

# Tunneling-and-transmission-analysis-system

TUNA

Wiktor Rogowski  
Paweł Ścibek  
Kacper Olszewski

AGH University of Krakow

May 2025

# Historia zmian

REV	DATA	ZMIANY		
1.1	26.01.2026	Wiktor Rogowski ski@student.agh.edu.pl		wiktrogow-
1.0	20.01.2026	Kacper Olszewski kolszewski@student.agh.edu.pl		kolszew-

# Spis treści

<b>Historia zmian</b>	<b>1</b>
<b>Lista oznaczeń</b>	<b>3</b>
<b>1 Wstęp</b>	<b>4</b>
1.1 Wymagania systemowe (requirements)	4
<b>2 Funkcjonalność</b>	<b>5</b>
<b>3 Analiza problemów</b>	<b>7</b>
3.1 Protokoły	7
3.2 Szyfrowanie	7
3.3 Parametry ruchu sieciowego	8
3.4 Sztuczne generowanie ruchu tła	9
<b>4 Projekt techniczny</b>	<b>10</b>
4.1 Opis struktury plików	10
4.2 Diagramy klas	10
<b>5 Opis realizacji</b>	<b>12</b>
5.1 Środowisko	12
5.2 Problemy i praca	12
<b>6 Opis wykonanych testów</b>	<b>13</b>
<b>7 Podział pracy</b>	<b>14</b>
7.1 Kacper Olszewski 35%:	14
7.2 Wiktor Rogowski 30%:	14
7.3 Paweł Ścibek 35%:	14
<b>8 Podręcznik użytkownika</b>	<b>16</b>

# **Lista oznaczeń**

<b>API</b>	round trip time
<b>RTT</b>	różnica pomiędzy n-tym, a (n-1)tym zmierzonym rtt
<b>jitter</b>	maximal transmission unit
<b>MTU</b>	Discrete-Time Fourier Transform
<b>OODB</b>	Object-Oriented database

# Rozdział 1

## Wstęp

Dokument dotyczy opracowania systemu serwer-klient, który pozwala symulować komunikację w różnych warunkach i na różnych protokołach. Do tego podłączona jest obiektowa baza danych (OODB), która umożliwia logowanie oraz weryfikację użytkowników.

### 1.1 Wymagania systemowe (requirements)

Podstawowe założenia projektu:

1. Przygotowanie modelu klas aplikacji od strony serwera i klienta.
2. Wybranie protokołów, szyfrowania, potrzebnych pomiarów
3. Testowanie programu na dwóch maszynach

