Projekt z Sieci

Michał Sobczak, Szymon Potrzebowski

22 czerwca 2025

Przykładowy scenariusz użycia

Klient na samym początku umieszcza pliki, które chce udostępnić w folderze `root`. Następnie uruchamia program, dołączając nazwę użytkownika. W przypadku poprawnego utworzenia struktury drzewiastej z naszych plików oraz pozytywnej odpowiedzi serwera, w termianlu pojawią się podstawowe opcje do wyboru.

Na początku należy wybrać użytkownika, naciskając enter przy odpowiedniej nazwie, z którym chcemy nawiązać połączenie (klawisz c). W przypadku sukcesu uzyskamy możliwość przeglądania zawartości plików wybranego użytkownika. Po wybraniu konkretnego pliku, w tle rozpocznie się pobieranie. Jeśli zakończy się sukcesem plik zostanie zapisany w folderze `Download/nazwa_uzytkownika` a informacja pojawi się w terminalu.

Nasze rozwiązanie

Teraz po krótce przedstawimy jak działa nasze rozwiązanie. Podzieliliśmy nasz projekt na kilka pakietów, zależało nam na komponentowanej strukturze. Moduły te komunikują się między sobą za pomocą kanałów. Postaralismy się również, żeby nasz program był jak najbardziej współbieżny, a co za tym idzie - szybki. Na większość poleceń od użytkownika, bądź żądania z zewnątrz jest tworzona osobna gorutyna - co pozwala na pobieranie kilku plików jednocześnie, obsługę kilku użytkowników. W przypadku gdy interlokutor[1] udostępnia kilka adresów, współbieżnie wysyłamy odpowiednie żądanie na każdy z nich, co również znacząco przyśpiesza działanie programu. Jesteśmy świadomi, że projekt ten opiera się na protokole UDP, dlatego zaimplenowaliśmy również politykę ponownego wysyłania. W kodzie umieściliśmy dużo logów, które są pomocne podczas debugowania. Można włączyć tę opcję ustawiając odpowiednią flagę. Natomiast podstawowe błędy bądź informacje niezbędne klientowi do korzystania z aplikacji są umieszczone w terminalu. Został również zaimplementowany mechanizm, który w przypadku błędu spowodowanego niepoprawnym żądaniem informuje oczekującego użytkownika stosownym komunikatem.

[1] interlokutor - rozmówca, w tym przypadku "peer"

merkle_tree

Moduł ten jest odpowiedzialny za stworzenie drzewiastej struktury naszych zasobów oraz przechowywanie dostępnych pozycji pozostałych użytkowników. Jesteśmy w stanie pobrać pliki od klientów, którzy udostępniają poprawne drzewa merkle tree. W ramach eksportowania możliwe jest przesyłanie plików i folderów o dowolnym rozmiarze. Struktury drzewa zdalnego i lokalnego są reprezentowane przez dwie odrębne struktury danych.

lokalne drzewo systemu plików

Drzewo reprezentujące pliki lokalne to mapa (hash → dane), która jest inicjalizowana na początku programu i przechowuje dane gotowe do wysłania. Całe drzewo plików lokalnych jest przechowywane w pamięci RAM.

zdalne drzewa systemu plików

Drzewa zdalne to mapy (hash → węzeł) wzbogacone o częściową weryfikację danych. Przez to, że drzewo jest niekompletne i budowane jest od korzenia, poprawość danych jest sprawdzana podczas aktualizowania wartości w drzewie. Początkowo każdy węzeł zdalny nie ma typu. Następnie może zostać wzbogacony o typ i dane, które przechowuje. Aby dobrze reprezentować jakiego typu wartości reprezentują węzły BIG, schodzimy po lewych synach do momentu aż nie napotkamy pierwszego węzła o typie DIRECTORY lub CHUNK. W momencie walidacji typu zakładamy, że reszta podrzewa zawieszonego w tym węźle też ma poprawny typ (np. w środku zweryfikowanego pliku nie ma folderu) - gdy jednak później napotkamy taką niespójność to poinformujemy użytkowanika o błędnych danych innego *zucha*[2] i zaprzestamy dalszej budowy drzewa. Drzewo zawsze pozostawione jest w stanie gdzie wszystkie hashe się zgadzają - niespójne aktualizacje są odrzucane. Pobieranie zdalnego pliku polega na ściągnięciu i zapisaniu całego poddrzewa pliku, a następnie przejście po gotowym poddrzewie i odczytanie oraz zapisanie bajtów w odpowiedniej kolejności. Implementacja pozwala na wielowątkowe pobieranie plików, jednak nie zaimplementowaliśmy congestion control, więc, zgodnie z treścią zadania, pobieranie odbywa się sekwencyjnie (liczba wątków ustawiona na 1). Przez bufor kanału, wielkość maksymalnego pliku jaki możemy pobrać tą metodą jest ograniczona, ale domyślnie jest ona też ograniczona przez fakt trzymania całego drzewa w pamięci - przy dobrym sprzęcie można zwiększyć maksymalny bufor. Dane pobrane od innych są przechowywane do momentu ręcznego wywołania odświeżenia przez użytkownika.

