Techniki Internetowe (TIN)- 2020Z

Projekt: Aplikacja NFS - Sprawozdanie końcowe

Zespół w składzie:
– Mateusz Kalinowski
– Mateusz Kiljan
– Paweł Szafrański
– Paweł Wieczorek

Data przekazania: 7.01.2021

Spis treści

Treść zadania

Celem projektu jest zaimplementowanie sieciowego dostępu do zewnętrznego systemu plików poprzez stworzenie biblioteki klienckiej, programu serwera oraz aplikacji klienckiej. Komunikacja zgodnie z treścią zadania powinna bazować na TCP. Przygotowane rozwiązanie ma udostępniać podstawowe funkcje do operacji na plikach, tj. otwarcie, odczyt, zapis, przesunięcie

(*lseek*), usuwanie oraz pobieranie właściwości (*fstat*). Ponadto, powinno umożliwiać blokowanie dostępu do plików w trybie „jeden pisarz albo wielu czytelników” (W12) oraz umożliwiać jednoczesny dostęp do wielu niezależnych serwerów z poziomu aplikacji klienckiej (W32).

Doprecyzowanie treści

Blokowanie w trybie „jeden pisarz albo wielu czytelników” jest realizowane z wykorzystaniem nowej flagi *O_MYNFS_LOCK* przekazywanej w argumencie *oflag* funkcji *mynfs_open*. Aplikacja chcąca uzyskać blokowanie „jeden pisarz” powinna wywoływać funkcję *mynfs_open* z tą nowododaną flagą oraz klasycznym *O_RDWR*. Serwer po wykryciu flagi *O_MYNFS*_LOCK podejmie próbę założenia blokady na otwieranym pliku z użyciem syscall’a *flock* (w trybie nieblokującym). Jeżeli założenie blokady nie będzie możliwe, z powodu tego, że inny klient już takową założył, serwer zwróci klientowi jednoznaczny komunikat o błędzie. Zdjęcie blokady jest realizowane poprzez zamknięcie pliku funkcją *mynfs_close* lub w momencie utraty połączenia z klientem (np. z powodu *timeout-u*). Poprawna obsługa blokady, w szczególności nie doprowadzenie do sytuacji, w której dwóch klientów otwiera ten sam do pliku do zapisu, ale tylko jeden korzysta z mechanizmu blokady, spoczywa wyłącznie w gestii klienta.

Oprócz tego, serwer przechowuje deskryptory lokalnie otwartych plików, a użytkownik dysponuje wyłącznie identyfikatorem pliku. Serwer przechowuje również listę realizującą mapowanie między deskryptorami lokalnie otwartych plików serwera, a identyfikatorami przekazywanymi klientom. Ponadto, serwer przechowuje listę z informacjami o otwartych plikach dla sesji klienta, tak żeby móc automatycznie zamknąć wszystkie deskryptory po jej zakończeniu. Dzięki temu poprawnie zostanie obsłużona sytuacja, gdy klient zerwie połączenie przed zamknięciem swoich plików.

Opis funkcjonalny

W ramach przygotowywanego projektu planujemy zaimplementować następujące funkcjonalności:

Po stronie klienta:

1. **Odczyt z oraz zapis do plików** - odpowiedniki systemowych read() oraz write()
2. **Usuwanie plików** - w sytuacji, gdy użytkownik próbuje usunąć otwarty plik, operacja ta zostanie wykonana dopiero gdy dany plik zostanie zamknięty u każdego użytkownika
3. **Przesunięcie kursora** - odpowiednik funkcji systemowej lseek()
4. **Pobieranie właściwości pliku** - odpowiednik funkcji systemowej fstat()
5. **Otwieranie plików** – można otworzyć plik w trybach *O_RDONLY, O_WRONLY, O_RDWR, O_CREAT* oraz *O_MYNFS_LOCK*

Po stronie serwera:

