**Wydział Nauk Informatyczno-Technologicznych**

Kierunek studiów: **Informatyka I stopnia**

Ścieżka rozwoju: Systemy operacyjne

**Damian Piotrowski**

10510

**PROJEKT I IMPLEMENTACJA APLIKACJI**

**DO WYMIANY DANYCH W SYSTEMACH KOMUNIKACJI KLIENT-SERWER**

Praca dyplomowa inżynierska napisana pod kierunkiem:

dr. inż. Andrzeja Sawickiego

Łomża 2024

**Streszczenie**

**Tytuł: Projekt i implementacja aplikacji do wymiany danych w systemach komunikacji klient-serwer**

Celem pracy jest zaprojektowanie i zaimplementowanie aplikacji umożliwiającej wymianę danych w systemach komunikacji typu klient-serwer. Aplikacja została stworzona  
w języku Python, a rolę serwera pełni urządzenie Raspberry Pi z zainstalowanym systemem operacyjnym Linux. Komunikacja pomiędzy klientem a serwerem została zrealizowana za pomocą biblioteki socket, umożliwiającej obsługę protokołów sieciowych, oraz modułu threading, który zapewnia wielowątkowość i efektywne zarządzanie połączeniami. Graficzny interfejs użytkownika aplikacji został zaprojektowany i zaimplementowany z użyciem biblioteki tkinter, oferując intuicyjną i funkcjonalną obsługę aplikacji. Praca opisuje proces projektowania, implementacji oraz testowania aplikacji, uwzględniając wyzwania związane  
z komunikacją sieciową i wielowątkowością.

**Słowa kluczowe**

Python, Raspberry Pi, Komunikacja sieciowa, Wielowątkowość, Linux

**Summary**

**Title: Design and implementation of an application for data exchange in client-server communication systems**

The aim of this project is to design and implement an application enabling data exchange in client-server communication systems. The application was developed in Python, with a Raspberry Pi device running a Linux operating system serving as the server. Communication between the client and the server was implemented using the socket library, which facilitates network protocol handling, and the threading module, which ensures multithreading and efficient connection management. The graphical user interface of the application was designed and developed using the tkinter library, providing an intuitive and functional user experience. This work describes the process of designing, implementing, and testing the application, addressing challenges related to network communication and multithreading.

**Keywords**

Python, Raspberry Pi, Network communication, Multithreading, Linux

Spis treści

[WSTĘP 5](#_Toc188771325)

[Problem inżynierski 5](#_Toc188771326)

[Cel pracy 6](#_Toc188771327)

[Zakres pracy 6](#_Toc188771328)

[1. Analiza tematu, literatury, dostępnych rozwiązań 7](#_Toc188771329)

[1.1. Analiza literatury 7](#_Toc188771330)

[1.1.1. Client-Server Architecture - System Design 7](#_Toc188771331)

[1.2. Analiza istniejących rozwiązań 9](#_Toc188771332)

[1.3. Analiza istniejących technologii 10](#_Toc188771333)

[1.4. Wybór technologii 15](#_Toc188771334)

[2. Projekt systemu informatycznego, aplikacji, Zaproponowanego rozwiązania problemu inżynierskiego 16](#_Toc188771335)

[2.1. Omówienie projektu 16](#_Toc188771336)

[2.1.1. Zasada działania systemu informatycznego / aplikacji 16](#_Toc188771337)

[2.2. Wymagania projektu 16](#_Toc188771338)

[2.3. Projekt systemu, aplikacji, rozwiązania problemu inżynierskiego 17](#_Toc188771339)

[2.3.1. Schemat blokowy systemu 17](#_Toc188771340)

[2.3.2. Diagram klas systemu 18](#_Toc188771341)

[2.3.3. Opis działania i logiki aplikacji 19](#_Toc188771342)

[2.4. Projekt interfejsu użytkownika 20](#_Toc188771343)

[3. Implementacja aplikacji / Realizacja systemu informatycznego / Rozwiązania problemu inżynierskiego 23](#_Toc188771344)

[3.1. Podstawowe funkcjonalności 23](#_Toc188771345)

[3.2. Wybrane algorytmy 23](#_Toc188771346)

[3.3. Działanie systemu 23](#_Toc188771347)

[3.3.1. Przedstawienie wyników działania zrealizowanej aplikacji / systemu 23](#_Toc188771348)

[3.3.2. Przedstawienie wybranych ścieżek działania aplikacji /systemu 23](#_Toc188771349)

[3.4. Badania wybranych elementów systemu rozwiązującego problem inżynierski 23](#_Toc188771350)

[4. Testy systemu informatycznego / Aplikacji 24](#_Toc188771351)

[4.1. Testy systemu 24](#_Toc188771352)

[4.2. Realizacja testów 24](#_Toc188771353)

[4.2.1. Opis testu 1 24](#_Toc188771354)

[4.2.2. Opis testu 2 24](#_Toc188771355)

[4.2.3. Opis testu 2 24](#_Toc188771356)

[4.3. Wyniki testów 24](#_Toc188771357)

[5. PODSUMOWANIE I WNIOSKI 25](#_Toc188771358)

[Bibliografia 26](#_Toc188771359)

[Spis Tabel 26](#_Toc188771360)

[Spis Rysunków 26](#_Toc188771361)

[Spis Listingów 26](#_Toc188771362)

[Spis ZałĄczników 26](#_Toc188771363)

WSTĘP

Współczesne systemy komunikacji oparte na architekturze klient-serwer odgrywają kluczową rolę w budowie aplikacji sieciowych i systemów IoT. Umożliwiają one efektywną wymianę danych między różnymi urządzeniami, co znajduje zastosowanie w wielu dziedzinach, takich jak automatyka domowa, zarządzanie danymi w chmurze czy systemy monitorowania w czasie rzeczywistym.

W pracy tej skupiono się na opracowaniu aplikacji, która w sposób elastyczny i wydajny realizuje wymianę danych w architekturze klient-serwer, wykorzystując popularne narzędzia    
i technologie, takie jak Python, Raspberry Pi oraz biblioteki socket i tkinter. Takie podejście pozwala na stworzenie rozwiązania, które jest zarówno dostępne cenowo, jak i uniwersalne   
w zastosowaniu.

## Problem inżynierski

Głównym problemem inżynierskim, który rozwiązano w tej pracy, jest zaprojektowanie i implementacja systemu wymiany danych w architekturze klient-serwer, który jednocześnie:

* Zapewnia stabilną i bezpieczną komunikację pomiędzy klientem a serwerem,
* Obsługuje wiele równoczesnych połączeń dzięki zastosowaniu wielowątkowości,
* Udostępnia użytkownikom intuicyjny interfejs graficzny,
* Może działać na urządzeniach o ograniczonych zasobach sprzętowych, takich jak Raspberry Pi.

## Cel pracy

W ramach pracy powstanie aplikacja do komunikacji asynchronicznej pomiędzy użytkownikami. Celem pracy jest zaprojektowanie i implementacja aplikacji umożliwiającej przesyłanie tekstu, zdjęć, filmów i wiadomości głosowych między klientami za pośrednictwem serwera. Drugim komponentem systemu będzie aplikacja serwerowa działająca zdalnie na urządzeniu Raspberry Pi z zainstalowanym systemem operacyjnym Linux, do którego użytkownicy mogą połączyć się za pomocą adresu IP oraz portu.

## Zakres pracy

Zakres pracy obejmuje:

* Analizę wymagań dotyczących systemów komunikacji klient-serwer,
* Projekt architektury aplikacji, uwzględniający specyfikę urządzeń o ograniczonych zasobach, takich jak Raspberry Pi,
* Implementację aplikacji serwera oraz klienta z wykorzystaniem języka Python,
* Wykorzystanie bibliotek socket i threading do obsługi połączeń sieciowych oraz wielowątkowości,
* Stworzenie graficznego interfejsu użytkownika za pomocą tkinter,
* Testowanie aplikacji pod kątem stabilności, wydajności oraz łatwości użytkowania,
* Dokumentację procesu projektowania, implementacji i testowania aplikacji, a także wnioski dotyczące możliwości rozwoju systemu.

# Analiza tematu, literatury, dostępnych rozwiązań

## Analiza literatury

### Client-Server Architecture - System Design

W artykule “Client-Server Architecture - System Design” zamieszczonym na stronie geeksforgeeks.org autorzy analizują rozwiązania związane z architekturą klient-serwer.

Autorzy podają, że architektura klient-serwer jest podstawą nowoczesnego projektowania systemów, gdzie infrastruktura składa się z jednego serwera centralnego oraz wielu klientów. Autorzy zaznaczają, że w tym modelu klient rozumiany jest jako urządzenie lub program, który wykonuje żądania usług lub zasobów a serwer jest maszyną lub programem, który na te żądania odpowiada. Komunikacja pomiędzy klientami odbywa się za pomocą protokołów żądanie-odpowiedź, takich jak HTTP/HTTPS dla systemów sieciowych, bądź SQL dla systemów bazodanowych.

Główne cechy systemu klient-serwer wynikające z tekstu artykułu:

* Wydajne zarządzanie danymi oraz przyporządkowywanie zasobów systemu centralizując krytyczne funkcje serwerowi,
* Klient zarządza interakcją z użytkownikiem i wysyła specyficzne żądania do serwera, który przetwarza te żądania i odpowiada w należyty sposób,
* Duża skalowalność systemu, który jest w stanie przyjąć więcej klientów dzięki skalowaniu zdolności serwera, bądź dodanie nowych serwerów,
* Daje podstawy pod bardziej skomplikowane systemy.