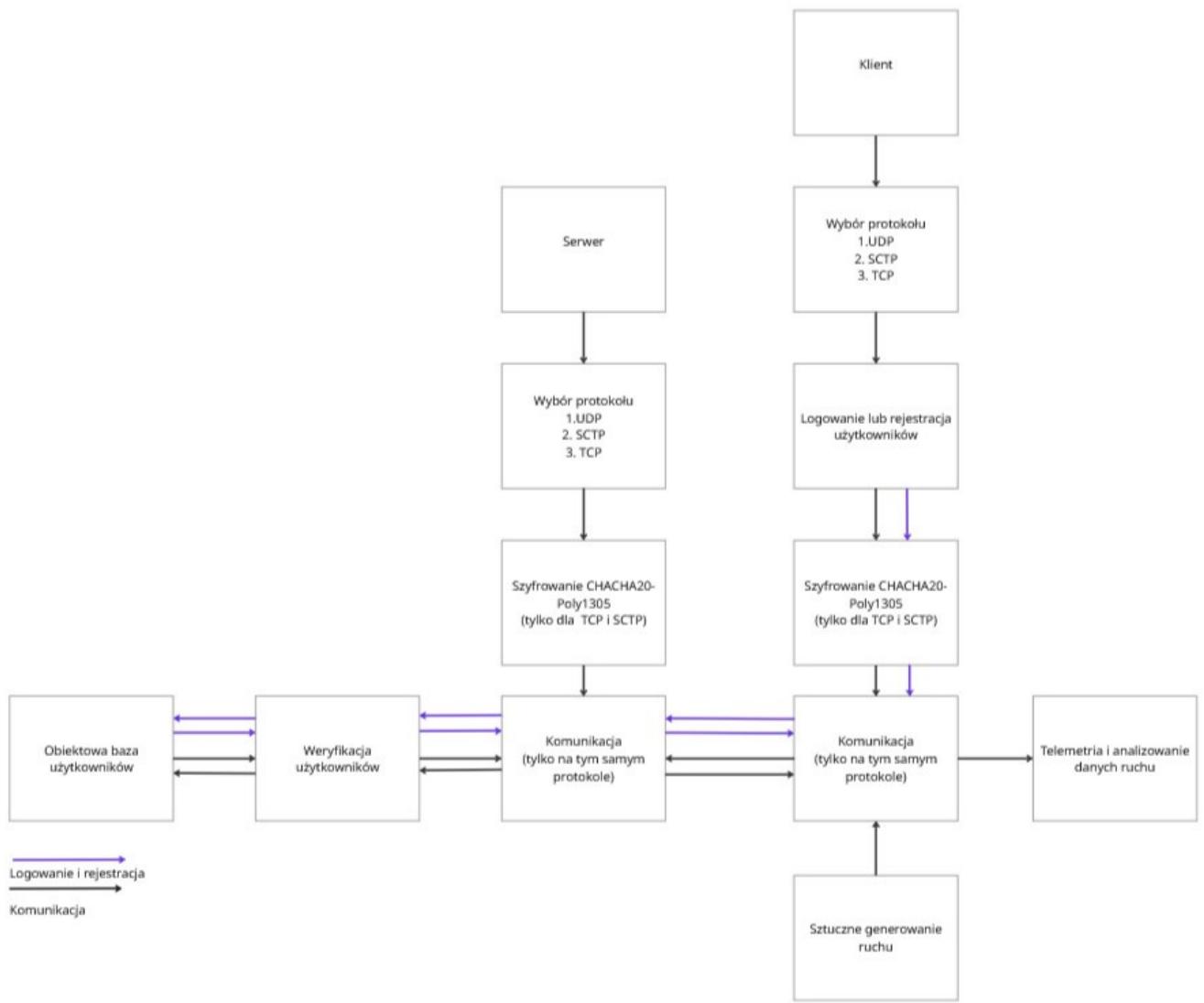
## Rozdział 2

### Funkcjonalność

Program zostaje uruchamiany na dwóch urządzeniach typu **Ubuntu 22.04 LTS** z kernellem **Linux 5.15**. Po jednej stronie serwer, a po drugiej klient. Przy uruchomieniu klienta należy podać do jakiego **ip** ma się podłączyć ( w tym przypadku serwera ) oraz wybrany protokół komunikacyjny. Przy serwerze wybiera się tylko protokół. Dokładne instrukcje są podane w pliku **README.md**.

Opis schematu:

1. Wybór protokołu - można wybrać spośród 3 protokołów : UDP, TCP i SCTP
2. Szyfrowanie (tylko SCTP i TCP) - na wiadomościach wykonywane jest szyfrowanie **CHACHA20**, nie udało się zaimplementować wyświetlania przykładowej wiadomości w terminalu
3. Występuje komunikacja i wiadomości wpisane w terminalu klienta pojawiają się po stronie serwera
4. Logowanie i rejestracja (od strony klienta) - po ustalonej komunikacji jest możliwość logowania się i rejestracji poprzez chat w terminalu
5. Weryfikacja do obiektowej bazy użytkowników (od strony serwera) - poprzez kanał komunikacyjny serwer odczytuje czy osoba próbuje się zalogować lub zarejestrować oraz sprawdza czy jest już w bazie danych
6. Telemetria ( od strony klienta ) - wykonywane są pomiary na podstawie wysyłanych ramek :
  - RTT - czas mierzony (za pomocą chrono steady clock) pomiędzy wysłaniem, a odebraniem wiadomości
  - jitter - różnica pomiędzy n-tym, a (n-1)tym zmierzonym rtt
  - MTU - największy datagram, który można przesyłać (bajty)
  - throughput - wszystkie bajty, które wysyłamy
  - goodput - = throughput - packet loss
  - packet loss - straty pakietów na wskutek jakiś niedociągnięć sieci
7. Sztuczne generowanie tuchu tła ( od strony klienta ) - żeby komunikacja była w rzeczywistych warunkach zrobiono sztuczny ruch tła który obniża poziom wydajności kanału komunikacyjnego



Rysunek 2.1: Schemat działania programu

# Rozdział 3

## Analiza problemów

### 3.1 Protokoły

Przy implementacji protokołów trzeba było dbać o to by nie wystąpiła fragmentacja, a bajty były przesyłane w dobrej kolejności.