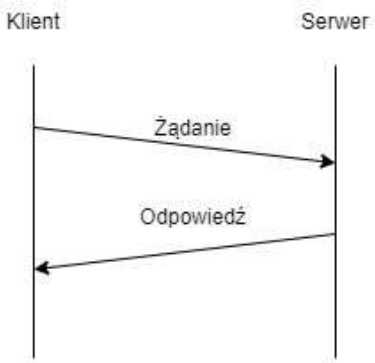
1. **Autoryzacja użytkowników z wykorzystaniem mechanizmu *Linux-PAM*** – takie rozwiązanie pozwala na wykorzystanie mechanizmu uprawnień obecnego w systemie Linux do prostego ograniczenia uprawnień dostępu do katalogów i plików
2. **Wsparcie dla prostego pliku konfiguracyjnego** – parametry działania aplikacji serwerowej jak: ścieżka do udostępnianego katalogu, maksymalna ilość jednocześnie otwartych deskryptorów czy lokalizacja pliku z logami będą podawane poprzez prosty plik konfiguracyjny o składni zbliżonej do tej stosowanej w pliku *.profile* (i.e. *nazwa=wartość*).
3. **Logowanie zdarzeń na terminal oraz do pliku lokalnego** – wszystkie istotne zdarzenia (w szczególności: logowanie nowego użytkownika, otwarcie czy usunięcie pliku, błędy etc.) będą odnotowane na wyjściu programu (*stdout*) oraz w osobnym pliku wraz z informacją o dacie wystąpienia oraz identyfikatorze użytkownika, który je spowodował.

4. **Obsługa przekroczenia czasu (timeout)** – dla każdego klienta serwer przechowuje informacje o czasie, do którego jego sesja jest ważna. Ilekroć klient wysyła żądania do serwera, czas ten jest ustawiany na 5 minut do przodu. W głównej pętli obsługującej socket'y, serwer periodycznie sprawdza czy czas któregoś z klientów nie uległ przekroczeniu; jeśli zaszła taka sytuacja, połącznie z klientem jest zamykane (close), a stosowna informacja trafia do logów.

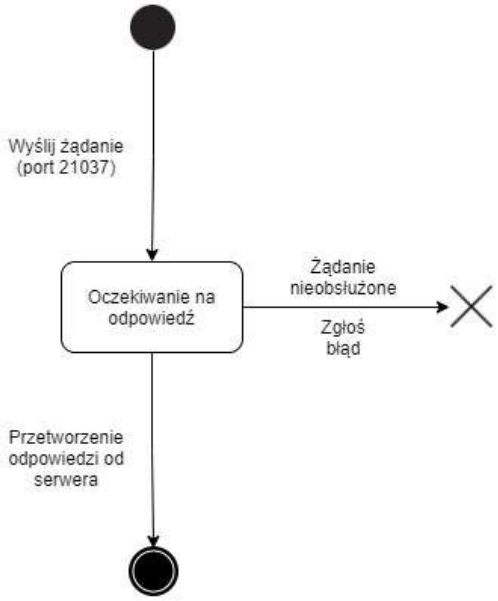
Protokół komunikacyjny

Komunikacja pomiędzy serwerem, a klientem obywać się będzie z wykorzystaniem socketów TCP: głównego stale otwartego oraz socketów indywidualnych dla każdego zgłaszającego się klienta, które podczas pracy serwera będą otwierane i zamykane. Każdy z tych socketów działa na porcie 21037. Komunikacja z klientem zostaje przerwana po upływie zadanego okresu (5 minut) bezczynności.

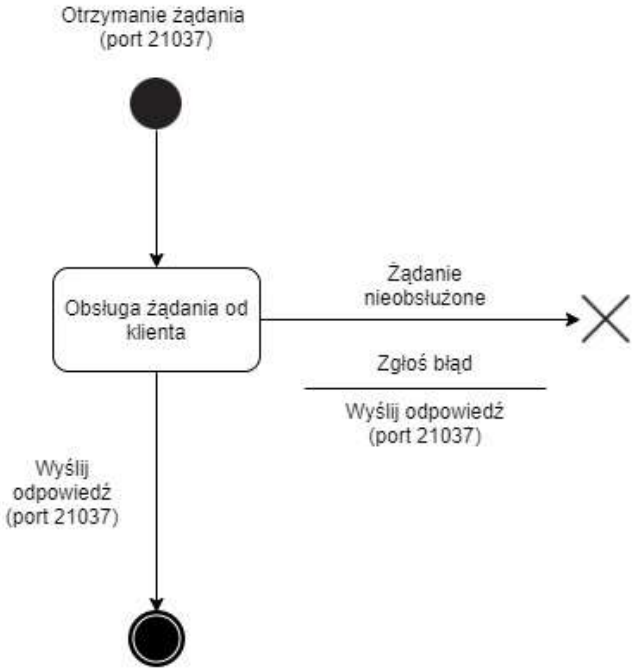
Przepływ informacji pomiędzy klientem a wybranym serwerem wygląda w następujący sposób:



Rys. 1. Schemat komunikacji pomiędzy klientem, a serwerem

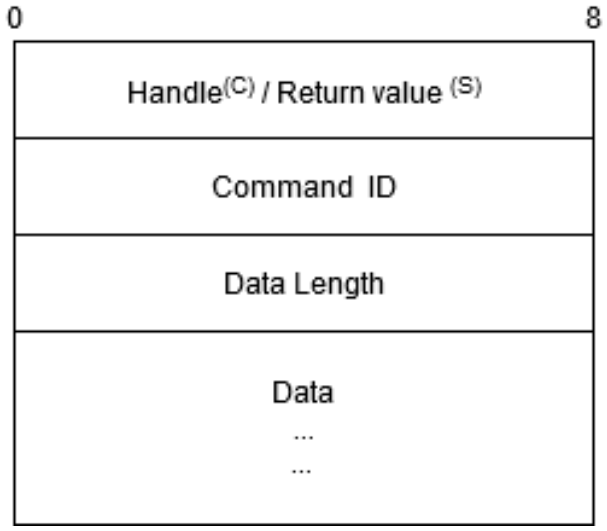


Rys. 2. Diagram stanów klienta



Rys. 3. Diagram stanów serwera

Każdy komunikat danych będzie enkapsulowany w następujący format:



Rys. 4 Budowa nadrzędnego komunikatu
(S) – pole używane w danych zwracanych z serwera,
(C) – w danych od klienta

Odpowiednikiem powyższego komunikatu jest następująca struktura w języku C:

Lst. 1. – Budowa komunikatu

```
struct mynfs_message_t {
   union {
      int64_t handle;
      int64_t return_value;
   };
   uint64_t cmd;
   size_t data_length;
   char data[0];
}
```

Rys. 5 Budowa nadrzędnego komunikatu jako struktury w języku C

Opis komunikatu

W ogólności, pole *handle* przechowuje identyfikator pliku, na którym ma zostać wykonana żądana operacja, pole *return_value* wartość zwróconą przez funkcję serwera, *cmd* kod operacji do wykonania, a *data_length* długość danych przekazywanych w tablicy *data*. Format danych zawartych w polu *data* zależny jest od rodzaju wywoływanej funkcji – szczegóły dla każdej z planowanych komend są przedstawione w poniższej tabeli:

CMD	Nazwa	Dane wejściowe – format C	Dane wyjściowe

0	mynfs_open	struct mynfs_open_t{
		int oflag;
		int mode;
		size_t path_length;
		char name[0];
		}
			brak – uchwyt zwracany jest w polu *return_value* pakietu nadrzędnego

1	mynfs_read	struct mynfs_read_t {
		size_t length;
		}
			tablica *data* zawiera odczytane bajty o długości określonej w *data_length*

2	mynfs_write	struct mynfs_write_t {
		size_t length;
		char buffer[0];
		}
			brak – ilość zapisanych bajtów zwraca jest w polu *return_value* (analogicznie jak w funkcji *read* z stdlib)

3	mynfs_lseek	struct mynfs_read_t {
		size_t offset;
		int whence;
		}
			brak

4	mynfs_close	**brak** – identyfikator jest przekazywany w polu *handle*	**brak**
5	mynfs_fstat	**brak** – identyfikator jest przekazywany w polu *handle*	tablica *data* zawiera strukturę *stat* zdefiniowaną w *sys/stat.h*

6	mynfs_unlink	struct mynfs_open_t{
		size_t path_length;
		char name[0];
		}
			brak

Budowa projektu

Projekt składa się z trzech części: aplikacji serwera, biblioteki klienckiej z rozszerzeniem .so oraz aplikacji klienckiej korzystającej z tej biblioteki. Implementacja każdej z tych części będzie podzielona na wydzielone moduły.

Aplikacja kliencka będzie składała się z modułu odpowiedzialnego za interfejs użytkownika i z modułu realizującego komunikację z serwerem (wykorzystanie funkcji z biblioteki). Podczas korzystania z aplikacji, klient będzie posiadał możliwość wyboru jednego z wielu niezależnych systemów plików znajdujących się na oddzielnych architekturach (maksymalnie 3). Możliwe będą zarówno inicjalizacja połączenia jak i korzystanie z tychże systemów w tym samym momencie. Po stronie klienta przechowywane będą informacje o serwerach tzn.: adresy IP oraz numery socketów, na których dany serwer nasłuchuje żądań.