Artykuł podkreśla krytyczne znaczenie architektury klient-serwer dla projektowania systemów. Poprzez centralizację zasobów systemu, ta architektura upraszcza w dużym stopniu utrzymanie, aktualizacje oraz bezpieczeństwo systemu. Administratorzy są w stanie efektywnie monitorować i zarządzać danymi, wgrywać aktualizacje oraz egzekwować zasady bezpieczeństwa z jednego miejsca. System pozwala na dużą skalowalność poprzez dodanie nowych serwerów lub zwiększenia możliwości istniejących serwerów, co pozwala rozwijać system, tak aby mógł on obsługiwać dowolną liczbę klientów, nie ingerując przy tym w ogólną architekturę systemu.

Autorzy podają także, że takie rozwiązanie niesie za sobą korzyści  
w aspektach optymalizacji zarządzania zasobami, niezawodności oraz bezpieczeństwa. Według autorów centralizowane serwery pozwalają na wprowadzenie lepszych protokołów bezpieczeństwa oraz ochrony danych.

Architektura klient-serwer zakłada istnienie kilku kluczowych składników, które działają razem, aby zapewnić wydajną komunikację. Są to klient, serwer, sieć, protokoły, urządzenia pośrednie, baza danych, interfejs użytkownika i logika aplikacji.

Autorzy podają kilka rozwiązań oraz narzędzi dla architektury klient-serwer. W ich skład wchodzą frameworki oraz programy dla serwera, klienta oraz bazy danych:

* React – biblioteka JavaScript służąca do budowania interfejsów użytkownika, utrzymywany przez Facebook,
* Django – wysokopoziomowy framework Pythona, który zawiera wbudowany interfejs admina oraz mechanizmy potwierdzania tożsamości,
* MySQL – system do prowadzenia baz danych. Działa on w formacie open-source.

Protokoły, API oraz narzędzia do budowania aplikacji:

* REST – styl architektury do projektowania aplikacji sieciowych. Używa standardowego protokołu HTTP,
* WebSocket – protokół pozwalający na pełną transmisję dwukierunkową używając jednego połączenia TCP. Idealne rozwiązanie dla aplikacji czasu rzeczywistego,
* Git – system kontroli wersji, służący do śledzenia zmian w kodzie przy projektowaniu aplikacji. Niezbędny do grupowej pracy nad aplikacjami.

Autorzy podają, że takie rozwiązanie ma szerokie zastosowanie w świecie rzeczywistym, w dziedzinach takich jak: bankowość, telekomunikacja, internet rzeczy czy systemy rządowe.

## Analiza istniejących rozwiązań

**Gotowe frameworki sieciowe**:

Frameworki takie jak Flask czy Django oferują zaawansowane mechanizmy komunikacyjne, ale są stosunkowo ciężkie dla urządzeń o ograniczonych zasobach, jak Raspberry Pi.

**Protokół MQTT**:

Lekki protokół często używany w systemach IoT. Zapewnia wysoką wydajność, ale wymaga dodatkowych brokerów danych, co komplikuje implementację w małych projektach.

**Protokół HTTP/HTTPS:**

Używany w aplikacjach webowych do komunikacji klient-serwer. Serwer odbiera żądania od klienta, przetwarza je, a następnie na nie odpowiada.

**Web Socket:**

Protokół umożliwiający komunikację pełnodupleksową w czasie rzeczywistym, wykorzystywany w aplikacjach wymagających natychmiastowej wymiany danych (np. czaty, powiadomienia).

## Analiza istniejących technologii

**Python**:

Python to jeden z najpopularniejszych języków programowania, ceniony za swoją prostotę, czytelność kodu oraz szeroki ekosystem bibliotek i narzędzi. Dzięki temu jest idealnym wyborem do implementacji aplikacji klient-serwer, zarówno po stronie serwera, jak  
i klienta. Python wspiera różnorodne protokoły komunikacyjne, takie jak TCP, UDP, czy HTTP, co pozwala na tworzenie systemów dostosowanych do specyficznych potrzeb.

Kluczowe zalety:

* **Prostota składni**: Python umożliwia szybkie prototypowanie i rozwój aplikacji. Intuicyjna składnia sprawia, że jest przystępny zarówno dla początkujących, jak  
  i doświadczonych programistów.
* **Bogaty ekosystem**: Python oferuje wiele bibliotek i frameworków, takich jak Flask, Django (do aplikacji webowych) czy Asyncio (do obsługi asynchronicznej).
* **Przenośność**: Kod Python działa na różnych systemach operacyjnych, co czyni go wszechstronnym narzędziem dla różnych platform sprzętowych i programowych.

**Raspberry Pi**:

Raspberry Pi to kompaktowy komputer jednopłytkowy, który stał się kluczowym elementem wielu projektów edukacyjnych, IoT (Internet of Things) oraz systemów automatyki. Dzięki pełnej obsłudze systemu Linux oraz wsparciu dla Python, Raspberry Pi doskonale nadaje się do budowy systemów klient-serwer.

Kluczowe zalety:

* **Wszechstronność sprzętowa**: Raspberry Pi obsługuje różnorodne interfejsy komunikacyjne, takie jak Ethernet, Wi-Fi, Bluetooth, czy GPIO, co pozwala na integrację z sensorami i innymi urządzeniami IoT.
* **Ekosystem Linuksa**: Umożliwia instalację bibliotek sieciowych, serwerów HTTP oraz narzędzi do monitorowania ruchu sieciowego.
* **Niski koszt**: Raspberry Pi jest przystępne cenowo, co czyni je idealnym narzędziem dla małych projektów i prototypowania.

**Biblioteka socket**:

Biblioteka socket w Python pozwala na niskopoziomową komunikację sieciową, umożliwiając tworzenie zarówno serwerów, jak i klientów. Oferuje pełną kontrolę nad procesem przesyłania danych, co pozwala na implementację protokołów dostosowanych do konkretnych wymagań.

Kluczowe zalety:

* **Tworzenie gniazd**: socket umożliwia konfigurację gniazd TCP i UDP, które obsługują połączenia niezawodne (TCP) oraz szybką wymianę danych (UDP).
* **Obsługa protokołów**: Oferuje wsparcie dla protokołów IPv4 i IPv6, co pozwala na obsługę nowoczesnych sieci.
* **Elastyczność**: Biblioteka pozwala na wysyłanie i odbieranie danych w różnych formatach (tekst, binarny).
* **Łatwość integracji**: Może być używana razem z innymi bibliotekami, np. do szyfrowania transmisji (SSL/TLS).

**Biblioteka threading**:

Biblioteka threading w Python umożliwia tworzenie wielowątkowych aplikacji, co jest kluczowe w systemach obsługujących wielu klientów jednocześnie. W połączeniu z socket pozwala na implementację serwerów, które mogą efektywnie zarządzać równoczesnymi połączeniami.

Kluczowe zalety:

* **Obsługa wielu klientów**: Każde połączenie może być obsługiwane w osobnym wątku, co zapobiega blokowaniu serwera przez długotrwałe operacje.
* **Łatwa implementacja**: Python dostarcza prosty interfejs do tworzenia wątków, co przyspiesza proces projektowania.
* **Integracja z innymi bibliotekami**: threading współpracuje z bibliotekami takimi jak queue, co pozwala na efektywne zarządzanie zadaniami między wątkami.

**Biblioteka Tkinter:**

Tkinter to wbudowana biblioteka Python do tworzenia graficznych interfejsów użytkownika. Jest lekka, łatwa w użyciu i umożliwia szybkie prototypowanie aplikacji, które wymagają interakcji użytkownika.

Kluczowe zalety:

* **Prostota i intuicyjność**: Umożliwia szybkie tworzenie okien, przycisków, pól tekstowych i innych elementów interfejsu.
* **Integracja z innymi bibliotekami**: Tkinter może współpracować z bibliotekami sieciowymi, umożliwiając tworzenie klientów z graficznym interfejsem.
* **Przenośność**: Działa na większości systemów operacyjnych obsługujących Python.

**Biblioteka cryptography:**

Biblioteka **cryptography** w Pythonie oferuje zaawansowane narzędzia do szyfrowania  
i deszyfrowania danych, co jest niezbędne w aplikacjach, które wymagają bezpiecznej wymiany informacji. Umożliwia korzystanie z różnych algorytmów kryptograficznych, zarówno symetrycznych, jak i asymetrycznych, oraz zapewnia łatwą implementację różnych metod uwierzytelniania i ochrony integralności danych.

Kluczowe zalety:

* **Szyfrowanie:** Umożliwia wykorzystanie algorytmów AES,RSA
* **Algorytmy hashujące: cryptography** wspiera różne algorytmy hashujące, takie jak SHA-256, które są wykorzystywane do zapewnienia integralności danych.
* **Łatwość użycia i dokumentacja: cryptography** dostarcza prosty interfejs API i jest dobrze udokumentowana, co ułatwia implementację szyfrowania w aplikacjach Pythonowych. Działa na wielu systemach operacyjnych i jest odpowiednia dla różnych zastosowań, od aplikacji webowych po systemy IoT.

**GitHub:**GitHub to platforma hostingowa dla projektów programistycznych, która wykorzystuje system kontroli wersji Git. Jest nie tylko miejscem do przechowywania kodu, ale także narzędziem do współpracy zespołowej, zarządzania projektami i automatyzacji procesów deweloperskich.

**Kluczowe zalety:**

* **System kontroli wersji:** GitHub pozwala na śledzenie zmian w kodzie, co umożliwia łatwe cofanie błędnych modyfikacji i analizowanie historii projektu.
* **Automatyzacja:** GitHub umożliwia automatyzację testów, budowania i wdrażania aplikacji.
* **Integracja z narzędziami:** Platforma współpracuje z wieloma narzędziami zewnętrznymi, np. Jenkins, Slack, czy Docker.

**Docker:**

Docker to platforma umożliwiająca konteneryzację aplikacji, co sprawia, że aplikacje mogą być uruchamiane w izolowanym środowisku niezależnie od systemu operacyjnego. Dzięki temu jest szczególnie użyteczny w projektach klient-serwer, gdzie wymagane jest łatwe wdrażanie i skalowanie aplikacji.

