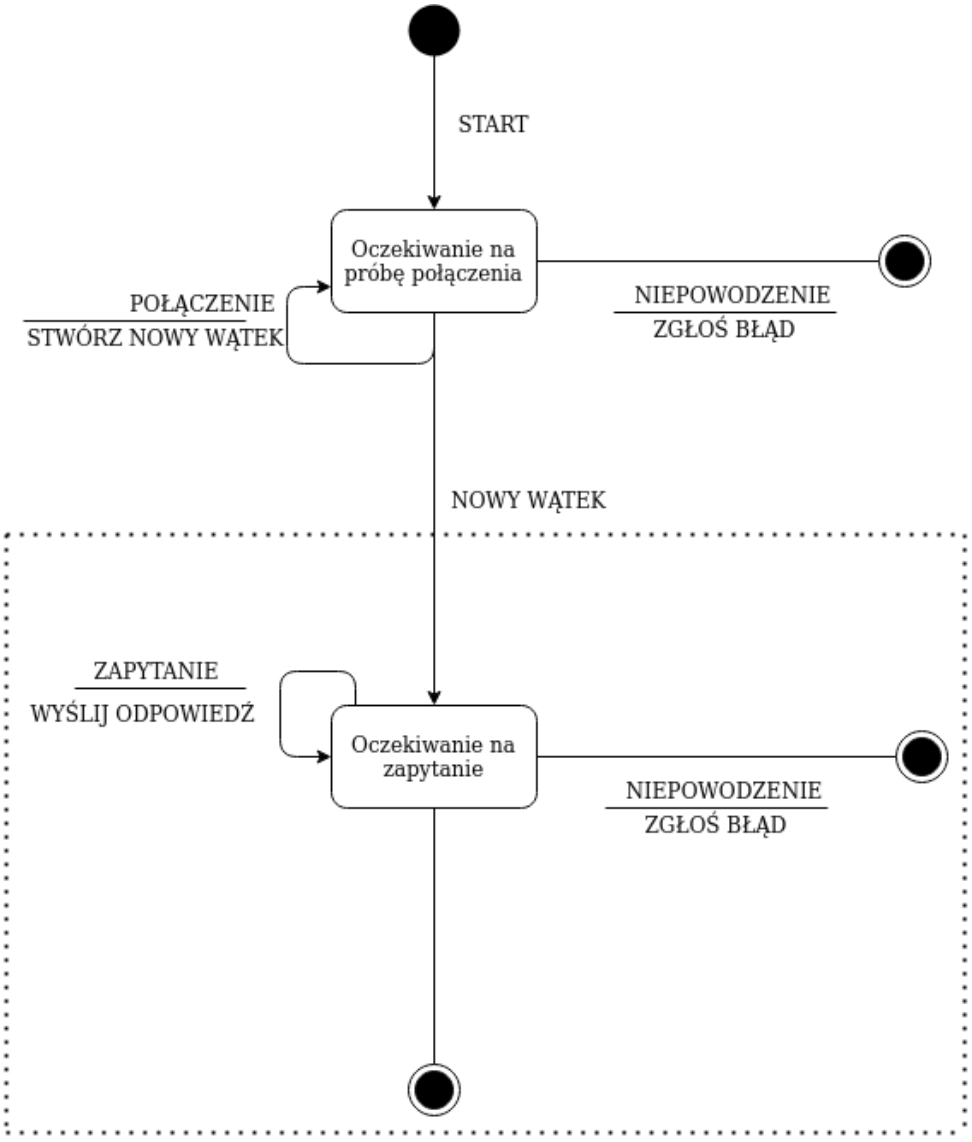
Interpretacja zadania

- Serwer otrzymuje żądania od programów klienckich wykorzystujących bibliotekę i wykonuje żądane operacje.
- Serwer wykonuje polecenia oraz przeprowadza komunikację współbieżnie, ale część jego struktur dostępnych jest tylko w trybie wzajemnego wykluczenia.
- Serwer będzie udostępniał pliki do jednoczesnego czytania przez wielu czytelników oraz do wyłącznej modyfikacji przez pisarza. W trybie wyłącznej modyfikacji (czyli wtedy kiedy plik jest zajęty przez pisarza) nowi czytelnicy, nie mogą czytać aktualnie modyfikowanego pliku. W przypadku przyjścia żądania pisarza podczas, gdy plik jest