Aplikacja serwera będzie składała się z trzech modułów:

1. **Moduł obsługi żądań.** Odpowiada on za nasłuchiwanie na głównym socketcie żądań od klientów. Po otrzymaniu prawidłowego żądania tworzy nowy socket (również na porcie 21037), który będzie służył do dalszej komunikacji z konkretnym klientem. Zwraca klientowi nowy numer portu.
2. **Moduł komunikacji z klientem.** Odpowiada za przyjmowanie wszystkich nadchodzących poleceń od klienta oraz wysyłaniu mu odpowiednich danych i komunikatów. W celu wykonania tych poleceń przekazuje je do modułu obsługi plików.
3. **Moduł obsługi plików.** Odpowiada za operacje na lokalnych plikach serwera oraz kontrolowanie blokowania w trybie „jeden pisarz albo wielu czytelników”. Zwraca modułowi komunikacji z klientem odpowiednie dane oraz komunikaty diagnostyczne.

Zastosowane narzędzia

Aplikacja serwera została napisana w języku C z użyciem standardowych bibliotek libc oraz modułu PAM, służącego do autoryzacji użytkowników. Biblioteka współdzielone oraz aplikacja kliencka wykonane są w języku C++, bez żadnych niestandardowych bibliotek. Za systemu kontroli wersji posłużył git, a system budowania oparty został o program *make*.

Opis aplikacji klienckiej

Aplikacja kliencka składa się z jednego programu, w którym wykonywana jest nieskończona pętla. Użytkownikowi prezentowane są stosowne komunikaty, w zależności od operacji jaką ma wykonać. Na samym początku proszony jest o wprowadzenie adresu IP oraz portu dla zdalnego serwera oraz loginu i hasła. Później ma on do wyboru kilka operacji takich jak : open, read, unlink, close, lseek, showhosts, hostchange, hostadd, hostremove, exit. Z założenia użytkownik może mieć skonfigurowanych maksymalnie trzech hostów. Polecenia będące odpowiednikami funkcji z biblioteki wykonywane są na jednym z wybranych hostów (pierwszym wybranym hostem jest pierwszy wprowadzony) i będą wykonane na nim do momentu, w którym nie zostanie on zmieniony za pomocą polecenia ‘hostchange’. Użytkownik może dodać, usunąć oraz zobaczyć aktualnie przechowywanych hostów. Główna pętla programu nie korzysta bezpośrednio z funkcji bibliotecznych. Są one opakowane w dodatkowe funkcje rozszerzające funkcjonalność funkcji bibliotecznych o wprowadzanie i weryfikację danych, wyświetlanie błędów, wybór deskryptora pliku etc. Dodatkowo zostały napisane funkcje pomocnicze łączące powtarzające się funkcjonalności np.: wybieranie deskryptora pliku, rozdzielanie adresu IP od portu, zmiana hosta etc. Po 5 minutach bezczynności w programie program automatycznie kończy swoje działanie.

Funkcjonalność oraz implementacja

Aplikacja serwera

Serwer został napisany w języku C. Wykonanie programu serwera można podzielić na 2 części, inicjalizację oraz główną pętlę.

W trakcie inicjalizacji odczytujemy plik konfiguracyjny, tworzymy plik z logiem, a następnie tworzymy socket, który będzie oczekiwać na nowych klientów. Socket ten jest przypisywany do konkretnego interfejsu i portu za pomocą funkcji *bind* (interfejs oraz port na którym nasłuchujemy określa plik konfiguracyjny). Po przypisaniu socket'a rozpoczynamy nasłuchiwanie za pomocą funkcji *listen.* Maksymalna ilość klientów jaka może czekać na akceptację przez serwer jest również określona w pliku konfiguracyjnym.

