

Telefonia IP

JoinMe

Autorzy:

Szymon Szczot

numer indeksu: 136317

adres e-mail: szymon.szczot@student.put.poznan.pl

Szymon Wasilewski

numer indeksu: 136323

adres e-mail: szymon.p.wasilewski@student.put.poznan.pl

1.	Charakterystyka ogólna projektu	3
2.	Architektura systemu	3
3.	Wymagania (z podziałem na aktorów)	5
3.1.	Funkcjonalne	5
3.2.	Niefunkcjonalne	5
4.	Narzędzia, środowisko, biblioteki, kodeki	6
5.	Opis najważniejszych protokołów	6
6.	Schemat bazy danych	7
6.1.	Model relacyjny	7
7.	Diagramy UML	8
7.1.	Diagram przypadków użycia	8
7.2.	Diagram sekwencji	9
7.3.	Diagram stanów	10
8.	Projekt interfejsu graficznego	11
9.	Najważniejsze metody i fragmenty kodu aplikacji	13
10.	Testy i przebieg sesji	17
11.	Analiza bezpieczeństwa	18
12.	Podsumowanie	18
12.1.	Podział prac	18
12.2.	Cele zrealizowane, cele niezrealizowane, napotkane problemy	18
12.3.	Perspektywa rozwoju	19