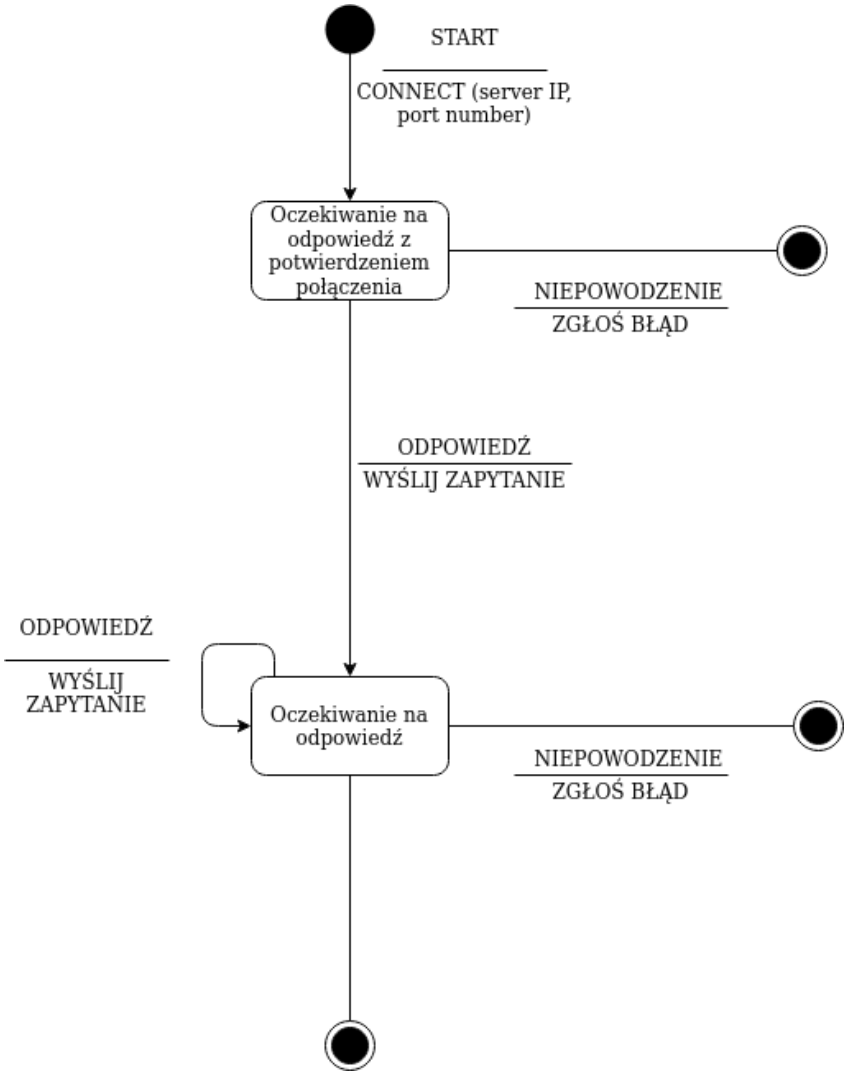
czytany przez czytelników pisarz otrzyma dostęp do pliku, czytelnicy utrzymają dostęp do odczytywania aktualnej wersji pliku. W przypadku przyjścia kolejnego żądania pisarza zwracana jest mu informacja o zajęciu zasobu przez innego pisarza.

- Program kliencki umożliwia wykonanie funkcji mount, która pozwala na montowanie systemu plików przez klienta i korzystanie z niej w sposób bezpośredni, bez konieczności bezpośredniego wskazywania serweru z plikami przy każdej komendzie.
- Plik po otwarciu przesyłany jest w całości do klienta, a informacja przechowywana na serwerze ogranicza się tylko do zapisania faktu czytania lub pisania przez danego użytkownika. W momencie wywołania metody close informacja o zamknięciu pliku wysyłana jest na serwer.
- Nie ma więc potrzeby mapowania deskryptora pliku na serwerze, gdyż ten nie jest tworzony. Tryb otwarcia jest przechowywany lokalnie w celu kontroli działania klienta. Po stronie serwera ogranicza się do sprawdzenia czy dany plik jest aktualnie zajęty przez pisarza.
- Pliki na serwerze można modyfikować, tworzyć i odczytywać, ale nie można modyfikować struktury katalogów.