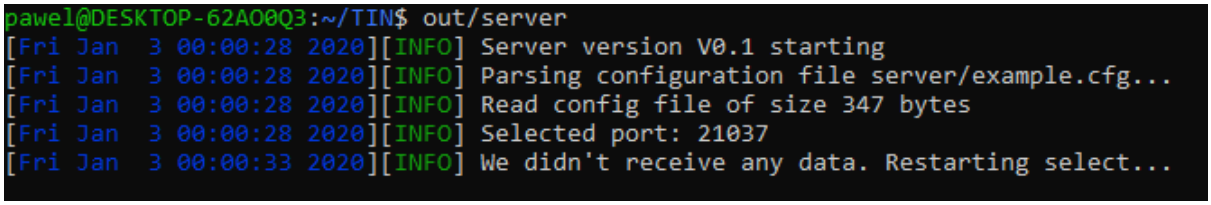
W głównej pętli zajmujemy się przyjmowaniem nowych klientów oraz obsługą już zaakceptowanych klientów. Aby uniknąć blokowania się na oczekiwaniu na dane użyliśmy funkcji *select* do sprawdzenia czy na którymś z socketów nie ma nowych danych. Jako nowe dane rozumiemy pojawienie się nowego klienta lub wysłanie żądania przez klienta już obsługiwanego. Jeśli pojawili się nowi klienci to wszystkich ich akceptujemy za pomocą funkcji *accept*, a następnie przystępujemy do obsługi żądań istniejących klientów. Dane od tych klientów odczytujemy za pomocą odwołań do funkcji *read (wpierw odczytywany jest nagłówek informujący m.in. o wielkości danych, a następnie same dane)*, a następnie kompletny pakiet jest przekazywany do handlera. Handler obsługuje żądanie, a następnie generuje odpowiedź, którą wysyłamy klientowi za pomocą funkcji *write*. W przypadku przesłania przez klienta żądania zamknięcia pliku po obsłużeniu tego żądania zamykamy również socket obsługujący danego klienta.

Handler sprawdza rozmiar komunikatu oraz poprawność zawartych w nich danych, a następnie przekazuję na konstrukcję *switch*, która przetwarza poszczególne komendy. Dla każdej z komend, wykonywana jest stosowna operacja na pliku, a następnie do klienta odsyłany jest jej wynik wraz ze zwróconą wartością (w przypadku wystąpienia błędu jest to *errno*). Ponadto, kod handlera odpowiada również za przechowywanie deskryptora otwartego pliku oraz monitorowanie czasu pozostałego do przymusowego odłączenia użytkownika (tzw. *timeout*).

Oprócz obsługi poleceń od klienta, serwer wyposażony jest w moduł do przetwarzania prostego pliku konfiguracyjnego, dostarczającego wszystkich koniecznych informacji do uruchomienia aplikacji, oraz wypisywania na stdout i do pliku logów programu, zawierających informacje o rodzaju wiadomości (informacja, ostrzeżenie, błąd) oraz dacie jego wystąpienia.

Lst. 2. – Przykładowy plik konfiguracyjny serwera

```
# Example configuration file
# Port on which server should listen for incoming connections
PORT = 21037
# Hostname of the server
HOSTNAME = 0.0.0.0
# Path to directory shared by this NFS server
NFS_PATH = /home/TIN/directory
# Whether logs should be printed in color
COLORED_LOGS = true
# How many clients can wait in queue for acceptance
CLIENT_QUEUE = 8
```



Rys 6. Wygląd logów programu wypisywanych na stdout

Biblioteka kliencka

Biblioteka kliencka została napisana w języku C++. Składa się z dwóch części: z części implementującej operacje na socketach oraz korzystającej z niej części implementującej wszystkie funkcje udostępniane użytkownikowi. Są to funkcje: *mynfs_open, mynfs_read, mynfs_write, mynfs_lseek, mynfs_close, mynfs_fstat, mynfs_unlink*.

Na każdy otwarty przez klienta plik przypada jeden powiązany z nim socket. Socket jest tworzony i łączony z serwerem po wywołaniu funkcji *mynfs_open*. Funkcja ta zwraca (poprzez wskaźnik w argumencie) deskryptor nowo utworzonego socketu, po czym nim komunikat zgodny z tabelką powyżej. Dzięki w ten sposób pozyskanemu deskryptorowi klient chcąc przesłać kolejne polecenia do serwera używa tego samego socketa (musi przekazać socketFd skojarzony z wybranym plikiem jako argument do funkcji bibliotecznej).

Socket jest zamykany przy wywołaniu funkcji *mynfs_close*. Zostaje wtedy wysłany komunikat z komendą zamknięcia pliku o zadanym deskryptorze, a po uzyskaniu odpowiedzi z serwera następuje zamknięcie socketa.