**Kluczowe zalety:**

* **Izolacja środowisk:** Każda aplikacja działa w swoim kontenerze, co eliminuje konflikty między zależnościami.
* **Przenośność:** Kontenery można uruchamiać na różnych platformach (serwery lokalne, chmura).
* **Szybkość wdrażania:** Docker pozwala na szybkie uruchamianie środowisk testowych i produkcyjnych.

**Flask:**

Flask to lekki framework webowy dla Pythona, idealny do szybkiego tworzenia aplikacji serwerowych. Dzięki prostocie i elastyczności, Flask jest często wykorzystywany w systemach klient-serwer, które wymagają łatwego zarządzania API.

**Kluczowe zalety:**

* **Minimalistyczna struktura:** Flask pozwala na tworzenie aplikacji bez zbędnych komplikacji.
* **Wsparcie dla API:** Prosta integracja z REST i WebSocket.
* **Rozszerzalność:** Flask obsługuje wiele wtyczek, np. Flask-SocketIO do komunikacji  
  w czasie rzeczywistym.
* **Elastyczność:** Framework daje pełną kontrolę nad strukturą projektu.

## Wybór technologii

Na podstawie analizy dostępnych technologii zdecydowano się na:

* **Python** jako główny język programowania, ze względu na jego prostotę, wsparcie dla wieloplatformowych bibliotek oraz aktywną społeczność.
* **Socket i threading** do realizacji połączeń sieciowych oraz obsługi wielowątkowości, co zapewnia pełną kontrolę nad architekturą aplikacji.
* **Cryptography** jako biblioteka zapewniająca bezpieczeństwo danych poprzez szyfrowanie i uwierzytelnianie. Umożliwia implementację protokołów SSL/TLS, szyfrowania AES/RSA, a także tworzenie hashy dla zapewnienia integralności danych w aplikacji klient-serwer.
* **Tkinter** do budowy graficznego interfejsu użytkownika, jako narzędzie zintegrowane  
  z Pythonem i odpowiednie dla projektów o niewielkiej złożoności wizualnej.
* **Raspberry Pi** jako serwer, dzięki jego niskiemu zużyciu energii, małym wymiarom oraz możliwości pracy w środowisku Linux.
* **GitHub** jako platforma do kontroli wersji. Zapewnia możliwość zarządzania kodem, przechowywania historii zmian, a także automatyzacji procesów co przyspiesza testowanie i wdrażanie aplikacji.
* **MobaXterm** jako narzędzie do zarządzania zdalnymi połączeniami, co umożliwia łatwą konfigurację i monitorowanie Raspberry Pi. Dzięki wbudowanym funkcjom terminal SSH, SCP, czy X11-forwarding, MobaXterm upraszcza pracę w środowiskach sieciowych.
* **Dodatkowe biblioteki takie jak os, sys, time, datetime, traceback oraz errno służące do synchronizacji wymiany danych pomiędzy klientami a serwerem, a także debugowania kodu aplikacji.**

Wybór tych technologii umożliwia stworzenie aplikacji, która jest wygodna w użyciu, funkcjonalna, elastyczna i zoptymalizowana pod kątem działania na urządzeniach  
o ograniczonych zasobach sprzętowych.

# Projekt systemu informatycznego, aplikacji, Zaproponowanego rozwiązania problemu inżynierskiego

## Omówienie projektu

### Zasada działania systemu informatycznego / aplikacji

Celem projektu jest stworzenie aplikacji umożliwiającej komunikację w architekturze klient-serwer, zapewniającej stabilną wymianę danych pomiędzy użytkownikami. System składa się z dwóch głównych komponentów: serwera działającego na Raspberry Pi oraz aplikacji klienckiej uruchamianej na komputerze lub innym urządzeniu z systemem operacyjnym obsługującym Pythona. Serwer zarządza połączeniami wielowątkowymi oraz wymianą danych, a aplikacja kliencka oferuje użytkownikowi graficzny interfejs do interakcji z systemem.

## Wymagania projektu

**Funkcjonalne**:

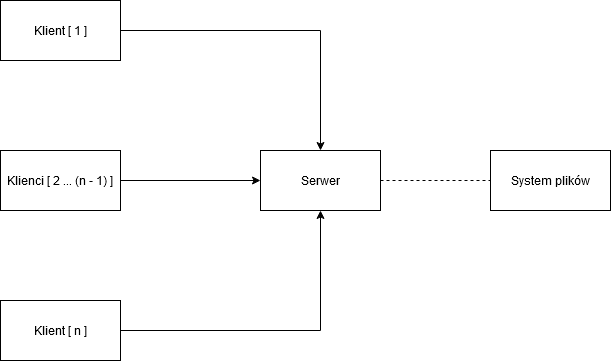
* Obsługa wielu klientów jednocześnie,
* Wysyłanie i odbieranie danych w czasie rzeczywistym,
* Prosty, intuicyjny interfejs graficzny.

**Niefunkcjonalne**:

* Działanie serwera na Raspberry Pi z systemem Linux,
* Minimalne opóźnienia w komunikacji,
* Bezpieczeństwo danych,
* Łatwa instalacja i konfiguracja aplikacji na urządzeniach klienckich.

## Projekt systemu, aplikacji, rozwiązania problemu inżynierskiego

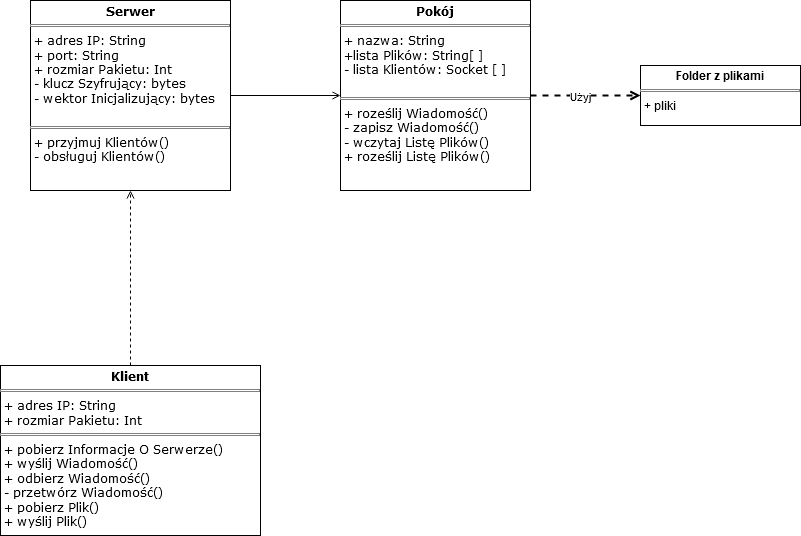
### Schemat blokowy systemu



Rysunek . Schemat blokowy systemu

Na rysunku 2.1 przedstawiony jest schemat blokowy aplikacji. Serwer jest połączony  
z systemem plików przez system operacyjny. Serwer może przyjąć n klientów, gdzie n zależy od zasobów systemowych urządzenia, na którym działa serwer.

### Diagram klas systemu



Rysunek . Diagram klas systemu

Na rysunku 2.2 przedstawiono diagram klas systemu, reprezentujący klasy, które biorą udział w wymianie danych w ramach działania aplikacji. Klasa „Pokój” istnieje wewnątrz klasy „Serwer” przechowuje listę nazw własnych plików oraz listę klientów obecnie zalogowanych do danego pokoju. Klasa „Serwer” przechowuje dane konieczne do odbierania i przetwarzania połączeń oraz wiadomości. Serwer wykorzystuje pokój do rozsyłania wiadomości do odpowiednich klientów. Klient zazwyczaj (nie jest to konieczne) istnieje na osobnym urządzeniu i posiada własny interfejs użytkownika. Przechowuje on klucz oraz wektor szyfrujący i jest w stanie wysyłać oraz odbierać wiadomości od serwera.

### Opis działania i logiki aplikacji

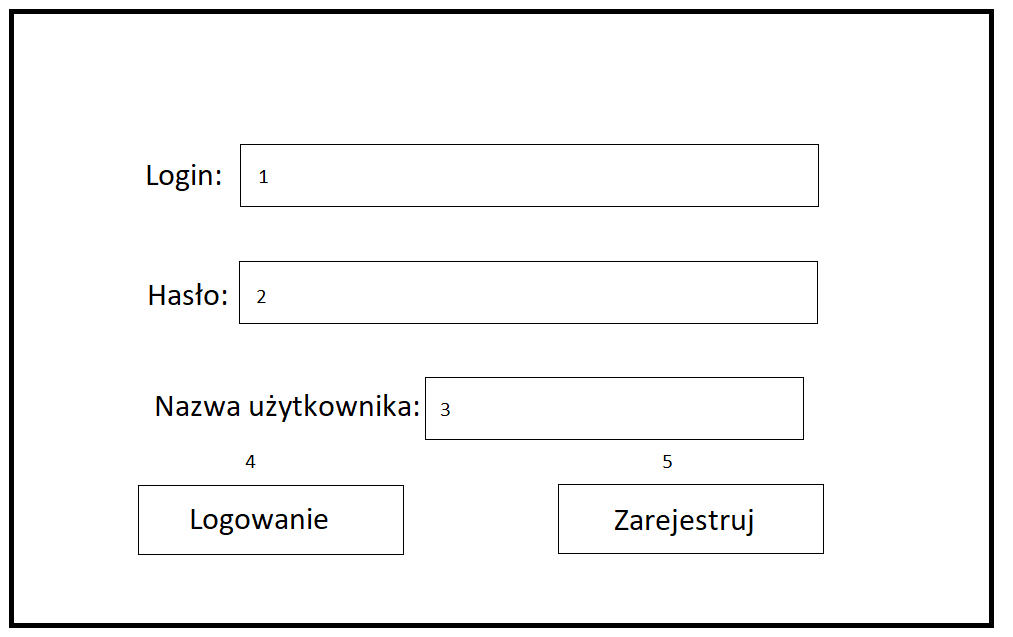
Poniżej znajduje się schemat działania aplikacji:

* Administrator uruchamia serwer,
* Serwer pobiera z pliku dane konfiguracyjne, takie jak adres IP hosta oraz port,
* Serwer generuje klucz AES oraz wektor IV, dane te są unikatowe dla każdego uruchomienia serwera,
* Serwer inicjalizuje się wczytując listę zarejestrowanych użytkowników,
* Serwer inicjalizuje pokoje jako podfoldery aplikacji. Każdy z nich ma osobny plik czatu oraz osobny folder z plikami,
* Klient uruchamia aplikację, rejestruje się i loguje,
* Po zalogowaniu klient otrzymuje klucz AES od serwera, którym będzie szyfrował oraz rozszyfrowywał wiadomości,
* Po nawiązaniu połączenia serwer tworzy nowy wątek dedykowany klientowi,
* Klient wysyła dane do serwera, który je odbiera i odpowiednio przetwarza,
* Serwer reaguje na odebrane wiadomości i wysyła na ich podstawie odpowiednie wiadomości do klientów, co jest wizualizowane w graficznym interfejsie.