Protokół komunikacyjny



Rysunek 1. Diagram zachowania serwera



Rysunek 2. Diagram zachowania programu klienckiego

Składnia wiadomości

Rodzaj wiadomości	Elementy wiadomości		
Zapytanie do serwera	DATA_SIZE (4 bajty)	REQUEST_TYPE (1 bajt opisujący typ żądania) OPEN | READ | WRITE | LSEEK | FSTAT | CLOSE | UNLINK	dodatkowe dane (opisane poniżej dla każdego typu zapytania)
Odpowiedź serwera	DATA_SIZE (4 bajty)	REPLY_TYPE (1 bajt opisujący typ odpowiedzi) ERROR_MSG | OPEN | READ | WRITE | LSEEK | FSTAT | CLOSE | UNLINK	dodatkowe dane (opisane poniżej dla odpowiedzi na każde zapytanie)

Pierwsze dwa elementy każdej wiadomości są stałe:

- 4 bajty opisujące rozmiar wiadomości (wyliczane i dołączane przez klasę wysyłającą wiadomość, nie przechowywane w strukturze wiadomości)
- 1 bajt (char) opisujący typ wiadomości

REQUEST_TYPE	Dane zapytania	Dane odpowiedzi
OPEN	char request_type = ‘O’; ushort open_mode; string file_path;	char reply_type = ‘O’;
READ	char request_type = ‘R’; string file_path;	char request_type = ‘R’; string file_path; string file_content;
WRITE	char request_type = ‘W’; string file_path; string content;	char request_type = ‘W’;
FSTAT	char request_type = ‘F’; string file_path;	char request_type = ‘F’; string file_stats;
CLOSE	char request_type = ‘Q’; string file_path;	char request_type = ‘Q’;
Dla dowolnego REQUEST_TYPE, gdy REPLY_TYPE == ERROR_MSG	Dowolne	char request_type = ‘E’; string error_msg;

Każdy typ wiadomości realizowany jest przez klasę z własnymi metodami serialize i deserialize.

Podział na moduły

Program serwera jest podzielony na trzy moduły.

- Pierwszy zajmujący się nasłuchiwaniem zapytań. Po otrzymaniu zapytania oraz po autoryzacji użytkownika otwiera nowy wątek, który zajmuje się komunikacją z klientem.
- Drugi moduł zajmuje się przetwarzaniem zapytań od klienta, wykonywaniem ich i odesłaniem efektów klientowi. Ten moduł będzie sekwencyjnie realizował polecenia od swojego klienta.
- Trzeci moduł przechowujący informacje globalne na temat stanu serwera, uprawnień, aktualnego stanu plików. Moduł ten jest zrealizowany jako singleton i jest chroniony przed współbieżnym dostępem.

Biblioteka kliencka podzielony na dwa moduły.

- Moduł komunikacji z serwerem. Obsługuje on zapytania do serwera, przyjmuje od niego dane oraz przekazuje informacje do interfejsu użytkownika.
- Interfejs użytkownika - implementuje metody widoczne i możliwe do wywołania przez klienta.