Każda z tych funkcji przygotowuje odpowiedni komunikat do przesłania przez socket wykorzystując odpowiednie struktury z tabelki powyżej. Struktury podrzędne (związane ze specyfiką konkretnej funkcji) są odpowiednio alokowane wewnątrz pola *data* komunikatu nadrzędnego.

Aplikacja klienta

Została ona wykonana w języku C++ wraz z wykorzystaniem biblioteki std oraz programu make. Główne funkcjonalności to:

- Korzystanie z funkcji bibliotecznych
- Konfiguracja hostów
- Konfiguracja loginu i hasła
- Wyświetlanie informacji o błędach

Szczegółowy opis działania aplikacji klienckiej opisany został w akapicie wyżej. Aplikacja została przetestowana ręcznie “na sucho” wykorzystując wydmuszki funkcji bibliotecznych. Zostały sprawdzone różne warunki takie jak: niepoprawna liczba hostów, niepoprawne komendy, usunięcie nieistniejącego hosta, niepoprawne dane. Została ona przetestowana również na działającej bibliotece wraz z serwerem, a rezultaty opisane są poniżej.

Obsługa błędów

Obsługa błędów została zaimplementowana po stronie klienta jak i po stronie serwera. Wszelkie błędy po stronie aplikacji klienckiej są ładowane do strumienia std::cerr i wyświetlane użytkownikowi na ekranie. Typami błędów które zaliczamy do tej grupy są: niepoprawnie utworzony socket, niepoprawna konwersja adresu IP, błąd przy otwieraniu połączeniu na danym sockecie.

Błędy występujące po stronie serwera przekazywane są do klienta w polu *return_value*

naszego komunikatu. Wyróżnione zostało 6 rodzajów błędów opisanych w poniżej zamieszonej tabeli.

Nazwa błędu	Kod błędu
MYNFS_INVALID_CLIENT	-1000
MYNFS_INVALID_PACKET	-500
MYNFS_UNKNOWN_COMMAND	-499
MYNFS_ALREADY_OPENED	-498
MYNFS_OVERLOAD	-497
MYNFS_CLOSED	100

W przypadku przesłania poprawnego komunikatu do pola return_value umieszczany jest kod o wartości 0 (w naszej strukturze ma on nazwę 'MYNFS_SUCCESS').

Za przekazywanie kodu błędu pomiędzy aplikacją kliencką, a serwerem odpowiedzialna jest biblioteka, w której dana funkcja po odczytaniu zawartości komunikatu zwraca kod błędu. Następnie kod ten odbierany jest w aplikacji klienckiej i jeśli nie jest on równy zeru to mapowany jest on na komunikat i wyświetlany w konsoli użytkownika.

Testowanie

Na potrzeby wiarygodnego przetestowania poprawności opracowanego systemu przygotowywaliśmy zestaw scenariuszy sprawdzających najpoważniejsze i najbardziej powszechne rodzaje błędów występujących przy komunikacji z użyciem socketów sieciowych, takich jak błędne wykorzystanie funkcji *read/write* czy obsługa *timeout-ów*.

Przesyłanie dużej ilości komunikatów

W pierwszym kroku zbadaliśmy czy nasz serwer oraz klient radzą sobie poprawnie z dużą ilością, często przychodzących pakietów. Jako że TCP jest protokołem strumieniowym, istnieje możliwość, że małe operacje *write* zostaną zbuforowane i przesłane w ramach tylko jednego pakietu. W przypadku, gdyby serwer wykorzystywał trywialną metodę odczytu danych z gniazda (np. *read(fd, buffer, BUFFER_SIZE*), mógłby naraz wczytać więcej niż jeden lub niepełny pakiet danych.

W celu sprawdzenia tego aspektu, zmodyfikowaliśmy aplikacje kliencką tak, by wysłała dane nie czekając na odpowiedź z serwera. Gdyby, serwer niepoprawnie obsługiwał czytanie z socketów, przychodzące pakiety powinny ulec złączeniu i stać się niezdatne do przetworzenia.

Wykonane badanie nie pokazało żadnego problemu z obsługą licznych pakietów. Działo się tak, dzięki faktowi, że serwer wpierw wczytuje z gniazda nagłówek pakietu (zawierający m.in. jego rozmiar), a dopiero później czyta docelową ilość bajtów, tym samym poprawnie obsługując sytuację, w których bufor zawiera więcej niż jeden komunikat. Ponadto, taka metoda rozwiązuje również problem wczytywania komunikatów, które nie mieszczą się w buforze – funkcja *read* zostanie wywołana wielokrotnie, aż odczyta właściwą ilość danych.