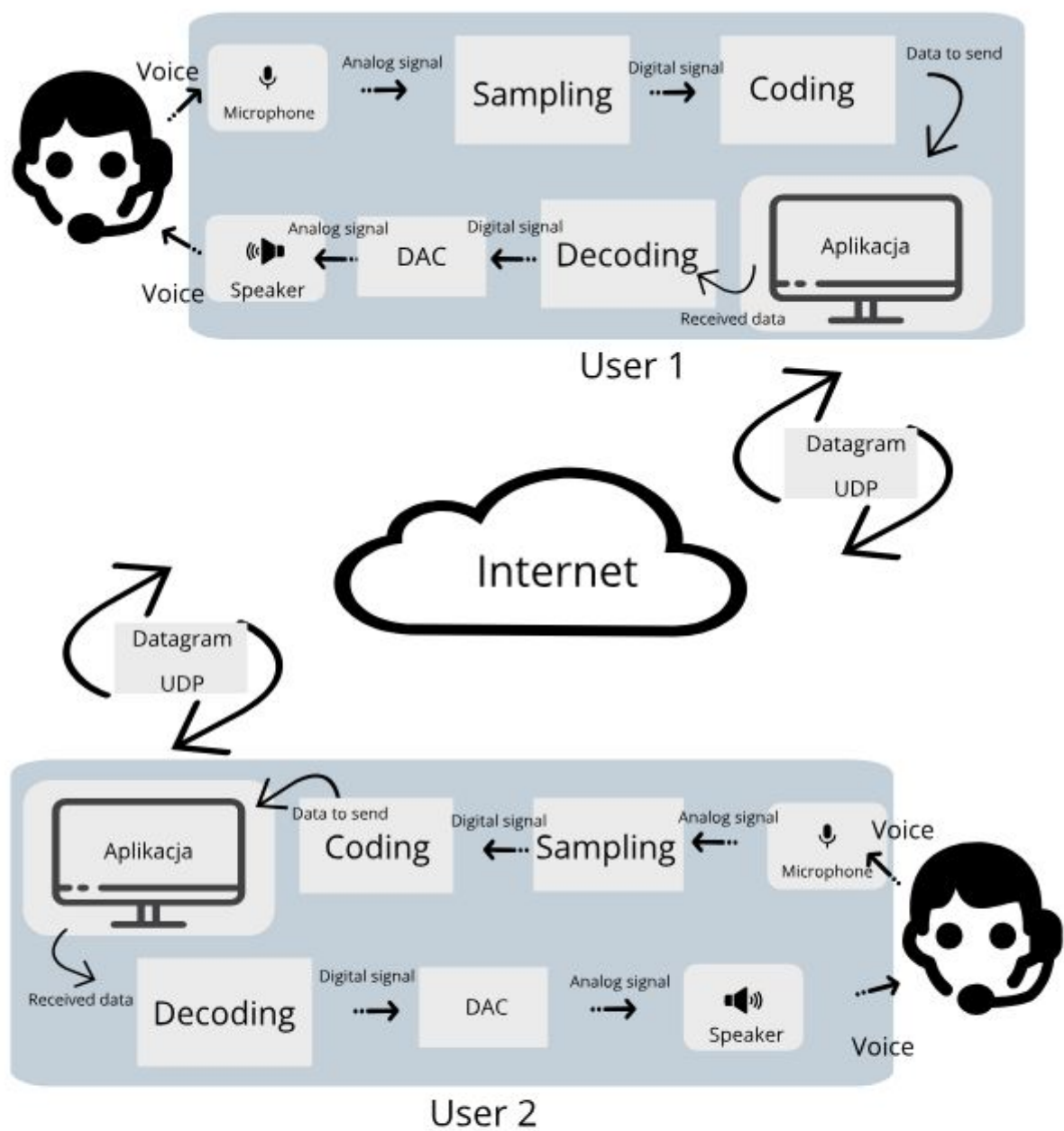
1. Charakterystyka ogólna projektu

Celem projektu jest opracowanie bezpiecznego systemu komunikacji głosowej w sieci IP (VoIP) wraz z jego implementacją. Aplikacja będzie przysyłać sygnały cyfrowe przekonwertowane z analogowych (głos) między dwoma użytkownikami i odtwarzała je odpowiednim klientom. Połączenie będzie kodowane dla zwiększenia bezpieczeństwa.

2. Architektura systemu

System wykorzystuje połączenie UDP między dwoma użytkownikami aplikacji do przesyłania mowy. Za odebranie sygnału z wejścia audio odpowiada aplikacja, przy pomocy wbudowanych bibliotek. Sygnał jest następnie próbkowany i przetwarzany do wersji cyfrowej - za ten etap odpowiada biblioteka PyAudio. Kolejnym krokiem jest kodowanie sygnału w taki sposób, aby możliwe było jego przesłanie do odbiorcy - aplikacja używa do tego gotowych kodeków. Ostatnim krokiem wykonanym przez nadawcę jest zapakowanie zakodowanego sygnału do datagramu UDP i przesłanie do odbiorcy.

Odbiorca wykonuje przeciwstawne kroki w odwrotnej kolejności - po odebraniu datagramu aplikacja odbiorcy rozpakowuje go, uzyskując zakodowany sygnał. Następnie dekoduje sygnał, przy użyciu tych samych kodeków. Kolejnym krokiem jest przetworzenie sygnału w taki sposób aby możliwe było jego odtworzenie na analogowym wyjściu, np. głośnikach.



Rys. 1 - schemat architektury systemu

3. Wymagania (z podziałem na aktorów)

3.1. Funkcjonalne

- Nadawca:
 - Zadzwoń

Opis: Użytkownik wpisuje numer telefonu z którym chce nawiązać połączenie.

Dane wejściowe: Numer telefonu

Źródło danych wejściowych: Użytkownik, lub baza danych (kontakty)

Przeznaczenie: Wykorzystywane do nawiązania połączenia

 - Przeszukaj książkę telefoniczną
 - Zapisz kontakt
- Odbiorca:
 - Odbierz

Opis: Użytkownik wybiera czy chce odebrać połączenie czy odrzucić

Przeznaczenie: Pozostawia decyzję o reakcji na zdarzenie użytkownikowi

 - Odrzuć

3.2. Niefunkcjonalne

- System przystępny dla każdego użytkownika
- Responsywny interfejs
- Maksymalne ograniczenie opóźnień
- Jasne komunikaty o wystąpieniu błędu

4. Narzędzia, środowisko, biblioteki, kodeki

System zostanie napisany w języku Python. Przy pracy zostanie wykorzystane IDE PyCharm Professional w wersji: 2019.2. Algorytmy i protokoły wykorzystane w systemie będą napisane przez autorów, a wykorzystane w nich będą biblioteki wymienione poniżej:

- PyAudio - biblioteka udostępniająca metody pozwalające zapisywać i odtwarzać dźwięki
- socket - biblioteka standardowa Python, umożliwia tworzenie i zarządzanie połączeniami UDP
- Django - framework, używany do obsługi serwera i bazy danych (ORM)
- SQLite - baza danych przechowująca dane użytkowników

Do kodowania sygnałów wykorzystany zostanie kodek MP3.

