PSI Dokumentacja

Bartosz Latosek

Grudzień 2022

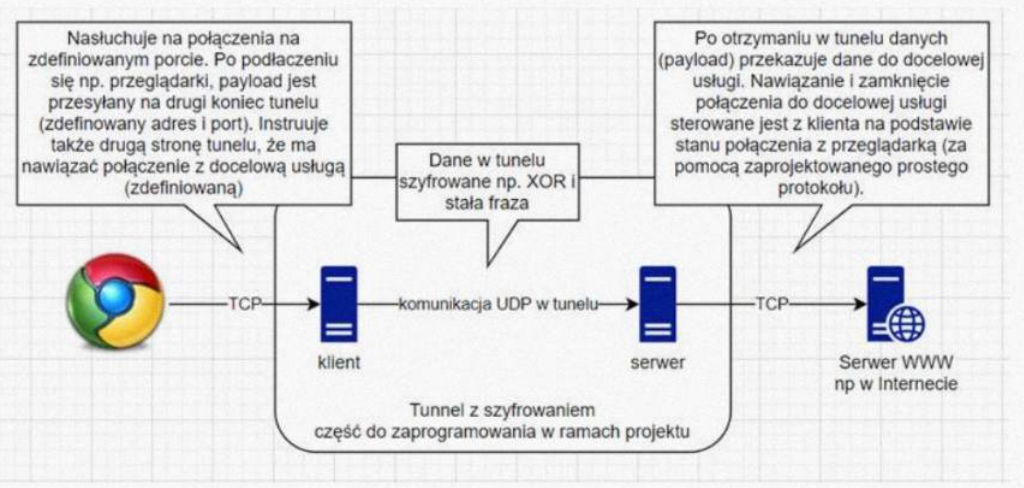
Spis treści

1 Treść zadania

Tunel UDP dla ruchu TCP

Klient łączy się do jednego końca tunelu (klient) na zdefiniowany port TCP. Dane z TCP będą przesyłane w połączeniu TCP do drugiego końca tunelu (serwer), który prześle dalej dane protokołem TCP do zdefiniowanego hosta. Porty klienckie/serwerowe, docelowa usługa TCP – konfigurowalna w prostym pliku konfiguracyjnym. Wystarczy tunelowanie pojedynczego portu TCP.

Tunel ma być przeźroczysty i uniwersalny, dla dowolnych protokołów aplikacyjnych – nie tylko HTTP. Należy także zaprojektować prosty protokół przesyłający stan połączenia pomiędzy końcami tunelu (główne stany TCP). Należy też pamiętać o ograniczaniu UDP na wielkość pakietów i dostosować obsługę połączenia TCP przez wykorzystanie bufora nie większego niż dozwolony w UDP. Możliwe, że trzeba będzie w UDP zrobić połączenie dwustronne – obie strony muszą nasłuchiwać na UDP, ponieważ TCP jest z natury dwustronny (full duplex) a UDP nie. Pewnie trzeba będzie użyć multipleksowania gniazd. Programy serwera i klienta wypisują na ekran log zdarzeń.



Rysunek 1: Schemat połączenia

Sugeruję zastosować ochronę poufności danych w tunelu, np. poprzez jakieś proste szyfrowanie danych (np. bitowy XOR ze stringiem (stałą wartością) będącą sekretem połączenia). Dlaczego takie szyfrowanie jest słabe? Proszę zastanowić się jak wygląda XOR tajnej wartości z ciągiem zer i pobawić się z XOR w *https://gchq.github.io/CyberChef/*

Dla uproszczenia można przyjąć, że jednocześnie może być obsługiwany jeden klient.

2 Zrozumienie zadania

Tunel UDP dla ruchu TCP, czyli tunneling UDP for TCP traffic, to technika polegająca na przesyłaniu danych protokołem TCP przez połączenie UDP. W takim tunelu, klient łączy się do jednego końca tunelu (klient) na zdefiniowanym porcie TCP, a dane z TCP są przesyłane w połączeniu UDP do drugiego końca tunelu (serwer). Serwer przekazuje dalej dane protokołem TCP do zdefiniowanego hosta. Porty klienckie/serwerowe oraz docelowa usługa TCP są konfigurowalne w prostym pliku konfiguracyjnym. Tunelowanie może obejmować pojedynczy port TCP.