Pomiar przekroczenia czasu (timeout)

Następnym etapem było sprawdzenie czy serwer radzi sobie z obsługą klientów, którzy nie dokończyli transmisji i powinni zostać automatycznie wyrzuceni po upływie pewnego okresu. Nie zapewnienie właściwej obsługi takiego scenariusza mogłoby doprowadzić do zużycia wszystkich

zasobów maszyny hostującej serwer, w szczególności deskryptorów plików, które nie zostały poprawnie zamknięte.

Test ten wykonaliśmy ponownie z użyciem zmodyfikowanej aplikacji klienckiej: tym razem tworzyła ono połączenie z serwerem, otwierała plik, a następnie porzucała gniazdo i powtarzała procedurę od początku. Równolegle na drugim terminalu sprawdzaliśmy logi z serwera, by ustalić zachowanie serwera na taki scenariusz.

Po upływie 5 minut od rozpoczęcia testu w logach pojawiła się informacja, że serwer rozłączył pierwszego klienta z powodu timeout'u. Wywołanie komendy *lsof* na pliku otwartym przez pierwszego klienta potwierdziło, że deskryptor pliku został poprawnie zamknięty, a działanie serwera nie budzi żadnych zastrzeżeń.

Przetwarzanie niepoprawnego pakietu

W kolejnym etapie skupiliśmy się na zbadaniu samego przetwarzania komunikatów przez serwer. Szczególny nacisk położyliśmy na potencjalne błędy związane z obsługą pamięci w przypadku odebrania wadliwego pakietu. Poprawna obsługa sytuacji wyjątkowych jest szczególnie istotna z perspektywy aplikacji internetowej, gdyż jako serwis wystawiony na świat zewnętrzny, może być łatwym celem dla ataków.

Pierwszym krokiem było skompilowanie aplikacji serwera kompilatorem *clang* z włączonym *Address Sanitizer-em* (ASan), aktywowanym poprzez dodanie flagi kompilacji `*-fsanitize=address*`. Ten zabieg pozwoli nam na dużo prostsze i pewniejsze znajdywanie błędów związanych z obsługą pamięci. Następnie wysyłaliśmy do serwera niepoprawne na różne sposoby pakiety: zły rozmiar, zły identyfikator plików, brak *NULLa* na końcu stringa. Nie zdecydowaliśmy się na zastosowanie bardziej wyrafinowanych narzędzi, takich jak fuzzery (AFL) czy symboliczne wykonanie (KLEE), gdyż uznaliśmy iż wykracza to poza ramy tego projektu. Poza tym, format danych jest na tyle prosty, że możliwości jego błędnego przetworzenia są mocno ograniczone.

Testowanie serwera nie wskazało w finalnej wersji programu żadnych nieprawidłowości – dla każdego błędnego pakietu, serwer poprawnie zareagował poprzez odesłanie stosownego kodu błędu.

Obsługa blokady „jeden pisarz"

Dalej, zbadaliśmy działanie funkcji blokowania dostępu w trybie „jeden pisarz, wielu czytelników". Przedmiotem badania, było stwierdzenie czy serwer uniemożliwi otwarcie tego samego pliku z flagą *O_MYNFS_LOCK* przez dwóch klientów jednocześnie.

W tym celu, na jednym kliencie otworzyliśmy plik z powyższą flagą. Następnie, z poziomu drugiego klient, wykonaliśmy identyczną operację. Zgodnie z założeniem projektowym, serwer zwrócił błąd, informując użytkownika, że ten plik jest już zablokowany.

Poprawność przesyłanych danych

Na koniec sprawdziliśmy działanie najważniejszej funkcjonalności przygotowywanego programu – poprawności przesyłanych danych. W szczególności, interesowało nas czy żądanie utworzenia i zapisu pliku przez klienta, poskutkuje utworzeniem na naszym NFS-ie pliku o właściwej zawartości.

Test z użyciem aplikacji klienckiej polegał na utworzeniu pliku X, zapisaniu do niego zawartości, a następnie sprawdzeniu czy w katalogu naszego NFSa się on znajduje oraz czy dane w nim są poprawne. Testowe uruchomienia wykazały, że system spełnia swoją najbardziej podstawową rolę i tym samym może być nazywany sieciowym systemem plików, a 1.5 miesięczny projekt można nazwać sukcesem.