Opis biblioteki klienckiej

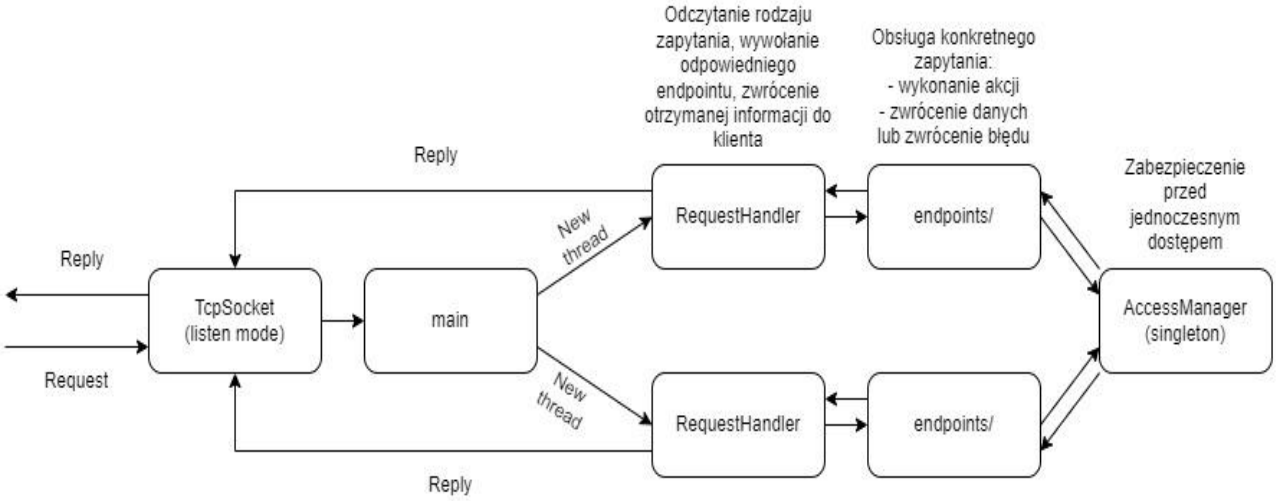
Wszystkie metody zaimplementowane w ramach programu klienckiego dostępne są poprzez instancję klasy **File System Manager**. Poszczególne metody zostały opisane poniżej:

- **mount** - pozwala połączyć się z serwerem NFS. Argumentami wywołania metody są numer IP maszyny na której pracuje serwer oraz numer portu identyfikujący proces serwera. Metoda ta umożliwia na zamontowanie wielu, niezależnych systemów. Przełączanie następuje automatycznie, w momencie wykrycia chęci “zamontowania” już połączonego serwera.
- **open** - pozwala otworzyć plik na aktualnie zamontowanym systemie plików. Argumentami wywołania metody są: pełna ścieżka do otwartego pliku oraz tryb

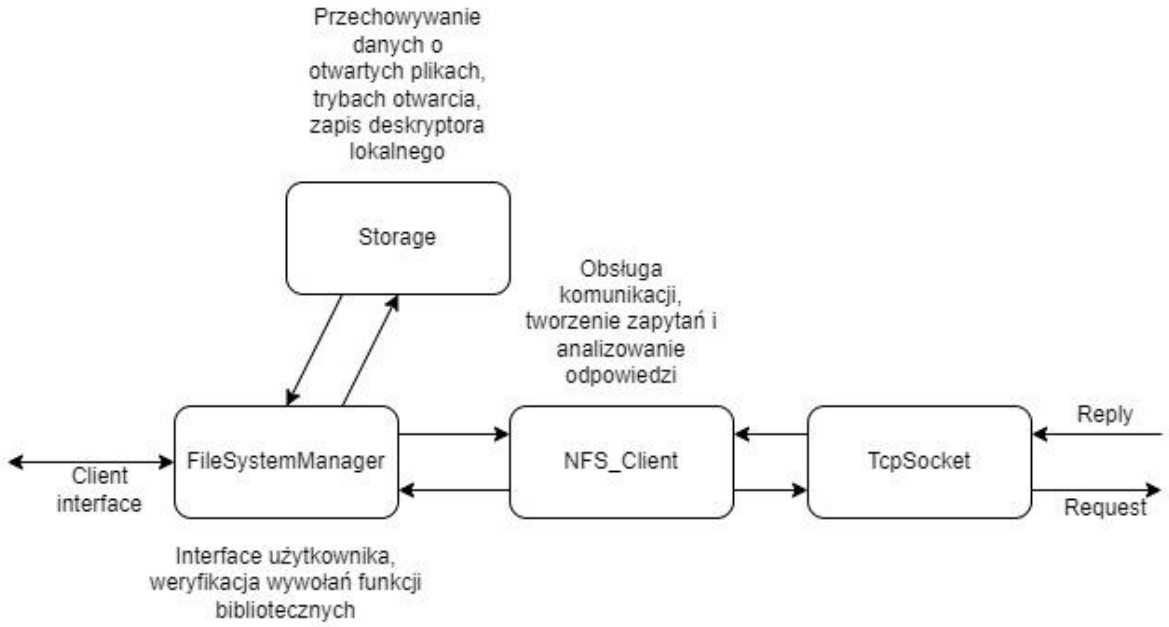
otwarcia pliku (CREATE, READ, WRITE lub READ_WRITE). Wynikiem działania funkcji open, jest numer deskryptora świeżo otwartego pliku.