Tunel UDP dla ruchu TCP jest często stosowany w celu obejścia ograniczeń sieciowych lub zapewnienia lepszej wydajności przesyłania danych. Może być również używany do przesyłania danych przez sieci o niskiej jakości lub niestabilnych połączeniach, ponieważ protokół UDP jest mniej wrażliwy na opóźnienia i utraty pakietów niż protokół TCP.

3 Opis funkcjonalności

1. Program umożliwia konfigurację portów klienckiego i serwerowego oraz docelowej usługi TCP.

2. Program umożliwia uruchomienie tunelu po naciśnięciu odpowiedniego przycisku lub polecenia z linii poleceń.

3. Po uruchomieniu tunelu, program nasłuchuje na porcie klienckim i oczekuje na połączenia od klientów.

4. Gdy klient łączy się z portem klienckim, program otwiera połączenie TCP z hostem docelowym przez port serwerowy.

5. Program przekazuje dane pomiędzy klientem a hostem docelowym poprzez połączenie TCP, używając protokołu UDP do przesyłania danych między końcami tunelu.

6. Program może również udostępniać interfejs graficzny lub polecenia z linii poleceń do konfiguracji i zarządzania tunelowaniem.

7. Program może zapewniać również funkcje logowania i monitorowania w celu ułatwienia diagnostyki i zarządzania tunelowaniem.

4 Opis i analiza poprawności stosowanych protokołów komunikacyjnych

4.1 Opis znanych protokołów wykorzystywanych w projekcie

1. Protokół TCP (Transmission Control Protocol): jest to protokół warstwy transportowej modelu OSI, który zapewnia niezawodne, kontrolowane przesyłanie danych między urządzeniami w sieci. TCP umożliwia wymianę danych w obu kierunkach i zapewnia spójność i poprawność przesyłanych danych poprzez mechanizmy kontroli błędów i retransmisji.

2. Protokół UDP (User Datagram Protocol): jest to protokół warstwy transportowej modelu OSI, który umożliwia przesyłanie danych w formie datagramów bez mechanizmów kontroli błędów i retransmisji. UDP jest często wybierany w sytuacjach, gdzie niezawodność przesyłania danych nie jest krytyczna, a priorytetem jest szybkość przesyłania danych.

3. Protokół HTTP (Hypertext Transfer Protocol): jest to protokół warstwy aplikacji, który umożliwia przesyłanie dokumentów hipertekstowych (np. stron internetowych) pomiędzy serwerem a klientem. HTTP jest używany przez przeglądarki internetowe do wymiany danych z serwerami WWW.

Protokół UDP jest mniej bezpieczny niż TCP, ponieważ nie zapewnia potwierdzenia odebrania danych przez odbiorcę. Może to prowadzić do utraty danych lub ich uszkodzenia podczas transmisji. Format komunikatów w protokole UDP jest prosty i niezależny, co oznacza, że każdy komunikat jest przesyłany osobno i nie jest zależny od poprzednich lub następnych komunikatów.

Protokół TCP jest bardziej bezpieczny niż UDP, ponieważ wymaga potwierdzenia odebrania danych przez odbiorcę. Format komunikatów w protokole TCP jest strukturalny, co oznacza, że komunikaty są grupowane w ramki i przesyłane w ramach połączenia. Protokół TCP jest również bardziej niezawodny niż UDP, ponieważ zapewnia potwierdzenie odebrania danych.

Oba protokoły różnią się również pod względem zależności czasowej. Protokół UDP nie posiada mechanizmów potwierdzających, co oznacza, że komunikaty nie są zależne od siebie i mogą być przesyłane w dowolnym momencie. Natomiast protokół TCP zapewnia zależność czasową pomiędzy komunikatami, ponieważ wymaga utworzenia połączenia i potwierdzenia odebrania danych przez odbiorcę.

4.2 Opis protokołu pomiędzy klientem a serwerem

W projekcie istotną rolę odegra synchronizacja stanu połączeń TCP pomiędzy klientem i serwerem. Ważne będzie również potwierdzenie odebrania pakietu i retransmisja wysyłanego pakietu w przypadku braku otrzymania potwierdzenia. W nagłówku pakietu należy więc umieścić liczbę symbolizującą stan połączenia TCP pomiędzy stronami tunelu oraz rosnący numer pakietów z danymi, aby usuwać duplikaty w razie utraty pakietu potwierdzającego. W przypadku dużych strat pakietów między serwerem a klientem, połączenie jest zamykane. Dodatkowo wielkość pakietów przesyłana pomiędzy klientem a serwerem zostanie ograniczona do 1024 mb.