## Projekt interfejsu użytkownika

Interfejs użytkownika oferuje klientom prostą w obsłudze i intuicyjną komunikację  
z serwerem. Interfejs przeznaczony jest dla przeciętnych użytkowników, dlatego musi być prosty, a jego możliwości powinny być oczywiste dla użytkownika. W celu stworzenia interfejsu wykorzystany będzie wbudowany moduł pythona Tkinter, który idealnie nadaje się do tego zadania.

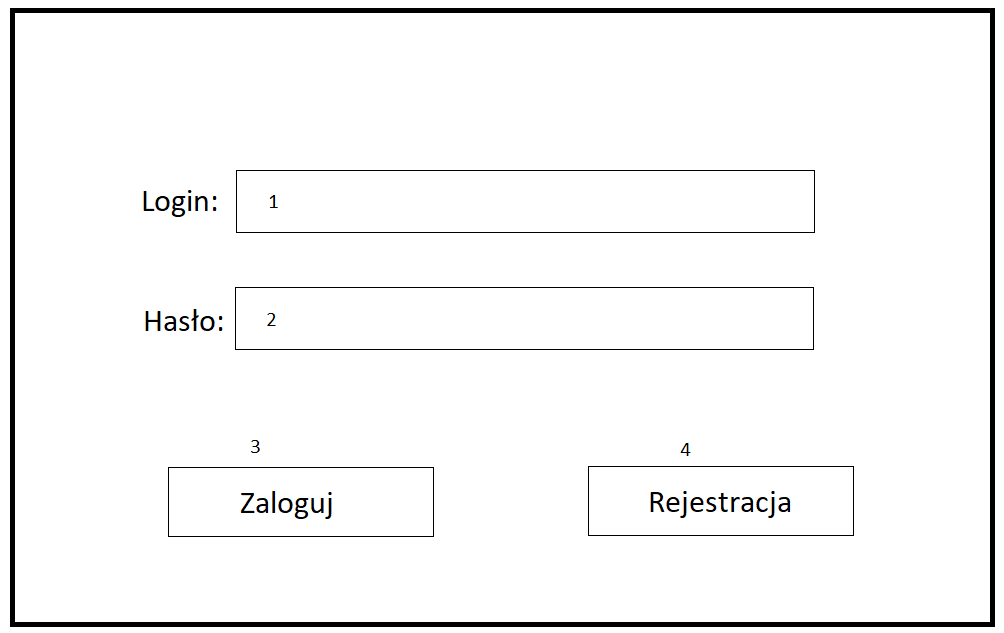
Graficzny projekt interfejsu użytkownika wraz z opisanymi elementami:



Rysunek . Szkic ekranu rejestracji użytkownika

Elementy:

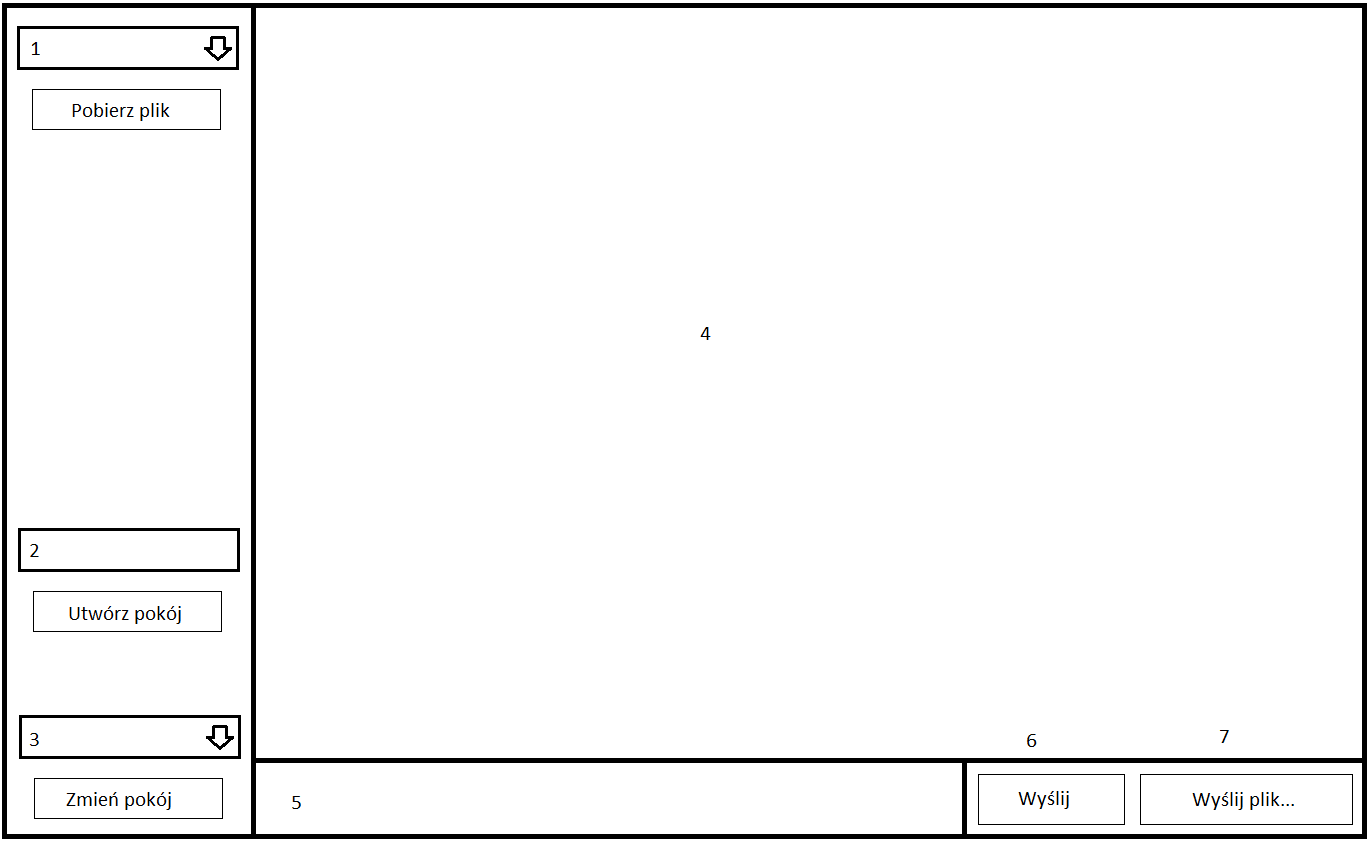
1. Pole do wpisywania loginu,
2. Pole do wpisywania hasła,
3. Pole do wpisywania nazwy użytkownika widocznej dla innych użytkowników,
4. Przycisk przenoszący do ekranu logowania,
5. Przycisk odpowiedzialny za przesłanie danych do serwera.



Rysunek . Szkic ekranu logowania użytkownika

Elementy:

1. Pole do wpisywania loginu,
2. Pole do wpisywania hasła,
3. Przycisk odpowiedzialny za przesłanie danych logowania do serwera,
4. Przycisk przenoszący użytkownika do ekranu rejestracji.



Rysunek . Szkic głównego ekranu aplikacji

Elementy:

1. Rozwijana lista dostępnych do pobrania plików z możliwością wyboru jednego z nich wraz z przyciskiem poniżej służącym do żądania pobrania danego pliku od serwera,
2. Pole do wpisywania nazwy pokoju, który użytkownik będzie chciał utworzyć wraz  
   z przyciskiem poniżej służącym do żądania utworzenia nowego pokoju,
3. Rozwijana lista dostępnych pokojów z możliwością wyboru jednego z nich wraz  
   z przyciskiem poniżej służącym do żądania zmiany pokoju przez użytkownika,
4. Okno z czatem aktywnego pokoju, zawiera wszystkie wiadomości wysyłane przez użytkowników w danym pokoju,
5. Pole do wpisywania wiadomości tekstowej do wysłania,
6. Przycisk służący do wysłania zawartości pola 5 do serwera,
7. Przycisk służący do wyboru pliku, który użytkownik chce przesłać.