Ważna uwaga: struktura zakłada, że w jednym drzewie może być kilka węzłów o tym samym hashu, ale oznacza to, że będą one traktowane jak jeden węzeł - przykładowo, jeżeli użytkownik ma 5 identycznych, ale być może o innych nazwach, co do zawartości folderów, to wyświetlimy 5 identycznych folderów -

[2] peer

każdy o tej samej nazwie i zawartości. Pozwala to na obsługę *zuchów* posiadających ten sam plik w kilku miejscach, ale też redukuje zużycie pamięci, ponieważ powtarzające się węzły będą zapamiętane tylko raz.

message_manager, utility, peer_message_parser

Pakiety te pełnią przede wszystkim funkcje techniczne, związane z mapowaniem obiektów, ich inicjalizacją oraz udostępnianiem podstawowych operacji wykorzystywanych przez inne komponenty projektu. Ich rola to również odkodowanie otrzymanych wiadomości i zweryfikowanie ich poprawności. W przypadku wysyłania - zakodowanie danych według specyfikacji dostarczonego protokołu.

encryption

Moduł ten odpowiada za szyfrowanie, konwersję bajtów na odpowiednie klucze prywatne i publiczne oraz odwrotnie. Przy uruchomieniu programu, moduł ten sprawdza czy istnieje klucz prywatny w folderze *encryption* pod nazwą *private_key.pem* i w przypadku jego braku - generuje go i zapisuje. Większość dostarczonego kodu pochodzi z suplementu do opisu projektu.

tui

Służy do wyświetlania podstawowych informacji oraz odbierania poleceń użytkownika z terminala i odpowiednie reagowanie. Zostało zaimplementowane wyświetlanie struktury drzewiastej oraz podstawowa implementacja pamięci podręcznej[3] - pamiętamy z którymi użytkownikami nawiązaliśmy już połączenie, żeby przy każdym wybraniu użytkownika nie nawiązywać połączenia od nowa. Dla każdego użytkownika zapamiętywana jest aktualnie wyświetlana struktura oraz zawartość folderu. W przypadku ponownego wybrania tego samego katalogu, nie ma potrzeby ponownego wysyłania żądania o jego zawartość.

networking

packet_manager

Moduł zarządzający gniazdem UDP, jego celem jest wysyłanie i odbieranie pakietów. Składa się on z 3 komponentów: Sender (wysyłający), Receiver(odbierający) i Waiter(lokaj). Przy uruchamianiu zarządcy pakietów[4] możemy dostosować liczbę instancji pracowników każdego z komponentów. Parametry te ustawiamy w pliku *main.go*. Każdy komponent rozdziela pracę między swoich pracowników. Główną motywacją do wydzielenia tej części kodu do oddzielnego modułu było uproszczenie interfejsu do wysyłania wiadomości, z ewentualnym powtarzaniem, i odbierania odpowiedzi lub żądań od innych *zuchów*.

[3] cache
[4] packet_manager - nazwa tego modułu

wysyłający

Wysyłający odbiera ze swojego kanału prośby o wysłanie wiadomości, następnie realizuje to przy pomocy socketu UDP, a na końcu przekazuje informacje o wiadomości do lokaja. Jeżeli nie udało się wysłać wiadomości przez problemy z socketem wiadomość nie trafia do lokaja, lecz od razu przekazywana jest zlecającemu informacja o błędzie.

odbierający

Odbierający czyta dane, które dostajemy na socket UDP. Wstępnie parsuje otrzymane wiadomości, tak aby wiedzieć czy powinny one w ogóle trafiać do naszego systemu. Po weryfikacji i sprawdzeniu typu wiadomości przesyła komuniakt do dalszej obsługi, a informacje o odpowiedziach przesyła do lokaja.

lokaj

Mózg zarządcy pakietów - zajmuje się zbieraniem informacji o wysłanych pakietach i otrzymanych odpowiedziać. To on odpowiada za konrolowanie powtarzania żądań. Gdy upłynie czas wskazany przez politykę powtarzania dla danego żądania zostanie o tym poinformowany wysyłający. Gdy otrzymamy odpowiedź na jedno z naszych żądań lokaj poinformuje o tym zlecającego, a dane żądanie nie będzie już powtarzane.

src_conn

Moduł ten jest odpowiedzialny za połączenie z serwerem. Udostępnia funkcje, dzięki którym możemy pozyskać niezbędne informacje o użytkownikach. Uzyskuje informacje od serwera używając zapytań HTTP. Zostały dostarczone również testy jednostkowe.

