

UNIwersytet Rzeszowski
Wydział Nauk Ścisłych i Technicznych
Instytut Informatyki



Krzysztof Filipczak
134907

Informatyka

Komunikator „Koliber”

Praca projektowa

Praca wykonana pod kierunkiem
mgr inż. Jarosław Szkoła

Rzeszów 2026

Spis treści

1. Wprowadzenie	6
2. Opis założeń projektu	7
2.1. Wymagania funkcjonalne	7
2.2. Wymagania нефункционалне	8
3. Opis struktury projektu	9
3.1. Struktura plików	9
3.2. Najważniejsze metody	9
3.2.1. Plik server.py	9
3.2.2. Plik client.py	10
4. Prezentacja działania i testy	12
4.1. Serwer Ubuntu: komunikaty rozpoczęcia pracy i dołączenia	12
4.2. Serwer Ubuntu: komunikaty rozłączania się użytkowników	12
4.3. Działanie aplikacji z perspektywy maszyn klienckich	13
4.3.1. Przesyłanie wiadomości i komunikat dołączenia	13
4.3.2. Rozłączenie się - komunikat	13
5. Instrukcja uruchomieniowa	14
6. Podsumowanie	15
Bibliografia	16
Spis rysunków	17

1. Wprowadzenie

Projekt prostego, anonimowego komunikatora sieciowego został zrealizowany na potrzeby przedmiotu **Sieci Komputerowe** na 2 roku Informatyki na **Uniwersytecie Rzeszowskim**. Głównym celem pracy było zaprojektowanie i implementacja systemu wielowątkowego wymiany informacji w architekturze **Klient-Serwer-Klient**. System pozwala na równoczesną komunikację wielu jednostek klienckich za pośrednictwem centralnego węzła pośredniczącego, którym w naszym przypadku jest serwer **Ubuntu**. Zarządza on przepływem danych w izolowanym środowisku sieciowym na potrzeby symulacji realnej sieci.

The project of a simple, anonymous network messenger was carried out for the purposes of the **Computer Networks** course in the second year of Computer Science at the **University of Rzeszów**. The main goal of the work was to design and implement a multithreaded information exchange system in the **Client-Server-Client** architecture. The system allows for simultaneous communication between multiple client units via a central intermediary node, which in our case is an Ubuntu server. It manages data flow in an isolated network environment for the purposes of simulating a real network.

2. Opis założeń projektu

Celem projektu było stworzenie prostego i anonimowego komunikatora sieciowego przy użyciu języka programowania **Python**. Program ma za zadanie umożliwić wielu użytkownikom jednoczesną wymianę wiadomości tekstowych w czasie rzeczywistym, wykorzystując architekturę wielowątkową również zaimplementowaną używając wspomnianego wcześniej języka **Python** dobierając wcześniej odpowiednie moduły takie jak `threading` czy `socket`. Program ma działać w środowisku rozproszonym, łącząc maszyny o różnych systemach operacyjnych pracując wewnątrz wirtualnej infrastruktury sieciowej.

Podstawowym problemem, który zostanie rozwiązany przez realizację tego projektu, jest zagrożenie prywatności w sieci oraz konieczność udostępniania danych osobowych w tradycyjnych systemach komunikacji. Obecnie wiele krajów zachodnich takich jak chociażby *Wielka Brytania* coraz częściej z wrogością zwraca się do prywatnej wymiany informacji w obawie o to iż użytkownicy będą komunikować między sobą informacje nie będące po drodze narracji rządowej. Projekt inspirowany forum obrazkowym *4chan* rozwiązuje ten problem poprzez brak konieczności zakładania kont oraz wykorzystanie tymczasowych identyfikatorów sesji, które są generowane losowo przez serwer i usuwane natychmiast po rozłączeniu użytkownika z pamięci operacyjnej węzła.

Aby problem został skutecznie rozwiązany, potencjalny zespół musi posiadać wiedzę z zakresu architektury sieciowej **TCP/IP**, modelu **ISO/OSI** (w szczególności warstwy transportowej i aplikacji) oraz podstawowej obsługi wątków.

Rozwiązanie problemu przebiegało w kilku zdefiniowanych krokach. W pierwszej kolejności nastąpiło zaprojektowanie topologii sieciowej w środowisku `VirtualBox`. Kolejnym krokiem było zaimplementowanie logicznej warstwy serwera, odpowiedzialnego za akceptowanie połączeń i przesyłanie komunikatów. Następnie wdrożono obsługę wielu wątków oraz system anonimowego przydzielania identyfikatorów. Końcowym etapem było stworzenie podstawowego interfejsu **CLI** dla klientów, gdzie Ci będą mogli odróżniać wiadomości swoje od cudzych za pośrednictwem kolorów.