# Implementacja aplikacji / Realizacja systemu informatycznego / Rozwiązania problemu inżynierskiego

## Podstawowe funkcjonalności

**Obsługa komunikacji między klientem a serwerem:**

* Wysyłanie i odbieranie wiadomości
  + Klient wysyła wiadomość w postaci zaszyfrowanego ciągu znaków.
  + Serwer odbiera wiadomość, rozpoznaje odbiorców (indywidualnych lub grupowych) i rozsyła ją do odpowiednich klientów.
* Obsługa wiadomości z różnymi nagłówkami

|  |  |
| --- | --- |
| **Nagłówek** | **Funkcja** |
| [REGISTER] | Rejestracja użytkownika |
| [LOGIN] | Próba zalogowania użytkownika |
| [ROOM] | Próba utworzenia nowego pokoju |
| [ROOMCHANGE] | Zmiana pokoju przez użytkownika |
| [FILE] | Rozpoczęcie odbierania pliku od użytkownika |
| [FILEREQUEST] | Żądania pobrania pliku przez użytkownika |
| [LOAD] | Żądanie załadowania użytkownika |
| [OK] | Wszystko w porządku, można wznowić pracę |
| [ERROR] | Wystąpił błąd przy wykonywaniu operacji |

Tabela . Tabela nagłówków przetwarzanych przez serwer

|  |  |
| --- | --- |
| **Nagłówek** | **Funkcja** |
| [LIST] | Dodaje plik do listy plików dostępnych do pobrania |
| [ROOM] | Dodaje pokój do listy dostępnych pokoi |
| [FILE] | Rozpoczęcie odbierania pliku od serwera |
| [MSG] | Wiadomość tekstowa |
| [OK] | Wszystko w porządku, można wznowić pracę |
| [ERROR] | Wystąpił błąd przy wykonywaniu operacji |

Tabela . Tabela nagłówków przetwarzanych przez aplikację klienta

Odpowiednie nagłówki automatycznie dodawane są do wiadomości wysyłanych przez klienta oraz serwer. Użytkownik nie musi się nimi przejmować i spokojnie może zignorować ich istnienie.

* Klient może:
  + Rejestrować się,
  + Logować się,
  + Tworzyć nowe pokoje,
  + Dołączać do istniejących pokoi,
  + Wysyłać pliki,
  + Pobierać pliki z pokoju,
  + Wysyłać wiadomości tekstowe
* Serwer zarządza listą aktywnych pokoi i użytkowników w nich obecnych
* Szyfrowanie wiadomości:
  + Użycie algorytmu AES w trybie CBC,
  + Wiadomości są szyfrowane z odpowiednim dopełnieniem przed wysłaniem,
  + Odszyfrowanie wiadomości po stronie odbiorcy.
  + Serwer generuje unikalne klucze dla każdej sesji,

**Interfejs użytkownika:**

* Okno czatu:
  + Pole tekstowe do wpisywania wiadomości,
  + Przycisk wysyłania wiadomości,
  + Wyświetlanie historii czatu w formie listy z nazwami użytkowników  
    i znacznikami czasu.
* Lista pokoi:
  + Lista dostępnych pokoi z możliwością dołączenia,
  + Możliwość tworzenia nowych pokoi o dowolnej nazwie.
* Załączniki:
  + Przycisk do wysyłania dowolnych plików,
  + Widoczny status wysyłania pliku.

**Obsługa błędów i wyjątków:**

* Rozróżnienie wyjątków sieciowych, takich jak:
  + Utrata połączenia (Connection Reset),
  + Próby komunikacji przy braku danych do odebrania z gniazda, gdy pracuje ono w trybie nieblokującym (WSAEWOULDBLOCK),
  + Mechanizm wielokrotnych prób ponownego połączenia w przypadku rozłączenia.
* Błędy szyfrowania:
  + Walidacja danych przed deszyfrowaniem,
  + Obsługa błędów związanych z dopełnieniem i długością wiadomości.

**Bezpieczeństwo:**

* Szyfrowanie treści wiadomości za pomocą AES zapewnia integralność  
  i poufność przesyłanych danych,
* Hashowanie haseł użytkowników przy logowaniu za pomocą SHA-256,
* Walidacja wiadomości.

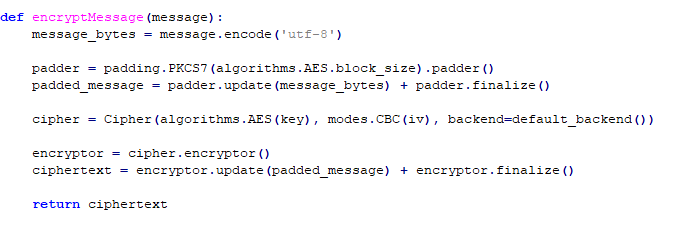
**Funkcje serwera:**

* Zarządzanie połączeniami
  + Akceptowanie wielu klientów jednocześnie,
  + Przechowywanie listy aktywnych użytkowników oraz pokoi w pamięci serwera.
* Rozsyłanie wiadomości
  + Rozpoznawanie adresatów wiadomości,
  + Obsługa broadcastów do wszystkich użytkowników w pokoju.
* Logowanie
  + Zapis historii czatu do pliku,
  + Zapis plików od użytkowników do odpowiedniego folderu,
  + Możliwość zapisu zdarzeń, takich jak dołączenie użytkownika do serwera lub utworzenie nowego pokoju.

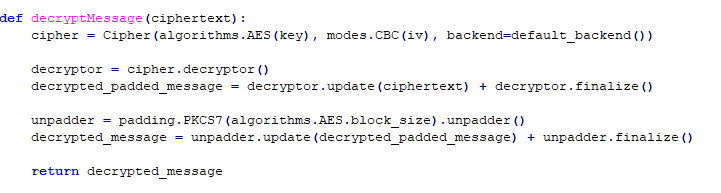
## Wybrane algorytmy

**Algorytmy szyfrowania:**

W ramach szyfrowania powstały następujące funkcje oparte na bibliotece cryptography:



Rysunek . Funkcja szyfrująca wiadomość



Rysunek . Funkcja odszyfrowująca wiadomość

Funkcje te działają analogicznie, ale przeciwnie do siebie. Funkcja encryptMessage konwertuje tekst wejściowy na bajty, dodaje padding PKCS#7, aby dopasować dane do długości bloku AES, szyfruje dane za pomocą algorytmu AES w trybie CBC z kluczem  
i wektorem inicjalizacyjnym, a następnie zwraca zaszyfrowane dane w postaci bajtów. Funkcja decryptMessage odszyfrowuje dane za pomocą AES w trybie CBC, używając dostarczonego klucza i wektora inicjalizacyjnego, usuwa padding PKCS#7, aby uzyskać oryginalne dane,  
a następnie zwraca wiadomość w postaci jawnego tekstu.

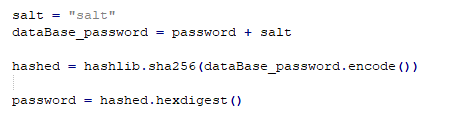
****

Rysunek . Kod generujący klucz oraz wektor

W momencie włączenia serwer generuje parę (klucz, wektor) unikatową dla każdej sesji serwera. Klient dostaje te informacje w momencie, gdy zaloguje się do systemu prawidłowymi danymi.

**Algorytm hashowania haseł:**

Hashowanie haseł użytkowników zrealizowane zostało za pomocą algorytmu SHA-256, dostępnego w ramach biblioteki hashlib.

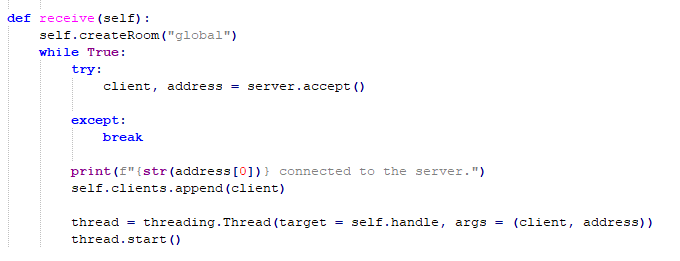


Rysunek . Algorytm hashujący hasło użytkownika

Funkcja łączy hasło użytkownika z solą, tworzy skrót hasła z wykorzystaniem algorytmu SHA-256 i zwraca wartość skrótu w postaci szesnastkowej, która może być przechowywana w bazie danych. Dzięki temu hasła użytkowników są zabezpieczone nawet przed podejrzeniem przez administratora, dlatego, że nigdy nie są wysyłane do serwera  
w formie niezaszyfrowanej.

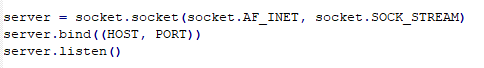
**Algorytmy komunikacji TCP/IP:**

Aplikacja zrealizowana została za pomocą biblioteki socket, która wykorzystuje protokół TCP/IP w celu osiągnięcia komunikacji między serwerem a klientami.



Rysunek . Funkcja obsługująca przyjmowanie nowych klientów

Funkcja receive serwera aktywnie przyjmuje nowe połączenia przychodzące, dodaje nowych klientów do swojej listy oraz rozpoczyna nowy wątek obsługujący każdego klienta.