5. Opis najważniejszych protokołów

- UDP - User Datagram Protocol - Beipołączeniowy protokół komunikacyjny. Jest używany przy połączeniach, które stawiają prędkość nad niezawodnością. Wysokie prędkości są uzyskiwane dzięki pomijaniu oczekiwania na potwierdzenie przez odbiorcę otrzymania wiadomości.
- Własny protokół binarny, składający się z zakodowanych binarnych reprezentacji tekstów oznaczających działania aplikacji.

KOD	Wypełnienie
1 do 8B	99 - 92B

Tab. 1 - budowa protokołu komunikacyjnego

Znaczenie kodów (należy zapamiętać, że Python wszystkie z nich przetwarza w reprezentacji bajtowej):

- 'invite' - zaproszenie do połączenia
- 'accepted' - przyjęcie zaproszenia
- 'no' - odrzucenie zaproszenia
- 'close' - zamknięcie połączenia

6. Schemat bazy danych

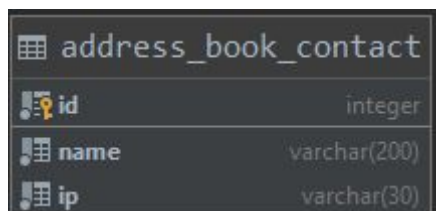
Do zastosowań aplikacji wystarczy bardzo prosta tabela składająca się z dwóch pól.

6.1. Model relacyjny

Baza danych została maksymalnie uproszczona ze względu na oszczędność pamięci, zwiększenie szybkości działania a także jest ona zwyczajnie wystarczająca do celów do których jest wykorzystywana.

Tabela w bazie danych składa się z pól:

- id - indywidualny identyfikator wpisu do książki adresowej
- name - nazwa użytkownika rozpoznawalna przez człowieka
- ip - adres ip użytkownika rozpoznawalny dla komputera i aplikacji



The image shows a diagram of a database table named 'address_book_contact'. It lists three fields: 'id' of type 'integer', 'name' of type 'varchar(200)', and 'ip' of type 'varchar(30)'. Each field is preceded by a small icon representing its data type.

address_book_contact	
id	integer
name	varchar(200)
ip	varchar(30)