2.1. Wymagania funkcjonalne

- **Nawiązywanie połączenia:** Użytkownik musi mieć możliwość połączenia się z serwerem za pomocą protokołu **TCP/IP** po podaniu adresu IP serwera oraz zdefiniowanego portu (55555) - poziom skryptu `client.py`.
- **Automatyczne przydzielenie tożsamości:** System musi wygenerować i przypisać każdemu użytkownikowi unikalny, 6-znakowy identyfikator sesji - ID natychmiast po nawiązaniu połączenia przez użytkownika.
- **Wymiana wiadomości w czasie rzeczywistym:** Wiadomość wpisana przez jednego użytkownika musi zostać natychmiast rozesłana przez serwer do wszystkich pozostałych aktywnych uczestników sesji.
- **Identyfikacja nadawcy:** Każda wiadomość przesłana w sieci musi być opatrzona identyfikatorem ID nadawcy, aby odbiorcy wiedzieli, od kogo pochodzi wiadomość.
- **Powiadomienia systemowe:** Program musi informować wszystkich uczestników o zmianach w strukturze sieci (dołączenia nowego użytkownika albo opuszczenie przez niego sesji).

- **Wizualne odróżnienie komunikatów:** Interfejs musi za pomocą kolorów odróżniać własne wiadomości od wiadomości innych użytkowników oraz komunikatów systemowych.
- **Automatyczne sprzątanie sesji:** Serwer musi automatycznie wykrywać rozłączanie klienta, zamykać powiązany z nim socket i zwalniać zajęte ID do ponownej puli losowania.

2.2. Wymagania нефunkcjonalne

- **Niezawodność transmisji:** Komunikacja musi opierać się na protokole TCP, aby zagwarantować, że wiadomości dotrą do odbiorców w całości i w poprawnej kolejności.
- **Wielowątkowość:** Serwer musi być zdolny do obsługi wielu gniazd sieciowych (socketów) jednocześnie bez blokowania głównego procesu aplikacji.
- **Wieloplatformowość:** Kod musi być kompatybilny z różnymi systemami operacyjnymi począwszy od Linux aż po systemy z rodziny Microsoft Windows.
- **Minimalizm:** Oprogramowanie ma działać wyłącznie w trybie tekstowym CLI minimalizując tym samym zużycie pamięci RAM i CPU.
- **Brak śladu cyfrowego:** Wszystkie dane o użytkownikach i treści wiadomości muszą być przechowywane wyłącznie w pamięci RAM. Brak logowania danych na dysku twardym serwera zapewnia anonimowość i prywatność.
- **Izolacja środowiska:** System musi poprawnie pracować wewnątrz wirtualnej sieci (VirtualBox).
- **Samowystarczalność:** Projekt musi być napisany w czystym języku **Python** przy użyciu wyłącznie bibliotek standardowych, co eliminuje konieczność instalowania zewnętrznych bibliotek i zależności.

3. Opis struktury projektu

3.1. Struktura plików

Projekt został zorganizowany w sposób modułowy, oddzielając kod źródłowy od plików dokumentacji. Poniżej przedstawiona została struktura katalogów w formie drzewa:

```
koliber/
├── documentation/ ..... Dokumentacja techniczna projektu
└── src/ ..... Katalog z kodem źródłowym
    ├── client.py ..... Aplikacja kliencka
    └── server.py ..... Skrypt serwera
```

3.2. Najważniejsze metody

3.2.1. Plik server.py

Listing 3.1. Metoda generująca unikalne ID sesji

```
def generate_id():
    pool = string.ascii_uppercase + string.digits
    id_result = ""
    for i in range(6):
        char = random.choice(pool)
        id_result = id_result + char
    return id_result
```