Rysunek . Kod inicjalizujący serwer

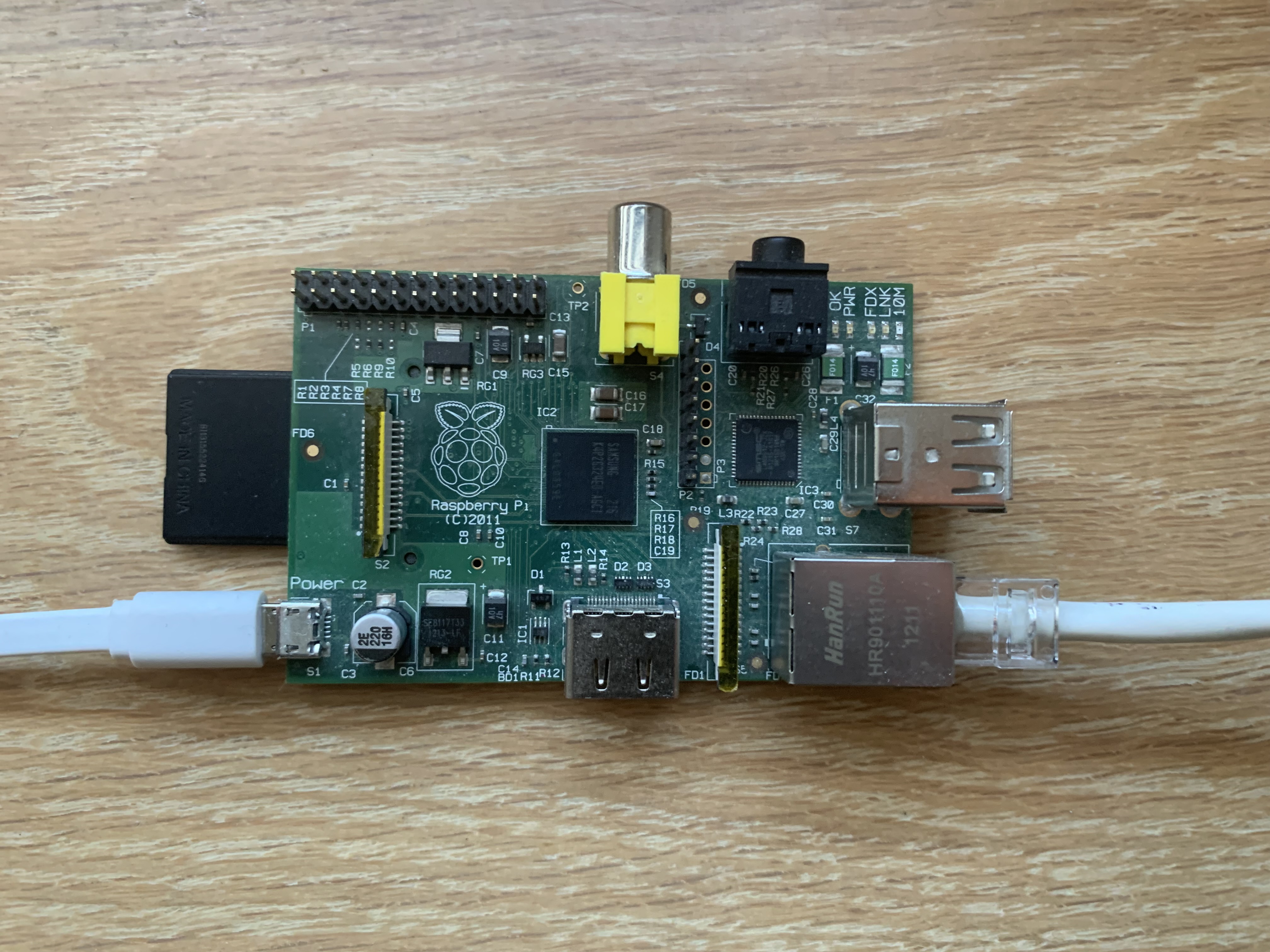
Obiekt server należy do klasy socket, dzięki czemu może korzystać z jej wbudowanych funkcji, takich jak bind, listen, recv. Serwer inicjalizowany jest za pomocą zmiennych HOST oraz PORT, które pobierane są z pliku, lub w przypadku nieobecności tego pliku albo problemów z jego odczytaniem, posiadają swoje domyślne wartości wpisane w kodzie programu.

## Działanie systemu

### Przedstawienie wyników działania zrealizowanej aplikacji / systemu

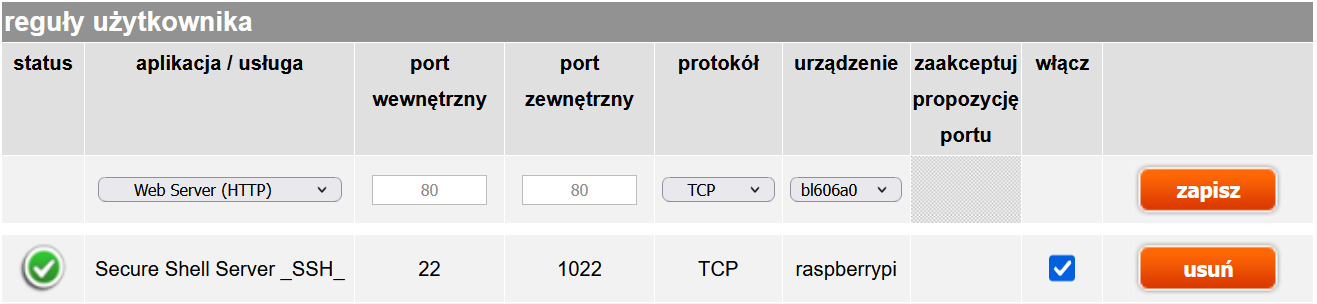
**Strona serwera:**

W celu uruchomienia aplikacji administrator powinien umieścić plik serwera na urządzeniu docelowym. W tym wypadku będzie to urządzenie Raspberry Pi zasilane napięciem o wartości 5V i prądem 2A. Na płytkę za pomocą karty SD został wgrany system Linux. Karta SD ma pojemność 8GB, dzięki czemu będzie w stanie przechować dużą ilość danych użytkowników oraz plików. Została ustawiona domyślna konfiguracja, ustawiono także podstawowego użytkownika z uprawieniami root zabezpieczonego hasłem. Płytkę włączono i wpięto do sieci.



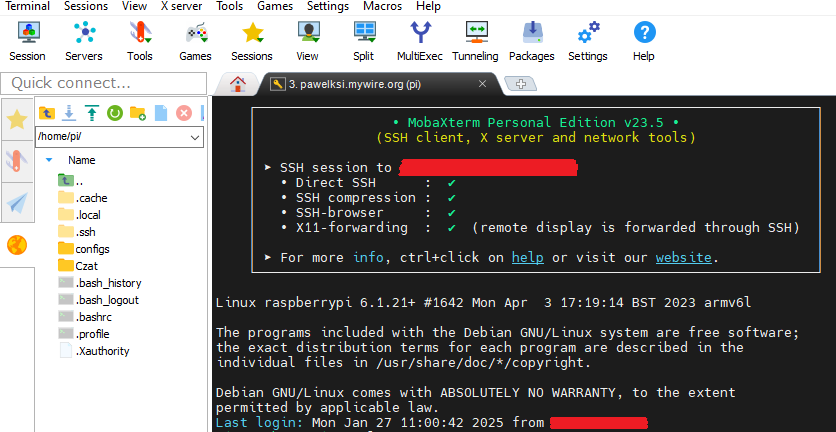
Rysunek . Płytka Raspberry Pi podłączona i gotowa do pracy

Zanim będzie można połączyć się z płytką, należy ustawić translację portów dla serwera SSH w ustawieniach routera, do którego podpięty jest system.



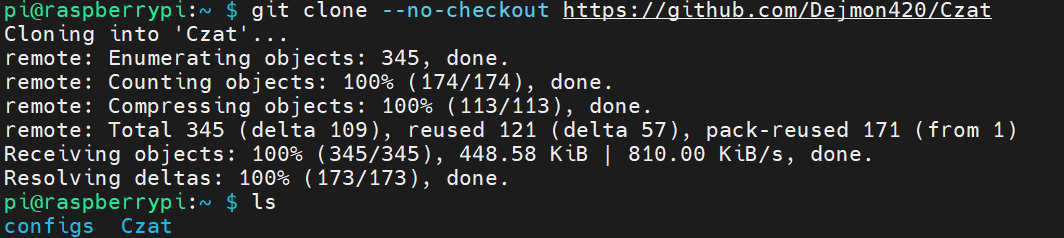
Rysunek . Ustawienia portu do komunikacji SSH

Gdy wszystko zostało ustawione poprawnie, można się połączyć z płytką za pomocą połączenia SSH wykorzystując w tym celu program MobaXterm. Po połączeniu i wpisaniu hasła wcześniej ustalonego użytkownika ukazuje się ekran terminala płytki.



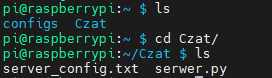
Rysunek . Widok z terminala po zalogowaniu do płytki

Płytka jest gotowa do pracy. W następnej kolejności administrator pobiera pliki serwera z repozytorium GitHub.



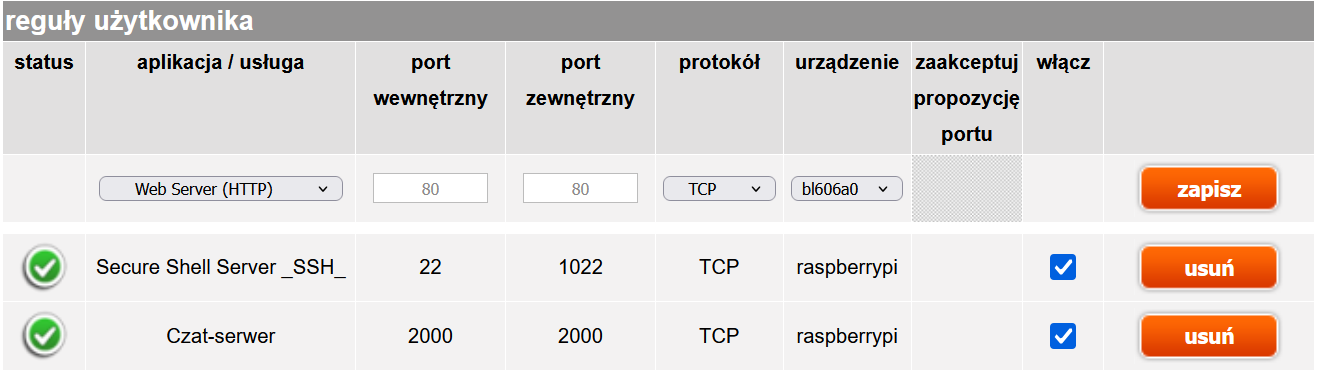
Rysunek . Widok z terminala po pobraniu plików z repozytorium

Po pobraniu odpowiednich plików na płytce w folderze o nazwie „Czat” powinny być dwa pliki: „serwer.py” oraz „server\_config.txt”. Administrator powinien także odpowiednio przygotować plik klienta, wpisując adres docelowy serwera do kodu programu. Zaleca się także spakowanie pliku klienta do rozszerzenia .exe za pomocą modułu pyinstaller, aby zminejszyć ryzyko nadużyć przez ingerencję użytkownika w kod programu.