handlers

Zadaniem tego modułu jest odbieranie przychodzących żądań oraz obsługa poleceń z TUI. W przypadku wysyłania danych - próbujemy dostarczyć informacje na wszystkie dostępne adresy. Wyświetlamy błąd wyłącznie w przypadku gdy żaden z udostępnionych adresów nie odpowiedział poprawnie. Wysyłanie na wszystkie adresy danego użytkownika zostało zaimplementowane, by wykonywało się równolegle przy pomocy gorutyn. W sytauacji kiedy przynajmniej jedno żądanie się uda - zwracamy od razu komunikat, nie czekając na pozostałe odpowiedzi. W przypadku otrzymania informacji zwrotnej, sprawdzamy poprawność zaszyfrowania danych. W przypadku otrzymania żądania z zewnątrz sprawdzamy również, czy jesteśmy połączeni z użytkownikiem - jeżeli nie to nie obsługujemy go w ogóle, chyba że jest to próba nawiązania połącznia. Gdy klient poprosi o dane zawsze w pierwszej kolejności próbujemy znaleźć je w pamięci podręcznej. W przypadku ich braku wysyłamy odpowiednie żądanie - pobieramy

tylko niezbędne pliki, a nie całe poddrzewo, co zapobiega przechowywaniu niepotrzebnych plików oraz przeciążania sieci. W sytaucji odswieżenia zawartości, usuwamy z pamięci podręcznej wszystkie informacje na temat plików użytkowników.

Nat traversal

Z pakietów NATTraversal korzystamy przy próbie połączenia z nowym użytkownikiem. Jeżeli nie uda nam się uzyskać odpowiedzi po kilku próbach, spróbujemy poprosić o pomoc jednego ze znanych nam *zuchów*, którzy mogą działać jako węzeł pośredni. Z góry zakładamy określoną liczbę prób przebicia się przez NAT i próbujemy przeprowadzić handshake ponownie. Nie oczekujemy na PING od drugiej strony. Jesteśmy świadomi, że stała liczba prób sprawia, że w dużej części przypadków potrzebne będzie mniej prób niż ta stała oraz że ten sposób może prowadzić do drobnych opóźnień (gdy wysyłamy kolejne prośby mimo przebicia NAT-u) lub nawet braku połączenia, gdy dużo pakietów się zgubiło. Brak czasu nie pozwolił na lepsze rozwiązania, a to rozwiązanie pozwala na komunikacje w zdecydowanej większości, a przynajmniej tak wynika z naszego doświadczenia, przypadków, a jeżeli nie uda się przebić NAT-u za pierwszym razem, zawsze można po prostu spróbować ponownie. Oczywiście nasz program implementuje również obsługę pakietów NATTraversal2, tak aby inni mogli od nas pobierać pliki gdy znajdujemy się za NAT-em.

Utrzymywanie połączeń

W celu utrzymania połączenia z serwerem jak i innymi użytkowniakmi co 20 sekund wysyłamy pakiet PING do każdego *zucha*, z którym zainicjowaliśmy połączenie. Brakuje jednak usuwania połączeń z zewnątrz, które są od dawna nieaktywne. Obecna struktura pozwala na dodanie tej funkcjonalności, ale niestety zabrakło nam czasu i sił aby to zaimplementować.

Podsumowanie projektu

Z naszego projektu najbardziej zadowoleni jesteśmy z implementacji pakietów **package-manager** oraz **handler**. Uważamy, że wykorzystaliśmy tutaj dużo elementów programowania współbieżnego. Dzięki umiejętnej implementacji funkcji pomocnicznych do dekodowania i odkodowywania kod jest przejrzysty. Podział *package-manager* na trzy komponenty, które komunikują się za pomoca kanałów, umożliwiło nam kontrolowanie polityki powtarzania. Dołączyliśmy również testy jednostkowe, które sprawdzają poprawność naszego rozwiązania. Jesteśmy świadomi, że wybierając mocno współbieżne rozwiązanie oraz tworzenie gorutyn na każde żądanie może być niebezpieczne, nawet jeśli ciało tych wątków jest proste. Stosując podstawową wersję *cache* w TUI oraz w handlerach zapewniamy, że korzystający użytkownik nie stworzy zbyt dużo procesów. Otrzymanie mnóstwa zapytań o dane od użytkownika z którym nawiązaliśmy handshake może spowodować, że nasz program przestanie działać. Nie zaimplementowaliśmy też

exponential backoff, ale nasz interfejs pozwala na łatwe i uporządkowane tworzenie różnych nowych polityk powtarzania żądań, w tym i exponential backoff. Nie wystarczyło jednak czasu aby przetestować tę implementację, tak aby mieć pewność, że jesteśmy z niej zadowoleni.

Pomoce z zewnątrz

W naszym projekcie korzystaliśmy z następujących paczek zewnętrznych:

- **Bubbletea** github - wykorzystaliśmy go jako interfejs tekstowy typu TUI. W naszej implementacji wzorowaliśmy się z udostępnionej dokumentacji.
- **Testify** github - Okazał się przydatny podczas testowania pojedynczych komponentów
- **Tint** github - poprawił czytelność logów i ułatwił wypisywanie zmiennych

Końcowe uwagi

W opisie projektu wspomnaino, że w strukturze danych wyróżniamy trzy typy: **Chunk**, **Directory**, **Big**. Zgodnie z opisem bit o wartości 3 odpowiada ostatniemu typowi. Zauważyliśmy jednak, że taka implementacja nie jest kompatybilna ze wzorcowym rozwiązaniem. Dlatego zdecydowaliśmy się traktować **Big** jako bajt o wartości 2.