- **close** - pozwala na zamknięcie pliku na aktualnie zamontowanym systemie plików. Argumentem wywołania metody jest numer deskryptora pliku, który powinien zostać zamknięty.
- **read** - pozwala odczytać zawartość pliku. Argumentami wywołania funkcji read są: deskryptor pliku, wskaźnik na bufor do którego mają zostać zapisane dane i liczba bajtów, które mają zostać odczytane z pliku. Wynikiem działania funkcji jest liczba bajtów faktycznie pobrana do bufora. Bajty pobierane są od aktualnej pozycji pliku (patrz funkcja lseek). Funkcja read nie przesuwa pozycji pliku.
- **write** - pozwala zapisać dane do pliku. Argumentami wywołania funkcji są: deskryptor pliku, wskaźnik na bufor, z którego mają zostać pobrane dane do zapisu i liczba bajtów, które mają zostać pobrane z bufora. Funkcja write zapisuje dane w miejscu wskazywanym przez pozycję pliku (patrz lseek). W momencie przekroczenia rozmiaru pliku, pozostałe znaki zapisywane są w trybie append.
- **lseek** - przesuwa pozycję pliku. Pozycja pliku określa miejsce w zawartości pliku, z którego będą pobierane dane. Argumentami wywołania funkcji są: deskryptor pliku oraz liczba całkowita określająca liczbę bajtów przesunięcia.
- **unlink** - usuwa plik z lokalnej struktury plików. Argumentem wywołania funkcji jest pełna ścieżka do pliku, który ma zostać usunięty.
- **fstat** - przesyła informacje diagnostyczne o pliku. Dane zawierają najważniejsze informacje o pliku, takie jak jego rozmiar czy moment ostatniej modyfikacji. Argumentem wywołania funkcji jest pełna ścieżka do pliku. Wynikiem działania funkcji jest komunikat diagnostyczny, dostarczony w postaci napisu.

Opis implementacji



Rysunek 3. Schemat przesyłu informacji po stronie serwera



Rysunek 4. Schemat przesyłu informacji po stronie klienta

Serwer

Klasa RequestHandler zajmuje się obsługą zgłoszeń: sprawdza typ zapytania, odczytuje dane i umieszcza je w ramach odpowiednich struktur, wywołuje kod obsługi danego zapytania umieszczony w folderze endpoints, a następnie opakowuje dane zwrócone przez odpowiednią funkcję w endpoints i wysyła do klienta.

Manager dostępu (Access Manager) to singleton tworzony po stronie serwera, odpowiadający za regulowanie dostępu do plików znajdujących się na serwerze. Pliki, razem z odpowiadającymi im informacjami o ich stanie, przechowywane są w strukturze nieuporządkowanej mapy. Informacje o pliku przechowywane są w klasie FileGueard zawierającej dwie zmienne: atomowy bool informujący o tym, czy plik w danej chwili jest otwarty przez piszącego na nim użytkownika oraz mutex chroniący plik przed błędami wynikającymi z jednoczesnego zapisu do oraz odczytu z pliku.

Blokowanie, odblokowywanie oraz sprawdzanie stanu blokady odbywa się poprzez wywołanie odpowiednio metod void block_file_for_writer(path), void remove_block(path) oraz bool is_file_blocked(path). Aby zapewnić jednolitość danych podczas wykorzystywania metod przez różne wątki, wywoływane są one na instancji managera dostępu uzyskanej poprzez wywołanie metody static AccessManager &get_instance().