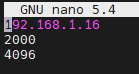
Rysunek . Widok folderu z aplikacją

Przed włączeniem serwera administrator musi otworzyć odpowiedni port zewnętrzny na routerze i przypisać go urządzeniu Raspberry Pi, tak jak aby użytkownicy zewnętrznych sieci mogli łączyć się do serwera za pomocą adresu zewnętrznego oraz portu. Jeśli administrator nie chce aby serwer był widoczny dla sieci zewnętrznych, może pominąć ten krok, wtedy można będzie połączyć się z nim za pomocą adresu wewnętrznego.



Rysunek . Ustawienia portów routera

Następnie administrator powinien odpowiednio zmodyfikować treść pliku „server\_config.txt”, tak aby wartości reprezentowały kolejno: wewnętrzny adres IP przypisany płytce, port otwarty wcześniej na stronie ustawień routera oraz rozmiar pakietu, który serwer będzie wysyłał i odbierał. Rozmiar pakietu na aplikacji klienckiej powinien zostać ustawiony identyczny do tego, który jest ustawiony na serwerze.



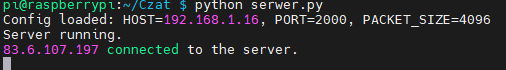
Rysunek . Przykładowa zawartość pliku server\_config.txt

Przed włączeniem serwera należy upewnić się, że zainstalowane są wszystkie niezbędne biblioteki pythona, które on wykorzystuje. Gdy wszystko zostało odpowiednio ustawione, administrator może włączyć serwer i jeśli wszystko poszło zgodnie z planem zobaczy na ekranie komunikat z wypisaną załadowaną konfiguracją oraz wiadomość „Server running.”



Rysunek . Widok administratora po włączeniu serwera

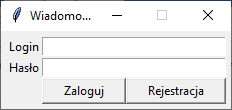
Serwer jest włączony i gotowy na przyjmowanie klientów. Gdy klienci będą łączyć się i rozłączać z serwerem, administrator będzie mógł obserwować odpowiednie komunikaty w oknie terminala.



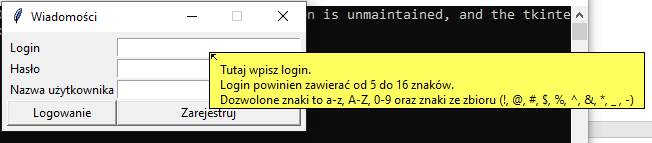
Rysunek . Widok z terminala po połączeniu się klienta

**Strona klienta:**

Klient dostaje plik „clientGUI.exe” od administratora. Jeśli wszystko przebiegło prawidłowo oraz zostało ustalone połączenie między klientem a serwerem, klient zobaczy swój interfejs, gdzie domyślnie przeniesiony zostanie do okna logowania.

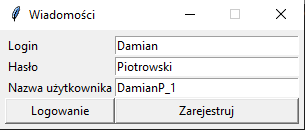


Rysunek . Okno logowania klienta

Po kliknięciu przycisku „Rejestracja” klient zostanie przeniesiony do okna rejestracji. Klient zostanie poproszony o wpisanie loginu, hasła oraz nazwy użytkownika. Znaki dozwolone przy rejestracji to litery od A do Z małe oraz duże, cyfry oraz niektóre znaki specjalne.  


Rysunek . Widok okna rejestracji z komunikatem

Użytkownik rejestruje się, informacja o rejestracji odbierana jest przez serwer oraz odpowiednio przetwarzana, na urządzeniu admninistratora pojawiają się dodatkowe pliki, które przechowują dane użytkowników oraz czaty ze wszystkich pokoi. Serwer domyślnie utworzy pokój o nazwie „global”, gdy jeszcze taki nie istnieje.



Rysunek . Przykładowe wypełnienie arkusza rejestracji

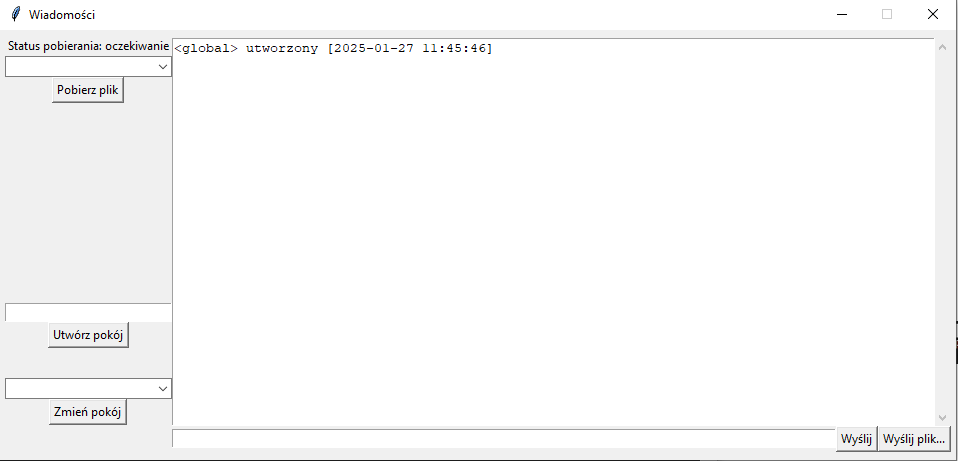


Rysunek . Lista plików serwera



Rysunek . Zawartość pliku users.txt po rejestracji

Na rysunku 3.20 widać zawartość pliku „users.txt”, który został utworzony przez serwer w celu przechowania danych o kontach użytkowników. Serwer zapisuje hasło w postaci zahashowanej, przez co administrator nigdy nie będzie miał wglądu w hasła użytkowników, gwarantując prywatność kont.



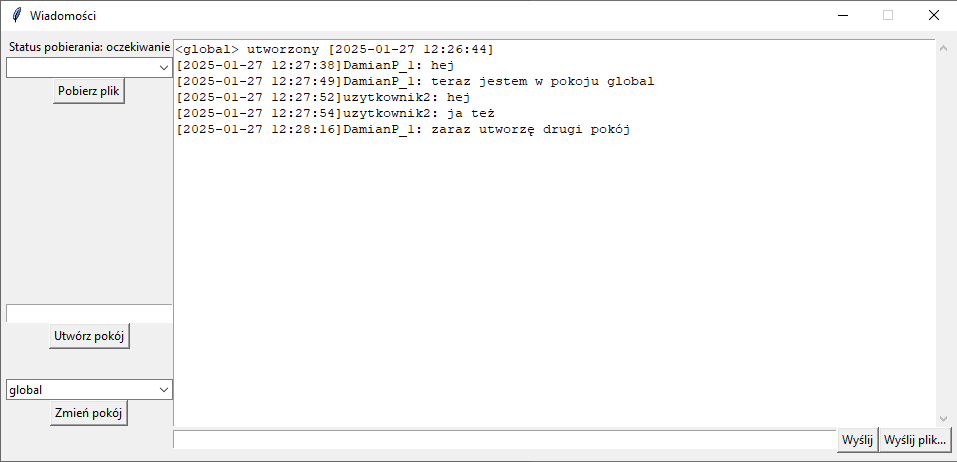
Rysunek . Widok użytkownika po zalogowaniu

Po zalogowaniu użytkownik zobaczy interfejs czatu, gdzie będzie mógł wprowadzić wiadomość do wysłania, utworzyć nowy pokój, zmienić pokój oraz wysłać plik do pokoju. Jak widać na rysunku 3.21 użytkownik domyślnie przypisany został do pokoju „global”. Widać też, kiedy został on utworzony.

### Przedstawienie wybranych ścieżek działania aplikacji /systemu

**Czat:**

Zalogowani użytkownicy czatu mogą wysyłać wiadomości tekstowe do innych osób, które znajdują się w danym pokoju. Osoby przyłączające się do pokoju mogą zobaczyć historię czatu danego pokoju. Poniżej znajduje się przykład rozmowy między dwoma użytkownikami systemu.



Rysunek . Przykładowa rozmowa między użytkownikami

**Tworzenie i zmienianie pokoi:**

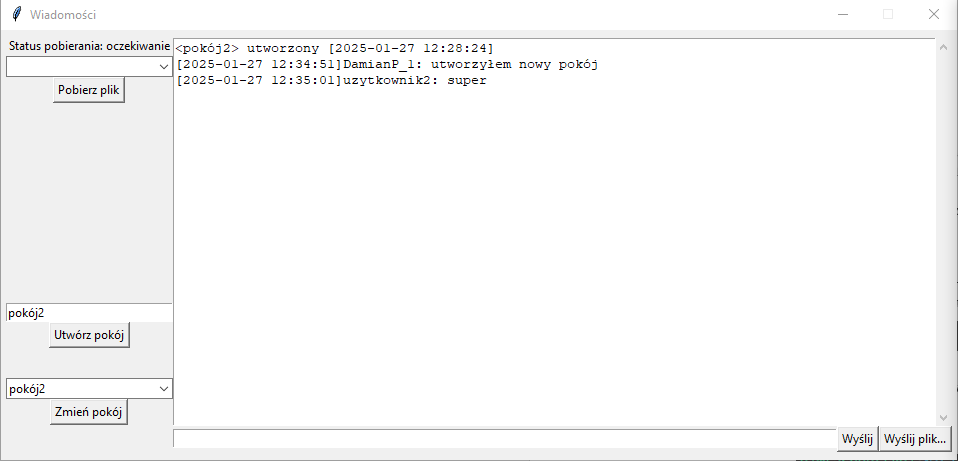
Jeden z użytkowników tworzy nowy pokój o nazwie „pokój2”, na urządzeniu serwera pojawia się folder „pokój2” z plikiem historii czatu pokoju oraz folderem z plikami.