Listing 3.2. Zarządzanie sesją i procesem Handshake

```
def handle_client(user, client_address):
    session_id = generate_id()
    while session_id in assigned_id:
        session_id = generate_id()

    assigned_id.append(session_id)

    user.send(("SET_ID:" + session_id).encode('utf-8'))

    handle_broadcast(("Uzytkownik_o_ID_" + session_id + "_dolaczyl.").encode('utf-8'), user)
    print("[UBUNTU]:_Adres:_ " + str(client_address) + "_ID:_ " + session_id + "_dolaczyl._")

    while True:
        try:
            data = user.recv(1024)
            if not data:
                break

            text = data.decode('utf-8')
            full_message = "Anon#" + session_id + ":_ " + text
            handle_broadcast(full_message.encode('utf-8'), user)
```

```

except Exception as e:
    print("ERROR:_uzytkownik_" + session_id + "_" + str(e))
    break

print("[UBUNTU]:_ID:" + session_id + "_sie_rozlaczyl._")

if session_id in assigned_id:
    assigned_id.remove(session_id)
if user in user_sockets:
    user_sockets.remove(user)

handle_broadcast(("Uzytkownik_o_ID_" + session_id + "_wyszedl.").encode('utf-8'), user)
user.close()

```

3.2.2. Plik client.py

Listing 3.3. Asynchroniczny wątek odbiorczy z obsługą komunikatów systemowych

```

def handle_messages():
    global my_id
    while True:
        try:
            received_bytes = client.recv(1024)
            if not received_bytes:
                break

            message = received_bytes.decode('utf-8')

            if message.startswith("SET_ID:"):
                my_id = message.split(":")[1]
            elif "dolaczyl" in message:
                print("\r" + CLEAR + YELLOW + BOLD + message + RESET)
                print(GREEN + "TY_(ID-" + my_id + "):_" + RESET, end="", flush=True)
            elif "wyszedl" in message:
                print("\r" + CLEAR + RED + BOLD + message + RESET)
                print(GREEN + "TY_(ID-" + my_id + "):_" + RESET, end="", flush=True)
            else:
                print("\r" + CLEAR + message)
                print(GREEN + "TY_(ID-" + my_id + "):_" + RESET, end="", flush=True)

        except Exception as e:
            print("ERROR:_odebranie_wiadomosci" + str(e))
            break

```

Listing 3.4. Pętla wysyłania z mechanizmem nadpisywania wejścia użytkownika

```

while True:
    try:
        current_prompt = GREEN + "TY_(ID-" + my_id + "):_" + RESET
        msg_to_send = input(current_prompt)

        if msg_to_send:
            print(UP + "\r" + CLEAR + GREEN + "TY_(ID-" + my_id + "):_" + msg_to_send + RESET)
            client.send(msg_to_send.encode('utf-8'))

    except Exception as e:

```

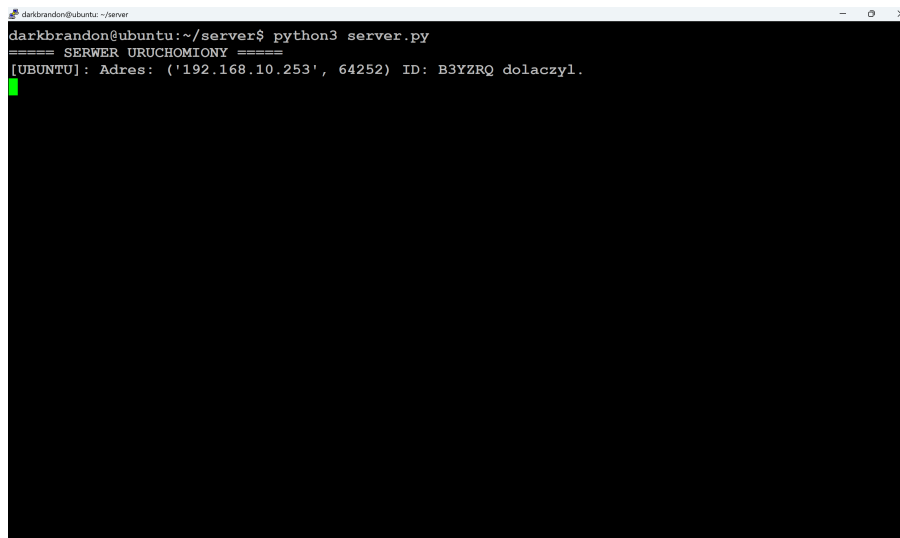


```
print("ERROR:_blad_wyslania_" + str(e))  
break
```

4. Prezentacja działania i testy

4.1. Serwer Ubuntu: komunikaty rozpoczęcia pracy i dołączenia

Działanie serwera wraz z komunikatami o dołączeniu się użytkowników są przedstawione na obrazku poniżej - rys. 4.1