Serwer zawiera również system tworzenia i zapisywania logów. Mechanizm raportowania (Logger) zaimplementowany jest jako singleton (podobnie jak manager dostępu). Logi zawierają informacje dotyczące poszczególnych wątków programu i wykonywanych przez nie operacji. Utworzenie nowego logu polega na wywołaniu funkcji Loggera *create_new_log()* z poziomu operacji, którą chcemy raportować. Wszystkie tworzone logi przekazywane są do pliku tekstowego *logs.txt.*, a dostęp do niego jest chroniony przez mutex zapewniający prawidłową obsługę zapisywania przez wiele różnych wątków.

Realizacja połączenia

Połączenie pomiędzy klientem i serwerem realizowane jest przy pomocy autorskiej implementacji interfejsu gniazda TCP. Implementacja dostarczona jest w formie samodzielnej biblioteki, wykorzystywanej przez jednocześnie przez program kliencki oraz serwer.

Implementacja gniazda pozwala na automatyczne wysyłanie wiadomości w postaci typu napisowego. Gniazda wspierają mechanizm defragmentacji wiadomości po stronie odbierającej. Implementacja mechanizmu wykorzystuje kodowanie rozmiaru wiadomości oraz dołączanie go w formie swoistego “nagłówka” do przesyłanego komunikatu.

Na najniższym poziomie implementacji, gniazdo wykorzystuje bibliotekę socket systemu linux. Wzbogaca ona jednak dostępne mechanizmy poprzez dodanie konstrukcji typowych dla języka C++ (takich jak mechanizm wyjątków).

Istotną cechą stworzonej implementacji klasy socket jest możliwość scalania otrzymanych porcji danych. Dzięki załączonemu “nagłówkowi”, zawierającemu rozmiar przesyłanej, zakodowanej wiadomości, gniazdo jest w stanie oczekiwać na otrzymanie wszystkich bajtów wchodzących w skład przesyłanej wiadomości. Mechanizm ten pozwala zapewnić bezpieczną i wydajną komunikację pomiędzy stworzonym serwerem oraz programem klienckim. Warto zaznaczyć, iż w momencie przesyłu danych, gniazdo monitoruje stan połączenia pomiędzy stronami. Ewentualne wykrycie sytuacji patologicznej (nagłe zamknięcie połączenia, zły format komunikatu) zgłaszane jest w postaci odpowiedniego wyjątku.

Wszystkie komunikaty przesyłane są w formacie danych napisowych, dostępnych w języku C++. Przesyłana wiadomość powinna ulec zakodowaniu do odpowiedniego napisu, który następnie może zostać przesłany w sposób bezpieczny do serwera.

Program kliencki

Implementacja programu klienckiego została zrealizowana przy pomocy zestawu specjalnych klas, zapewniających lokalne środowisko dostępu do serwera NFS. Zadaniem wspomnianych klas jest zapewnienie poprawności komunikatów wysyłanych do serwera, lokalne przechowywanie pobranych danych oraz obsługa wyjątków. Poniżej znajduje się dokładniejszy opis klas tworzących program kliencki:

- **File System Manager** - naczelna klasa odpowiedzialna za poprawne funkcjonowanie programu klienckiego. Zawiera implementację całości “logiki” programu. Klasy tej metody powinny być wykorzystywane w celach dostępu do systemu.
- **Storage** - klasa odpowiedzialna za funkcjonowanie lokalnej wersji systemu plików. Pełni ona rolę pamięci podręcznej stanu połączenia oraz lokalnej kopii systemu plików.
- **File** - klasa odpowiedzialna za lokalną reprezentację pliku. Udostępnia ona funkcjonalności niezbędne do realizacji poprawnego dostępu do pojedynczego pliku.
- **NFS Client** - klasa odpowiedzialna za obsługę zapytań i odpowiedzi. Tworzy ona warstwę abstrakcji dla głównej klasy, aby ta nie musiała zajmować się tworzeniem struktur zapytań i odpowiedzi oraz analizą danych zwróconych od serwera.