### 3.2 Szyfrowanie

Żeby zaszyfrować dane użyto szyfrowania Chacha20:

1. Przygotowanie wiadomości do wysłania:
  - (a) Wybór plaintextu (dane do wysłania).
  - (b) Wygenerowanie losowego nonce (wektora inicjalizacyjnego).
2. Szyfrowanie wiadomości:
  - (a) Wykonanie XOR między plaintextem a strumieniem ChaCha20 generowanym z klucza i nonce.
  - (b) Dołączenie nonce do początku zaszyfrowanej wiadomości → ciphertext + nonce.
3. Odbiór wiadomości:
  - (a) Oddzielenie nonce od odebranej wiadomości.
  - (b) Odszyfrowanie danych przez XOR ciphertextu z tym samym strumieniem ChaCha20 generowanym z klucza i nonce.
  - (c) Uzyskanie oryginalnego plaintextu.

### 3.3 Parametry ruchu sieciowego

W powyższym projekcie uwzględniono pomiar następujących parametrów sieciowych:

1. RTT - parametr definiowany, jako minimalny czas wymagany do przesłania w obu kierunkach, od nadawcy (klient) do odbiorcy (serwer). W praktyce obliczane, jako czas upływający między przesaniem pakietów przez klienta, aż do otrzymania wiadomości zwrotnej od serwera.
2. Jitter - parametr definiowany, jako krótkookresowe odchylenie od ustalonych, określowych charakterystyk sygnału. W praktyce obliczane, jako różnica pomiędzy n-tą, a (n-1)-tą próbką zmierzonego RTT.
3. MTU - parametr definiowany, jako rozmiar największego datagramu (wyrażonego w bajtach), który dopuszczony jest do przesyłu. Jest on pobierany za pomocą funkcji getsockopt() poprzez jądro systemu (rozpatrując aktywny socket).
4. Packet Loss - parametr definiowany, jako brak otrzymania pełnej ilości pakietów przesłanych do danego źródła. Obliczany za pomocą specjalnych struktur:
  - (a) TCP - struktura tcp-info. Obliczenia za pomocą stosunku retransmisji do wszystkich przesyłanych pakietów.
  - (b) SCTP - struktura sctp-assoc-stats. Również obliczenia za pomocą stosunku retransmisji do wszystkich przesyłanych pakietów.
  - (c) UDP - obliczana ręcznie za pomocą stosunku 1 - ([otrzymane pakiety] / [przesłane pakiety]). W końcowym etapie pomnożona przez 100 dla otrzymania wyniku procentowego.
5. Throughput - parametr definiowany, jako przepustowość sieci, odnoszący się do szybkości przesyłania wiadomości kanałem komunikacyjnym w sieci komunikacyjnej. Jest to ilość wysyłanych danych (rozpatrując konkretny, n-ty pomiar) w stosunku do czasu ich przepływu (wyrażonego w sekundach). Końcową jednostką parametru są kilobity na sekundę [kbps].
6. Goodput - parametr definiowany, jako liczba użytecznych bitów dostarczanych przez sieć do określonego miejsca docelowego w jednostce czasu. Obliczana, jako różnica pomiędzy parametrem Throughput, a Packet Loss (cały przepływ pakietów - stracone pakiety). Końcową jednostką parametru są kilobity na sekundę [kbps].

## 3.4 Sztuczne generowanie ruchu tła

W celu przeprowadzenia miarodajnych testów obciążeniowych, konieczne było zaprojektowanie wydajnego mechanizmu zalewania sieci (*Traffic Flood*):

1. Problem nasycenia łącza:
    - (a) Niezależnie od medium transmisyjnego, kluczowym warunkiem pomiaru degradacji parametrów sieci jest doprowadzenie do rywalizacji o zasoby przepustowości.
  2. Konstrukcja generatora (*Traffic Generator*):
    - (a) Zastosowanie protokołu UDP pozwala na generowanie strumienia danych niezależnego od mechanizmów kontroli przepływu.
  3. Obsługa odbioru (*Traffic Sink*):
    - (a) Implementacja dedykowanego wątku na serwerze, odbierającego pakiety na osobnym porcie.
    - (b) Mechanizm ten zapobiega generowaniu zwrotnych komunikatów ICMP (*Port Unreachable*), które mogłyby sztucznie obciążać procesor serwera, niezależnie od tego, czy urządzenia połączone są kablem, czy działają na jednym hoście.
1. Zasada działania systemu
- Funkcjonowanie zrealizowanego analizatora sieciowego opiera się na sekwencyjnym przetwarzaniu żądań w architekturze klient-serwer:
- (a) Inicjalizacja i autoryzacja: Po stronie serwera inicjalizowany jest wątek „Traffic Sink” (port 9999) do utylizacji ruchu tła. Po zestawieniu połączenia (zgodnie ze specyfiką wybranego protokołu TCP/UDP/SCTP) następuje uwierzytelnienie klienta.
  - (b) Symulacja obciążenia: W celu weryfikacji stabilności łącza uruchamiany jest asynchroniczny generator ruchu tła. Wysyła on pakiety UDP w trybie ciągłym (*tight loop*) na port techniczny serwera, co wymusza rywalizację o zasoby sprzętowe (CPU, karta sieciowa) i pozwala na obserwację degradacji parametrów sieciowych w czasie rzeczywistym.