Przykładowy format protokołu może mieć postać:

- Nagłówek:
  - Typ: wartość całkowita oznaczająca rodzaj wiadomości (np. nawiązywanie połączenia, przesyłanie danych, zrywanie połączenia).
  - Długość: wartość całkowita oznaczająca długość pola payload w bajtach.
- Payload: pole binarne zawierające faktyczne dane lub dodatkowe informacje potrzebne dla konkretnego rodzaju wiadomości.

Na przykład, jeśli pole typu jest ustawione na 0, może to oznaczać wiadomość nawiązywania połączenia. Pole payload mogłoby zawierać informacje takie jak adres IP i numer portu klienta, adres IP i numer portu docelowego hosta i inne niezbędne informacje potrzebne do nawiązania połączenia.

Jeśli pole typu jest ustawione na 1, może to oznaczać wiadomość przesyłania danych. Pole payload mogłoby zawierać dane przesyłane pomiędzy klientem a docelowym hostem.

Jeśli pole typu jest ustawione na 2, może to oznaczać wiadomość zrywania połączenia. Pole payload mogłoby być puste lub mogłoby zawierać dodatkowe informacje potrzebne do prawidłowego zamknięcia połączenia (np. potwierdzenie, że połączenie zostało zamknięte).

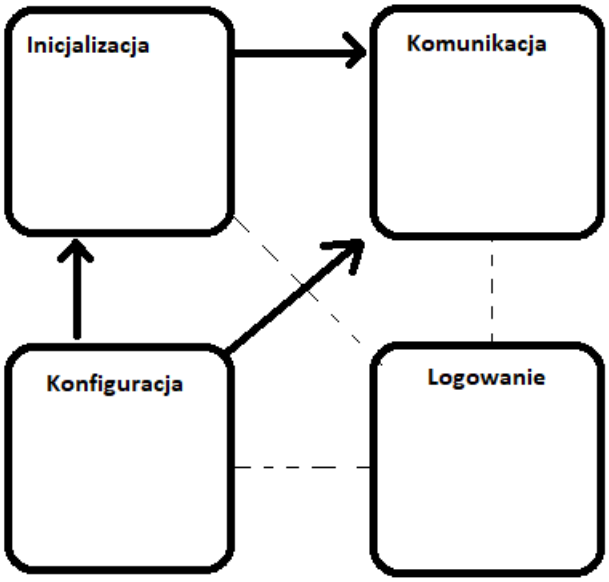
5 Podział na moduły

5.1 Moduły

1. Moduł inicjalizacji: odpowiada za inicjalizację socketów i podłączenie ich do odpowiednich portów.
2. Moduł komunikacji: odpowiada za wymianę danych pomiędzy klientem a serwerem za pomocą socketów i protokołu UDP.
3. Moduł konfiguracji: umożliwia konfigurację parametrów, takich jak porty klienckie/serwerowe i docelowa usługa TCP za pomocą prostego pliku konfiguracyjnego lub interfejsu graficznego.
4. Moduł logowania: odpowiada za rejestrowanie informacji o działaniu aplikacji, takich jak błędy i inne ważne informacje.

5.2 Komunikacja między modułami

1. Moduł inicjalizacji wysyła informacje o zainicjalizowaniu socketów do modułu komunikacji.
2. Moduł komunikacji odbiera dane od klienta i wysyła je do serwera za pomocą socketów i protokołu UDP. Następnie odbiera dane od serwera i wysyła je do klienta.
3. Moduł konfiguracji udostępnia interfejs do konfiguracji parametrów aplikacji, takich jak porty klienckie/serwerowe i docelowa usługa TCP.
4. Moduł logowania rejestruje informacje o działaniu aplikacji, takie jak błędy i inne ważne informacje.



Rysunek 2: Schemat połączeń między modułami

* W praktyce podział na moduły służy bardziej jako wzorzec, który w kodzie jest reprezentowany przez funkcjonalność końcowego programu. Nie ma wyraźnego podziału na moduł konfiguracyjny, komunikacyjny lub moduł logowania, ale w kodzie zaimplementowana jest obiecana funkcjonalność konfiguracji, komunikacji między węzłami i logowanie przepływu danych i akcji wewnętrznych.