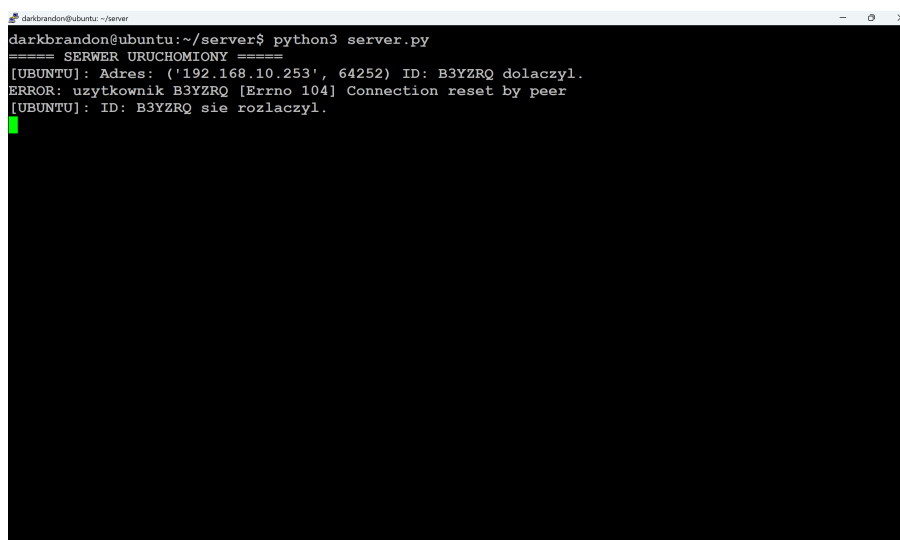


```
darkbrandon@ubuntu: ~/server
darkbrandon@ubuntu:~/server$ python3 server.py
===== SERWER URUCHOMIONY =====
[UBUNTU]: Adres: ('192.168.10.253', 64252) ID: B3YZRQ dołączył.
```

Rys. 4.1. Działanie serwera i komunikat dołączenia użytkownika

4.2. Serwer Ubuntu: komunikaty rozłączania się użytkowników

Działanie serwera wraz z komunikatami o dołączeniu się użytkowników i ich następnym rozłączeniu są przedstawione na obrazku poniżej - rys. 4.2



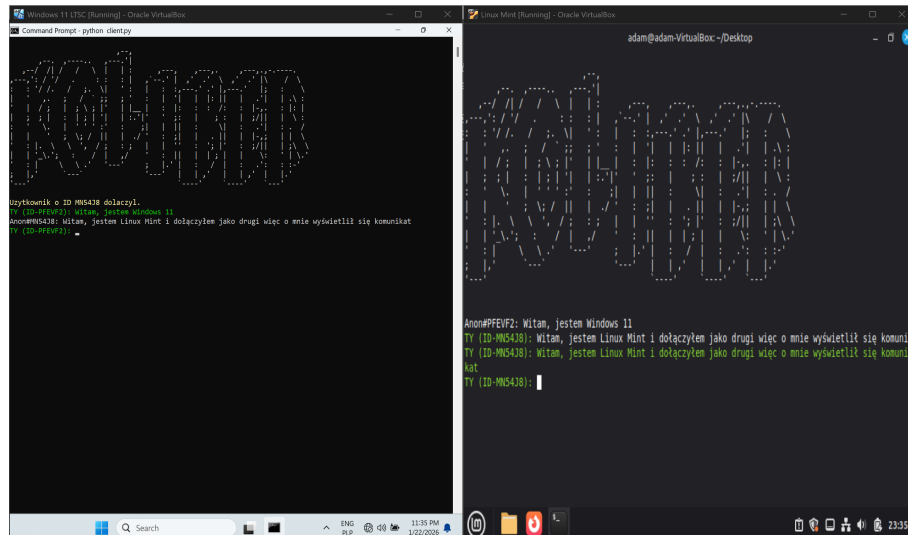
```
darkbrandon@ubuntu: ~/server
darkbrandon@ubuntu:~/server$ python3 server.py
===== SERWER URUCHOMIONY =====
[UBUNTU]: Adres: ('192.168.10.253', 64252) ID: B3YZRQ dołączył.
ERROR: użytkownik B3YZRQ [Errno 104] Connection reset by peer
[UBUNTU]: ID: B3YZRQ się rozłączył.
```