Rys. 2 - tabela przechowująca informacje o kontaktach

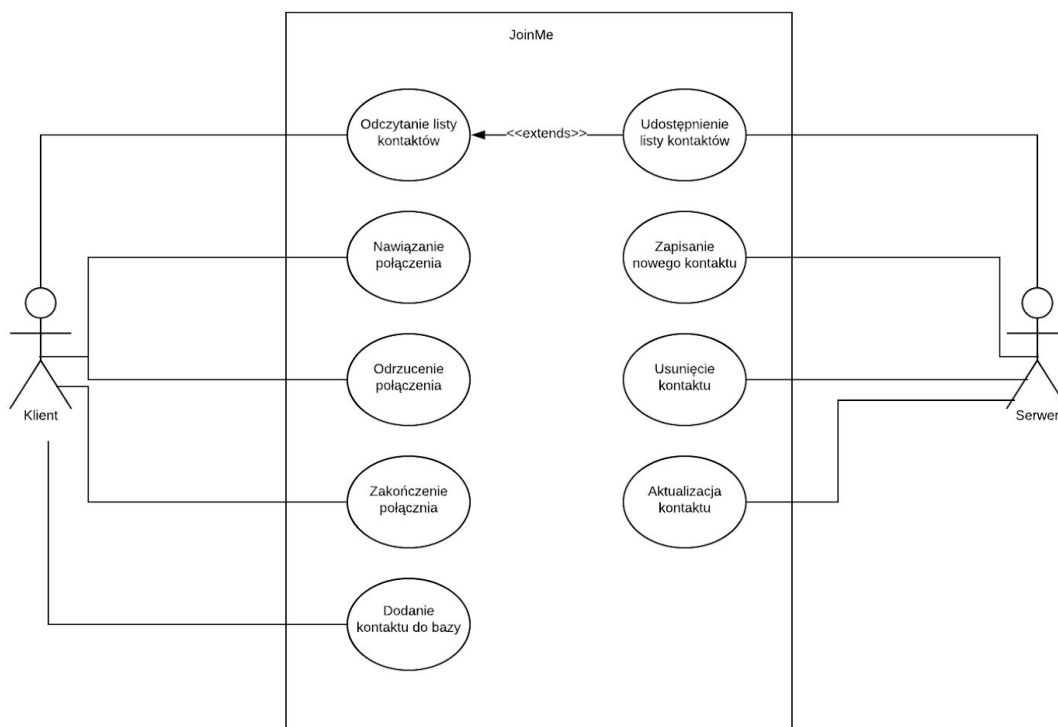
7. Diagramy UML

W tym punkcie zostaną przedstawione diagramy UML pokazujące działanie systemu z różnych perspektyw.

7.1. Diagram przypadków użycia

Na diagramie widocznym na rysunku 3. przedstawione zostały funkcjonalności możliwe do wykonania przez jednego z dwóch głównych aktorów:

- po stronie użytkownika:
 - odczytywanie listy kontaktów
 - nawiązanie połączenia
 - odrzucenie połączenia
 - zakończenie połączenia
 - dodanie kontaktu do bazy
- po stronie serwera:
 - udostępnienie listy kontaktów użytkownikom
 - zapisanie nowego kontaktu do bazy
 - usunięcie kontaktu z bazy
 - aktualizacja danych kontaktu

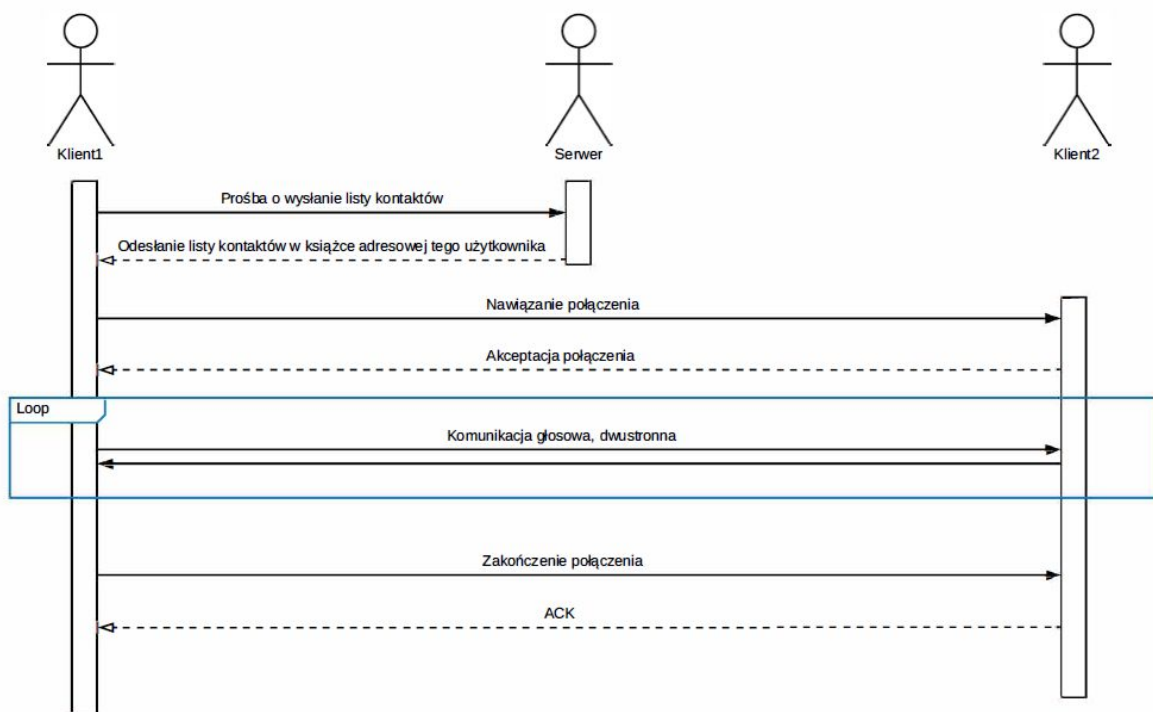


Rys. 3 - diagram przypadków użycia

7.2. Diagram sekwencji

Na diagramie widocznym na rysunku 4. przedstawiono sekwencje występujące w scenariuszu nawiązania i przeprowadzenia rozmowy głosowej.