# Rozdział 4

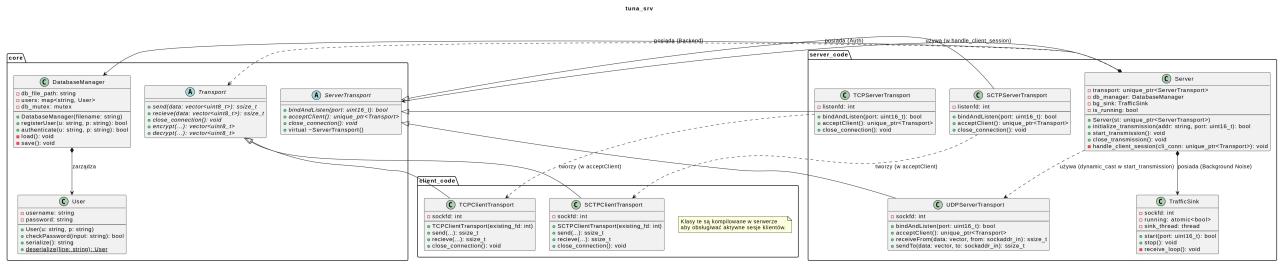
## Projekt techniczny

### 4.1 Opis struktury plików

Projekt został podzielony na dwie części: serwera `tuna_srv` i klienta `tuna_cli`, które odpowiednio są włączane po dwóch stronach.

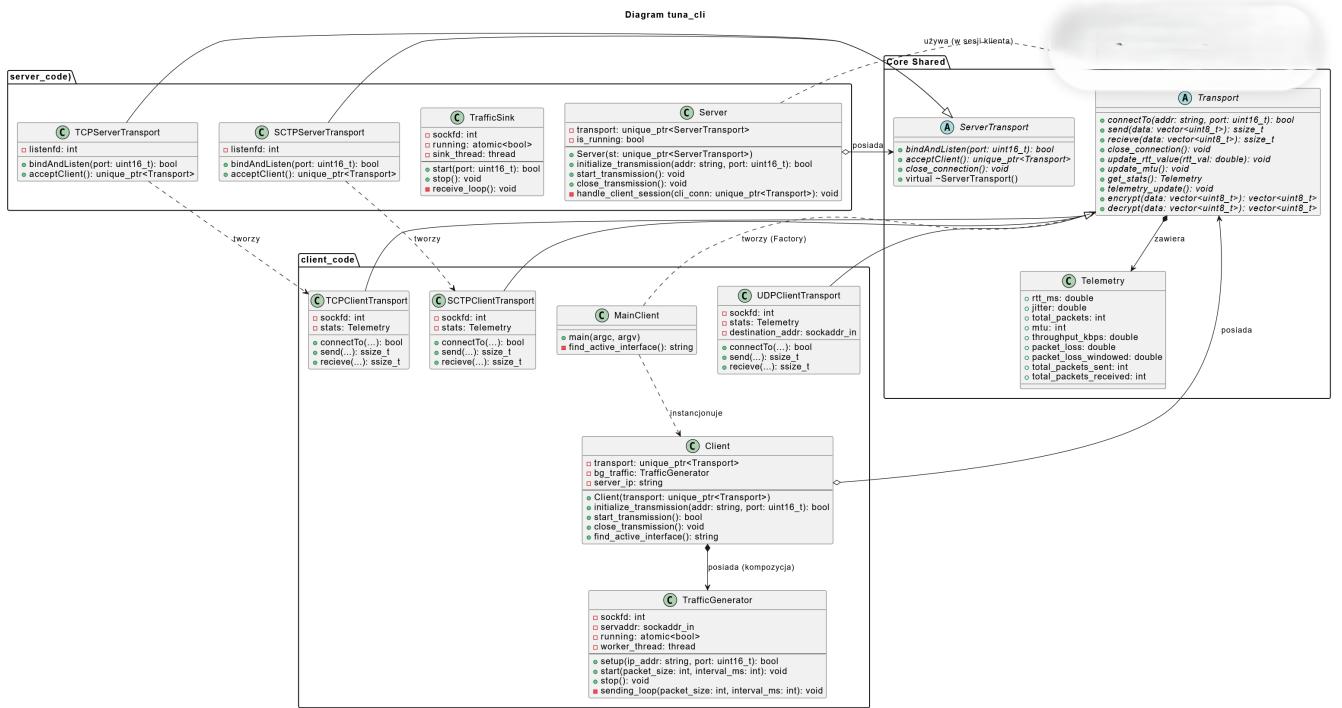
### 4.2 Diagramy klas

Diagram klas części serwera `tuna_srv`:



Rysunek 4.1: Diagram klas serwera

Diagram klas części klienta `tuna_cli`:



Rysunek 4.2: Diagram klas klienta

# Rozdział 5

## Opis realizacji

### 5.1 Środowisko

Do uruchamiania i testowania aplikacji użyto:

1. Dwóch maszyn wirtualnych z systemem **Ubuntu 24.04.3 LTS**, potrzebna była powłoka linuxowa do komunikacji, której używaliśmy na Programowaniu sieciowym.
2. Program budowano za pomocą CMake (plik `Makefile`) oraz komplikowano używając g++ od razu potrzebnymi bibliotekami.
3. Działanie programu odbywa się w terminalu, wiadomości jak i statystyki

### 5.2 Problemy i praca

Dużym problemem był czas przy sprawdzaniu działania, ponieważ zmiany trzeba było za każdym razem aktualizować na maszynach. Kod dopracowywany był na zwykłym komputerze klasy PC ponieważ na maszynach działanie było powolne przez mniejsze ilości RAM'u.

Kolejnym problemem było synchronizowanie zmian na githubie. I strona serwera i strona klienta potrzebowały nawzajem niektórych swoich części, a zarazem miały czasami mniej użytych klas przez co nie można było po prostu skopiować jednego `client_code` z `tuna_cli` do `tuna_srv`.

Przy pracy nad projektem były 3 osoby i wystąpiły problemy takie jak jednoczesne mergowanie. Przy jednej sytuacji dwie osoby pracowały jednocześnie nad zmianami i nie dało się rozwiązać tego konfliktami. Powstał branch MergeAttempt i była potrzeba ręcznego łączenia wszystkiego.

## Rozdział 6

### Opis wykonanych testów

Nie wykonano połączenia z gtest, jak wcześniej wspomniano dużym problemem było testowanie programu na dwóch maszynach oraz nie było to wykonane w Microsoft Visual Studio.

UDP	Dla UDP jest potrzeba podwójnego logowania
TCP	RTT i jitter są duże z niewiadomych powodów

# Rozdział 7

## Podział pracy

Niektóre części kodu nie zostały zcommitowane przez odpowiednie osoby. Było dużo komplikacji przy mergowaniu przez co czasami czyś kod był wysłany przed inną osobę.

### 7.1 Kacper Olszewski 35%:

1. Protokół SCTP
2. Generowanie ruchu tła
3. Pojedyncze testy i merge
4. Część serwera bazy obiektowej
5. Dokumentacja raportu

### 7.2 Wiktor Rogowski 30%:

1. Protokół UDP
2. Szyfrowanie
3. Opracowanie raportu
4. Pojedyncze testy i merge
5. Część klienta bazy obiektowej

### 7.3 Paweł Ścibek 35%:

1. Zwierzchnictwo, wytyczanie kierunku oraz ram projektu
2. Utworzenie struktury projektu
3. Implementacja pomiaru parametrów sieciowych dla każdego protokołu
4. Implementacja protokołu TCP
5. Pojedyncze testy i merge

6. Opracowanie konspektu

# Rozdział 8

## Podręcznik użytkownika

Szczegóły przedstawiono w README.md w repozytorium zdalnym na githubie

# Bibliografia

- [1] User Datagram Protocol [online]. wikipedia : wolna encyklopedia, 2025-09-03 06:06Z. [dostęp: 2026-01-26 14:53Z]. Dostępny w Internecie: [//pl.wikipedia.org/wiki/UserDatagramProtocol?oldid = 77528412](https://pl.wikipedia.org/wiki/UserDatagramProtocol?oldid=77528412).
- [2] Stream Control Transmission Protocol [online]. wikipedia : wolna encyklopedia, 2025-10-19 10:48Z. [dostęp: 2026-01-26 14:52Z]. Dostępny w Internecie: [//pl.wikipedia.org/wiki/StreamControlTransmissionProtocol?oldid = 77846183](https://pl.wikipedia.org/wiki/StreamControlTransmissionProtocol?oldid=77846183).
- [3] Wikipedia contributors. Chacha20-poly1305 — Wikipedia, the free encyclopedia, 2026. [Online; accessed 26-January-2026].
- [4] Wikipedia contributors. Transmission control protocol — Wikipedia, the free encyclopedia, 2026. [Online; accessed 26-January-2026].