Rys. 4.2. Rozłączenie użytkownika - perspektywa serwera

4.3. Działanie aplikacji z perspektywy maszyn klienckich

4.3.1. Przesyłanie wiadomości i komunikat dołączenia

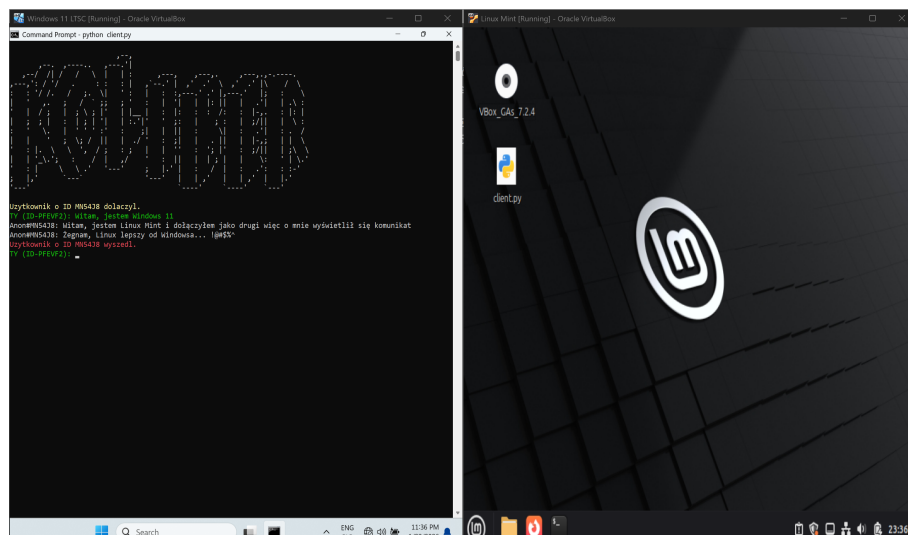
Działanie aplikacji po stronie użytkowników na maszynach Windows 11 LTSC i Linux Mint, gdzie widoczne jest przesyłanie wiadomości oraz komunikat o dołączeniu użytkownika widoczne jest na rysunku poniżej - rys. 4.3



Rys. 4.3. Przekazywanie wiadomości i dołączenie użytkownika

4.3.2. Rozłączenie się - komunikat

Komunikat informujący o rozłączeniu się jednego z użytkowników widoczny jest na obrazku poniżej - rys. 4.4



Rys. 4.4. Rozłączenie użytkownika - komunikat

5. Instrukcja uruchomieniowa

Aby poprawnie uruchomić skrypty a co za tym idzie kompletną aplikację, należy przeprowadzić następujące kroki:

1. Zainstalować środowisko **VirtualBox**, najlepiej w najnowszej wersji.
2. Utworzyć maszynę wirtualną - Ubuntu Server. W tym przypadku wersja Ubuntu 24.04.3 LTS.
3. Utworzyć klienckie maszyny - w przypadku projektu Windows 11 LTSC oraz Linux Mint.
4. Plik `client.py` przesłać do maszyn klienckich np. na pulpit.
5. Plik `server.py` przesłać na serwer Ubuntu.
6. Plik `server.py` uruchomić na serwerze poprzez komendę `python3 server.py` - po wykonaniu komendy powinien przywitać nas odpowiedni komunikat.
7. Plik `client.py` uruchomić poprzez komendę `python client.py` (Windows) / `python3 client.py` (Linux) uprzednio przechodząc w odpowiednią ścieżkę w terminalu - CMD / Bash.
8. Po wykonaniu wcześniejszych kroków całość powinna działać poprawnie.

6. Podsumowanie

Zrealizowany projekt pozwolił na stworzenie w pełni funkcjonalnego, wielowątkowego komunikatora sieciowego. Wszystkie założenia oraz cele zostały w pełni osiągnięte. Podczas prac szczególną uwagę poświęcono stabilności połączenia oraz czytelności "GUI" w terminalu poprzez kolory odróżniające komunikaty. W przyszłości można dodać bardziej rozbudowany interfejs użytkownika poprzez odpowiednie moduły oraz opcję przesyłania plików między użytkownikami.

Bibliografia

- [1] NeuralNine, 2026. <https://www.youtube.com/@NeuralNine>.
- [2] Python, 2026. <https://docs.python.org/3/library/socket.html>.
- [3] Stack Overflow, 2026. <https://stackoverflow.com/>.
- [4] W3Schools, 2026. <https://www.w3schools.com/>.

Spis rysunków

4.1	Działanie serwera i komunikat dołączenia użytkownika	12
4.2	Rozłączenie użytkownika - perspektywa serwera	12
4.3	Przekazywanie wiadomości i dołączenie użytkownika	13
4.4	Rozłączenie użytkownika - komunikat	13