Rysunek . Zawartość serwera po dodaniu pokoju



Rysunek . Zawartość folderu pokój2

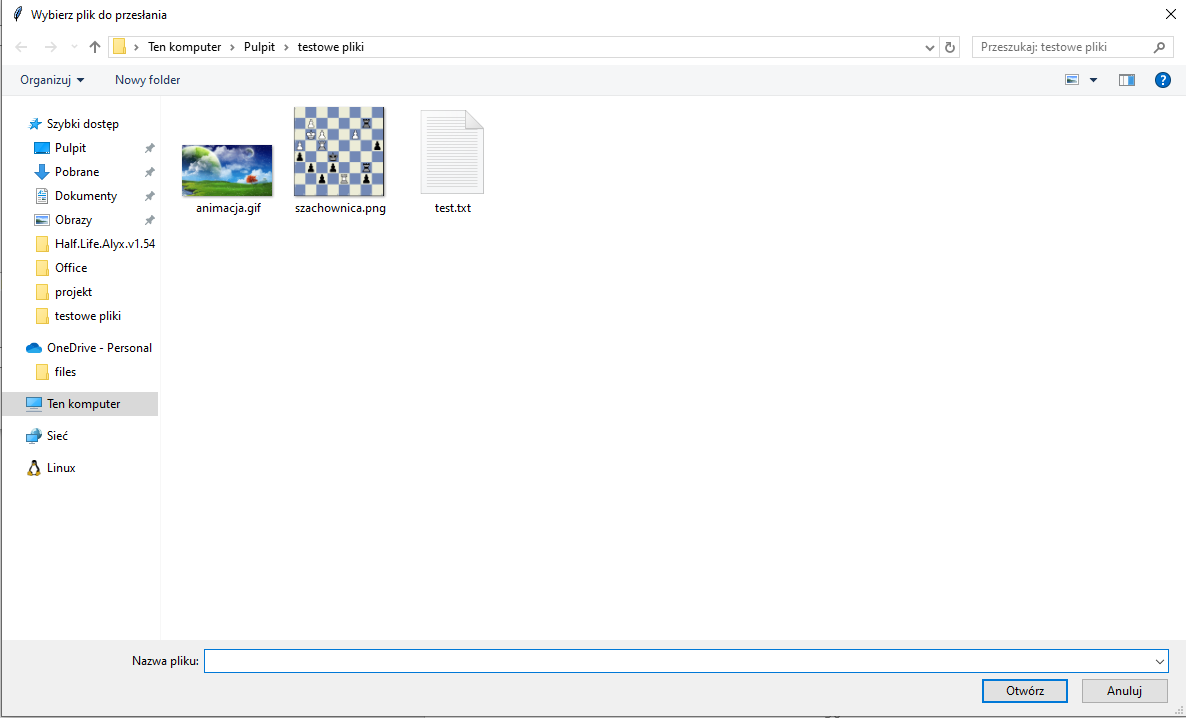


Rysunek . Przykładowa rozmowa z pokoju 2

Wiadomość o utworzonym pokoju została rozesłana do wszystkich zalogowanych użytkowników, którzy teraz widzą go na liście pokojów i mogą do niego dołączyć.

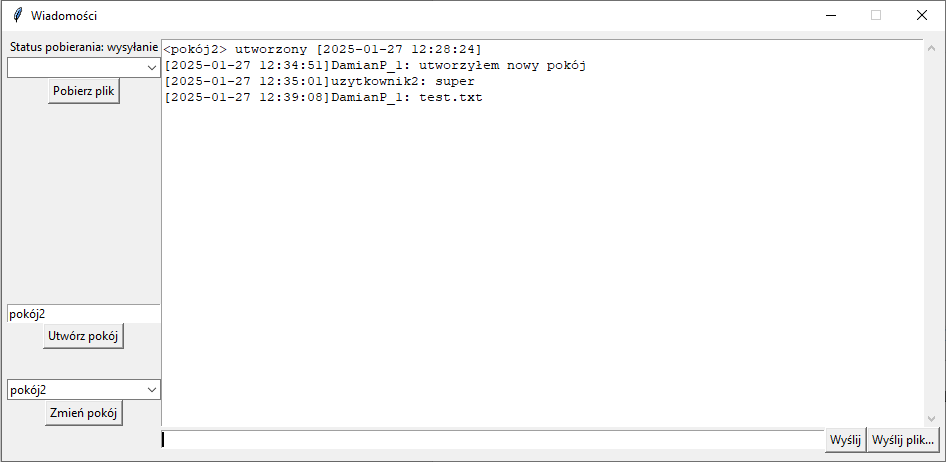
**Wysyłanie i odbieranie plików:**

Użytkownicy są w stanie wysyłać dowolne pliki znajdujące się na ich komputerze do pokoju serwera, w którym aktualnie się znajdują. Do wysyłania plików służy przycisk „Wyślij plik…”. Po wciśnięciu przycisku użytkownikowi ukaże się okno dialogowe systemu, które wymaga wybrania pliku.



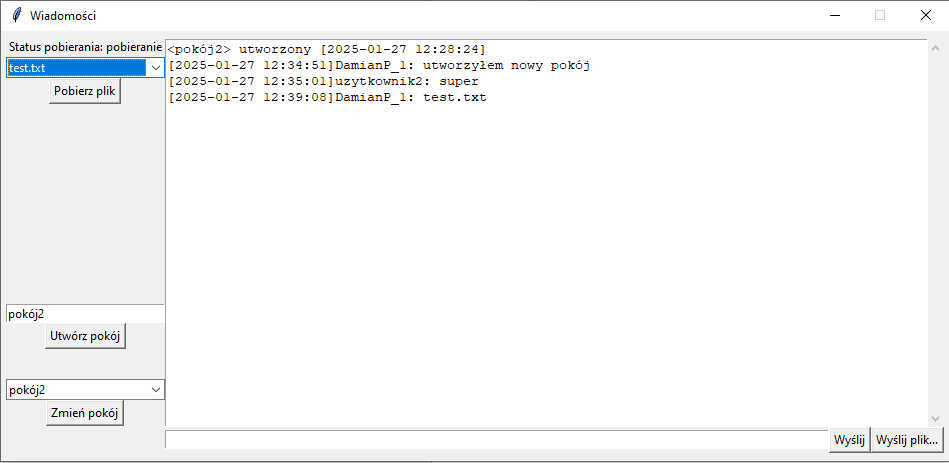
Rysunek . Przykładowy widok okna dialogowego

Użytkownik wysyła przykładowo plik test.txt do pokoju „pokój2”. Po odebraniu pliku przez serwer, zostanie on zapisany na urządzeniu oraz do wszystkich użytkowników zalogowanych do pokoju dotrze wiadomość o nowym pliku dostępnym do pobrania. Na czacie zostanie napisana wiadomość z informacją kto kiedy jaki plik zamieścił, tak jak widać na poniższym rysunku.



Rysunek . Widok czatu pokój2 po otrzymaniu pliku przez serwer

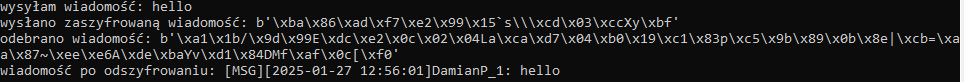
Drugi użytkownik jest w stanie pobrać plik, wybierając go z listy dostępnych plików, a następnie klikając przycisk „Pobierz plik”. Użytkownik zostanie poinformowany o pobieraniu poprzez zmianę napisu status pobierania z „oczekiwanie” na „pobieranie”.



Rysunek . Widok użytkownika po wybraniu pliku z listy i kliknięciu przycisku

**Szyfrowanie wiadomości:**

Za każdym razem, gdy użytkownik lub serwer wysyła wiadomość, zostaje ona najpierw zaszyfrowana za pomocą klucza AES. Odpowiedzialne za to funkcje przedstawiono w rozdziale 3.2.



Rysunek . Przykład działania algorytmu szyfrującego

Jak widać na rysunku 3.29 klient wysyła do serwera zaszyfrowaną wiadomość „hello”, która dociera do serwera, który rozszyfrowuje ją, dodaje do niej nagłówek „[MSG]” oraz informację z datą i czasem wysłania, a następnie szyfruje ją i wysyła do klientów. Na rysunku widać, że klient otrzymuje wiadomość w postaci zaszyfrowanej, którą rozszyfrowuje za pomocą klucza AES.

## Badania wybranych elementów systemu rozwiązującego problem inżynierski

# Testy systemu informatycznego / Aplikacji

## Testy systemu

## Realizacja testów

### Opis testu 1

### Opis testu 2

## Wyniki testów

# PODSUMOWANIE I WNIOSKI

Bibliografia

1. <https://www.geeksforgeeks.org/client-server-architecture-system-design/>
2. <https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2>

Spis Tabel

[Tabela 3.1 Tabela nagłówków przetwarzanych przez serwer 23](#_Toc188866670)

[Tabela 3.2 Tabela nagłówków przetwarzanych przez aplikację klienta 23](#_Toc188866671)

Spis Rysunków

[Rysunek 2.1 Schemat blokowy systemu 17](#_Toc188866665)

[Rysunek 2.2 Diagram klas systemu 18](#_Toc188866666)

[Rysunek 2.3 Szkic ekranu rejestracji użytkownika 20](#_Toc188866667)

[Rysunek 2.4 Szkic ekranu logowania użytkownika 21](#_Toc188866668)

[Rysunek 2.5 Szkic głównego ekranu aplikacji 22](#_Toc188866669)

Spis Listingów

Nie można odnaleźć pozycji dla spisu ilustracji.

Spis ZałĄczników

1. Kod źródłowy aplikacji – załącznik w APD, plik *Aplikacja\_kod\_zrodlowy.zip*
2. Instrukcja instalacji i uruchomienia aplikacji - załącznik w APD, plik *Aplikacja\_Instrukcja\_instalacji.zip*
3. Płyta DVD z projektem aplikacji, bazą danych i wersją instalacyjna aplikacji.