Wykorzystywane narzędzia

Program napisany został w języku C i C++ z wykorzystaniem gniazd BSD. Ponadto, do implementacji rozwiązania wykorzystano narzędzia CMake i skrypty napisane w języku Bash. Biblioteki message (request.h, reply.h), socket.h oraz program kliencki został skompilowane do postaci dynamicznych bibliotek i dołączane podczas uruchomienia programu.

Oprogramowanie wersjonowane było przy użyciu systemu kontroli wersji Git ze zdalnym repozytorium umieszczonym na platformie GitHub.

Testowanie rozwiązania odbywało się również za pomocą maszyn wirtualnych udostępnionych przez Google Cloud Platform. Na jednej z nich zainstalowane zostało oprogramowanie napisanego przez nas serwera, do którego połączenie odbywało się za pomocą publicznego IP maszyny wirtualnej. Dzięki temu można zweryfikować działanie systemu na rzeczywistej sieci.

Testy

W ramach testowania przygotowanego rozwiązania, przygotowaliśmy zestaw testów, mający na celu weryfikację poprawności implementacji. Są one dostępne w folderze client_test_program. Poniżej znajduje się dokładny opis przygotowanych testów.

- **lseek** - w ramach testowania poprawności implementacji funkcji lseek przygotowano testy sprawdzające przesunięcie o: zerową, dodatnią, dodatnią znaczną, ujemną oraz ujemną znaczną liczbę bajtów. Rezultat operacji weryfikowany był poprzez porównanie wyniku funkcji read, wykonanej na pliku, do którego odnosiła się procedura lseek.
- **open** - w ramach testowania otwierania pliku przygotowano testy sprawdzające otwarcie pliku z różnymi trybami oraz zachowanie podczas przekazania błędnego parametru (nieistniejącego trybu otwarcia lub nieistniejącego pliku poza trybem create). Ponadto zweryfikowano zachowanie podczas próby otwarcia pliku zablokowanego przez pisarza lub przez innego klienta pracującego równocześnie.
- **close** - w ramach testowania zamykania pliku przygotowano testy sprawdzające zamknięcie pliku z poprawnym deskryptorem, nieistniejącym deskryptorem oraz próbę zamknięcia deskryptora należącego do innego klienta pracującego równocześnie.

- **fstat** - w ramach testowania uzyskiwania informacji dotyczących danego pliku przygotowano testy sprawdzające zachowanie w przypadku błędów podczas otwierania pliku (w celu pozyskania deskryptora używanego do wywołania systemowej metody fstat). Dodatkowo sprawdzane jest zachowanie w przypadku gdy plik jest używany przez innego klienta.
- **read** - w ramach testowania poprawności implementacji funkcji read przygotowano zestaw testów sprawdzających zachowanie podczas przekazania błędnego parametru lub użycia nieprawidłowego trybu otwarcia. Do tego weryfikowane jest działanie w przypadku wielu użytkowników korzystających z danego pliku oraz jednego użytkownika korzystającego z wielu plików. Sprawdzane są także interakcje funkcji z funkcją lseek.
- **write** - w ramach testowania poprawności implementacji funkcji write przygotowano zestaw testów sprawdzających zachowanie podczas przekazania błędnego parametru lub użycia nieprawidłowego trybu otwarcia. Do tego weryfikowane jest działanie w przypadku wielu użytkowników korzystających z danego pliku oraz jednego użytkownika korzystającego z wielu plików. Sprawdzane są także interakcje funkcji z funkcją lseek.
- Dodatkowo testowana jest poprawność interakcji funkcji **write** i **read** podczas używania ich jednocześnie przez różnych użytkowników na jednym pliku.

Wyniki testów

Udało się przeprowadzić wszystkie zaplanowane testy. Początkowo został zlokalizowane błędy związane z:

- otwieraniem pliku w trybie do czytania przez dwóch użytkowników,
- formułą aktualizującą wskaźnik pliku.

Zostały jednak naprawione i w ostatniej wersji wszystkie testy zakończyły się powodzeniem.