6 Opis rozwiązania

Rozwiązanie projektu jest podzielone na 4 części:

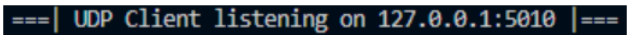
1. **tcp_client.py** - reprezentuje użytkownika, służy do wysyłania wiadomości do zdefiniowanego serwera TCP poprzez tunel UDP, oraz odbierania potwierdzeń otrzymania wiadomości.
2. **tcp_server.py** – odbiera i potwierdza otrzymanie wiadomości i odsyła do niego potwierdzenia, wiadomości otrzymuje poprzez tunel UDP reprezentowany przez klienta i serwer UDP
3. **udp_client.py** – reprezentuje początkowy węzeł tunelu UDP, komunikuje się z udp_server.py przekierowując ruch pomiędzy klientem a serwerem TCP. Razem z udp_server są odpowiedzialne za szyfrowanie ruchu przez tunel UDP.
4. **udp_server.py** – reprezentuje końcowy węzeł tunelu UDP komunikujący się ze zdefiniowanym serwerem TCP. Razem z udp_client są odpowiedzialne za szyfrowanie ruchu przez tunel UDP.

Przedstawiony system stanowi reprezentację tunelu UDP w protokole TCP. Obsługa błędów i komunikatów powoduje, że ruch jest dobrze zdefiniowany i odporny na błędy serwera. W celu uproszczenia projektu zakładamy, że nie występują błędy w komunikacji pomiędzy klientem a serwerem UDP, to znaczy zakładamy, że będą one działały poprawnie przed połączeniem klienta. Zostaną za to obsłużone błędy komunikacji pomiędzy klientem a serwerem TCP, tak jakby to miało miejsce przy faktycznej próbie połączenia.

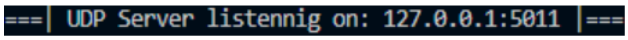
7 Opis interfejsu

7.1 Łączenie komponentów

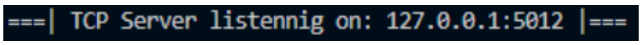
Aby uruchomić program należy w oddzielnych terminalach uruchomić pliki:



Rysunek 3: **udp_client.py**

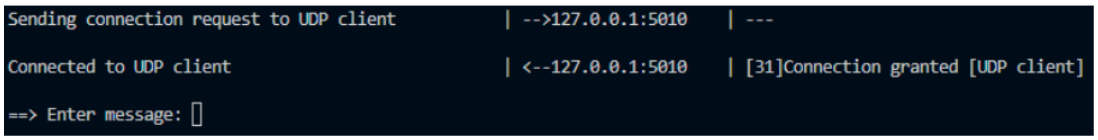


Rysunek 4: **udp_server.py**

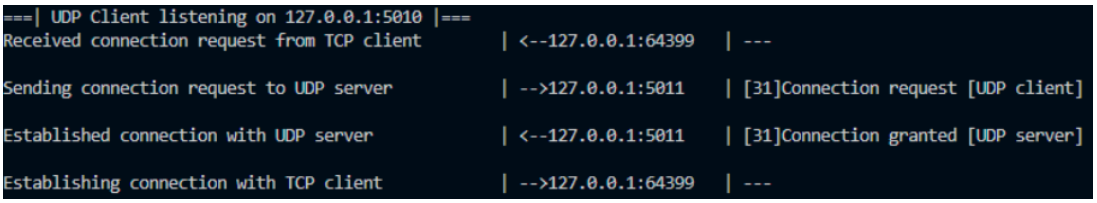


Rysunek 5: **tcp_server.py**

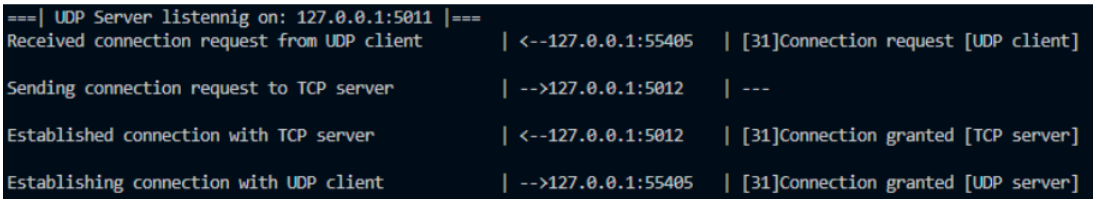
Po uruchomieniu tcp_client.py wszystkie komponenty zostają połączone:



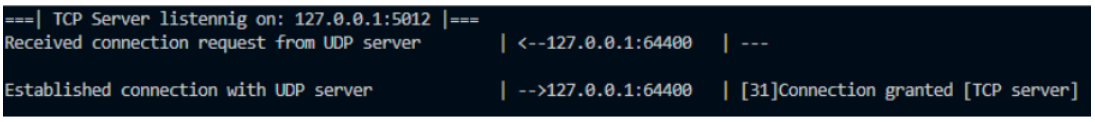
Rysunek 6: **Logi z kleinta TCP**



Rysunek 7: **Logi z klienta UDP**



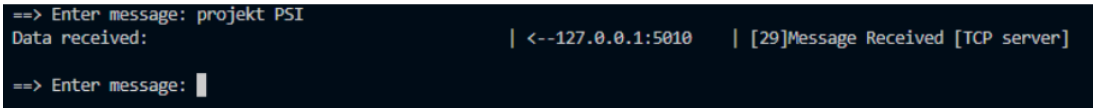
Rysunek 8: **Logi z serwera UDP**



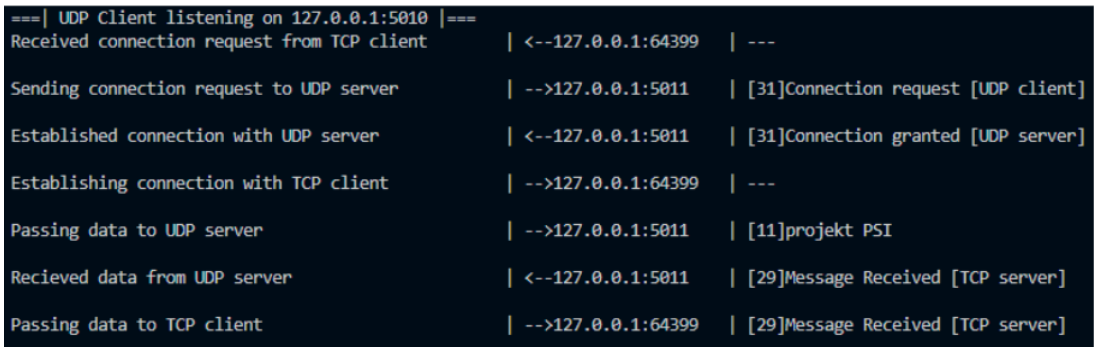
Rysunek 9: **Logi z serwera TCP**

7.2 Wysyłanie wiadomości

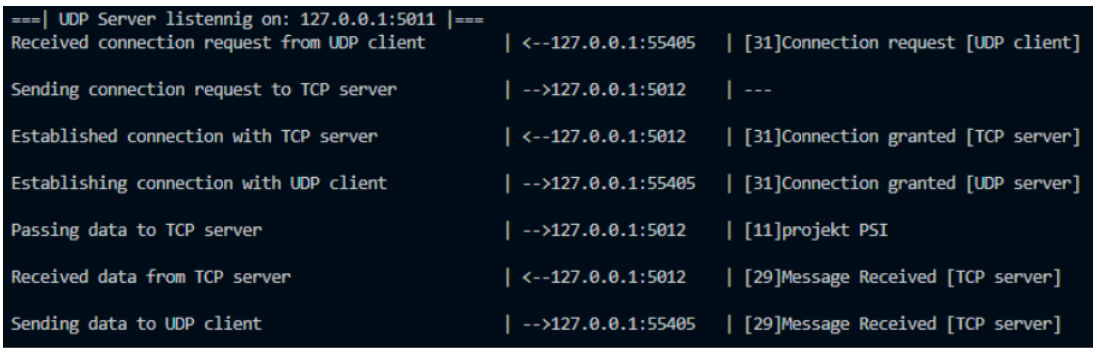
W tym momencie można wysyłać wiadomości z klienta TCP:



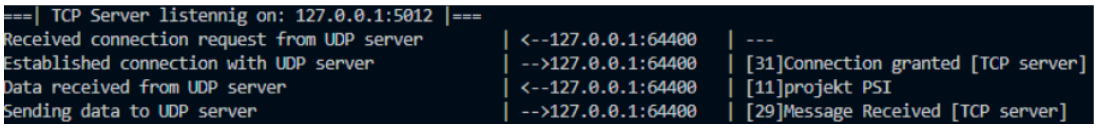
Rysunek 10: **Logi z kleinta TCP**



Rysunek 11: **Logi z klienta UDP**



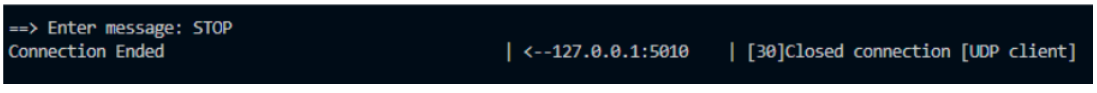
Rysunek 12: **Logi z serwera UDP**



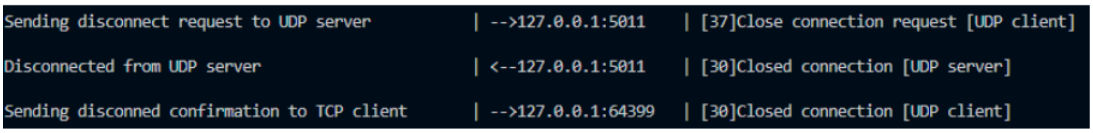
Rysunek 13: **Logi z serwera TCP**

7.3 Rozłączenie połączenia

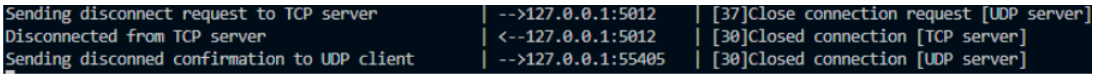
Po wysłaniu wiadomości STOP połączenie zostanie zerwane



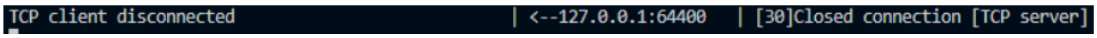
Rysunek 14: **Logi z kleinta TCP**



Rysunek 15: **Logi z klienta UDP**



Rysunek 16: **Logi z serwera UDP**

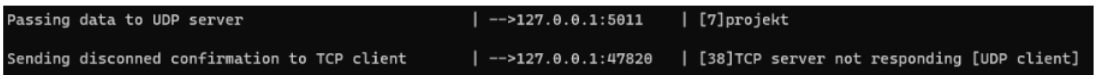


Rysunek 17: **Logi z serwera TCP**

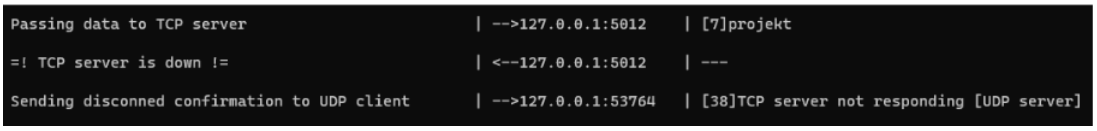
7.4 Nagłe zerwanie połączenia

Jeśli przerwiemy działanie serwera TCP oraz wyślemy wiadomość z klienta TCP to zostaniemy powiadomieni o tym przez komponenty UDP, a połączenie się zakończy:

Rysunek 18: **Logi z kleinta TCP**



Rysunek 19: **Logi z klienta UDP**

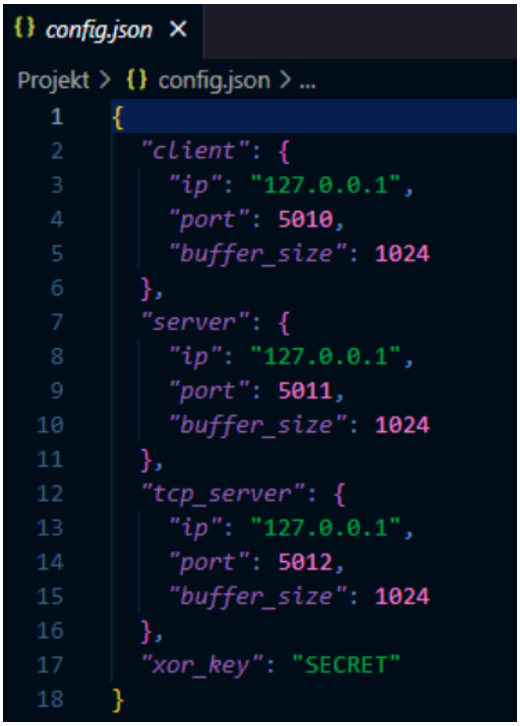


Rysunek 20: **Logi z serwera UDP**

8 Postać konfiguracji i logów

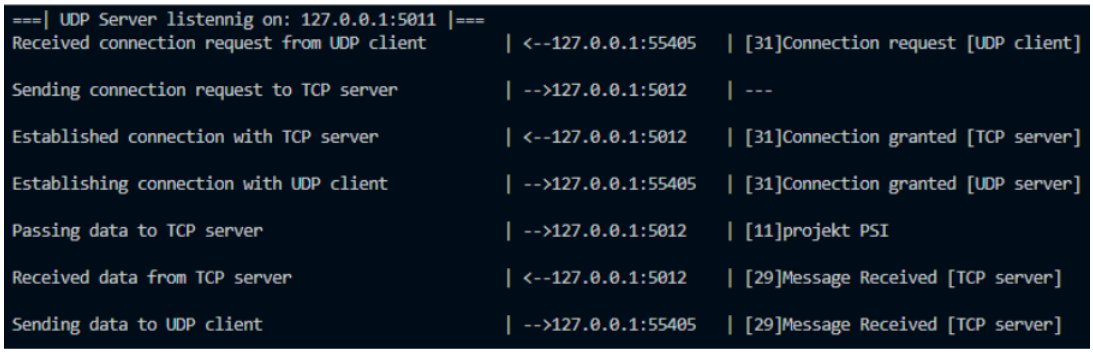
W projekcie znajduje się jeden plik konfiguracyjny config.json. Znajdują się w nim informacje dotyczące klienta i serwera UDP oraz serwera TCP, a także klucza do szyfrowania. Każdy uczestnik połączenia jest opisany:

- adresem IP
- numerem portu
- rozmiarem bufora



Rysunek 21: **config.json**

Program informuje na temat błędów oraz tego co się dzieje poprzez wypisywanie wiadomości w konsoli. Logi mają określony schemat, pierwsza kolumna zawsze zawiera opis akcji, druga kolumna zawiera adres, z którym jest powiązana akcja oraz strzałkę reprezentującą, czy akcja przychodzi z adresu, czy jest wysyłana na dany adres. Trzecia kolumna zawiera wiadomość związaną z akcją poprzedzoną jej długością wyrażoną w polu 'size' nagłówka wiadomości. Przykładowe logi:



Rysunek 22: **Logi z serwera UDP**

9 Wykorzystane narzędzia

Do tworzenia projektu wykorzystano język programowania Python oraz jego pakiety:

1. socket – do zarządzania gniazdami
2. select – do aktywnego oczekiwania na gniazdo
3. pickle – do przygotowywania wiadomości do wysłania, przeformatowanie na ciąg bajtów oraz odwrócenia tego procesu i przywrócenia wiadomości do stanu początkowego

4. json – do łatwej współpracy z formatem pliku konfiguracyjnego
5. os – do ustalenia ścieżek do pliku konfiguracyjnego

IDE służącym do edycji kodu było VisualStudioCode.

10 Opis scenariuszy testowych

1. Po włączeniu serwera TCP oraz tunelu UDP (udp_server, udp_client) a następnie połączeniu klienta TCP, klient TCP poprawnie łączy się ze zdefiniowanym serwerem TCP poprzez tunel UDP.
2. Przy sprawnym połączeniu między klientem TCP a serwerem TCP przez tunel UDP, przesyłane wiadomości przechodzą przez wszystkie węzły a do klienta TCP zwracana jest wiadomość z potwierdzeniem.
3. Wiadomości pomiędzy klientem UDP a serwerem UDP są szyfrowane i odszyfrowywane poprawnie.
4. Po włączeniu klienta i serwera UDP, ale bez włączenia serwera TCP i próbie połączenia klienta TCP z niedziałającym serwerem, połączenie nie zostaje nawiązane, a do klienta przesyłany jest odpowiedni komunikat.
5. Przy poprawnie nawiązanym połączeniu i później wyłączeniu serwera TCP, przy próbie wysłania wiadomości przez klienta TCP połączenie zostanie zerwane z odpowiednim komunikatem.

Zdecydowaliśmy się nie wklejać zrzutów ekranu potwierdzających działanie programu zgodnie z każdym scenariuszem w celu utrzymania zwięzłej kompozycji. Wszystkie scenariusze zostały przetestowane manualnie i mogą być w łatwy sposób zrekonstruowane.