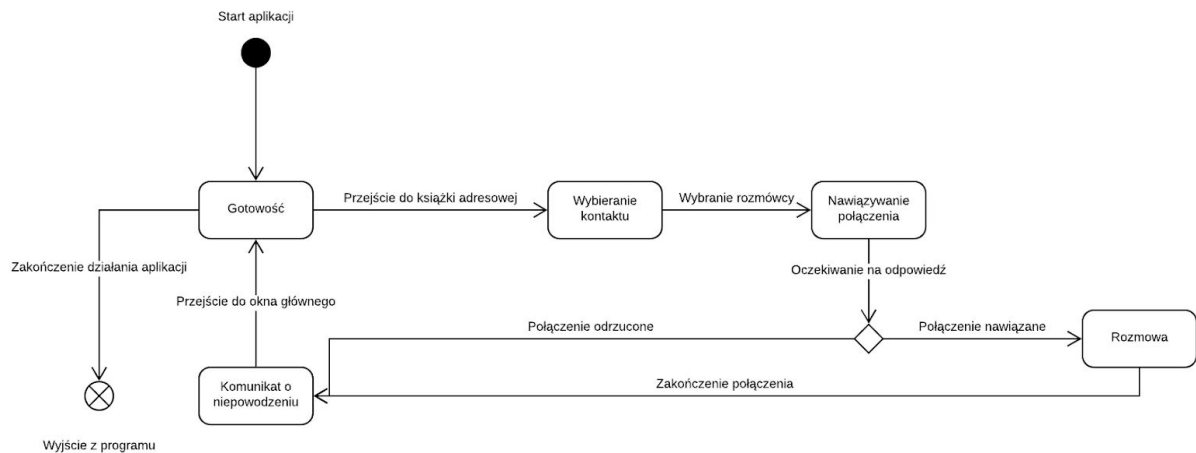
- Klient1 wysyła zapytanie do serwera o swoją książkę adresową zapisaną w bazie danych
- Serwer odsyła listę zapisanych kontaktów
- Klient1 wybiera kontakt i nawiązuje połączenie z użytkownikiem Klient2
- Klient2 w tym scenariuszu akceptuje połączenie
- pomiędzy klientami, w pętli, przebiega dwustronna rozmowa
- Klient1 inicjuje zakończenie połączenia
- Klient2 wysyła ACK i kończy połączenie.



Rys. 4 - diagram sekwencji

7.3. Diagram stanów

Na diagramie widocznym na rysunku 5. przedstawiono stany w których może się znaleźć aplikacja po spełnieniu odpowiednich warunków.



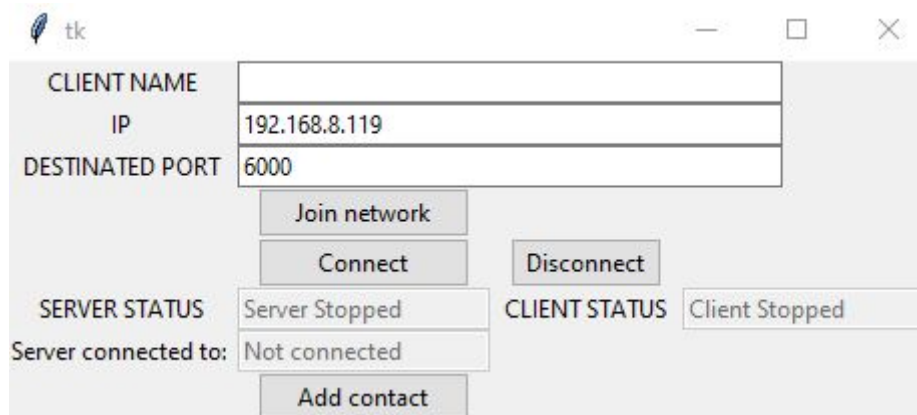
Rys. 5 - diagram stanów

8. Projekt interfejsu graficznego

Do projektu interfejsu graficznego wykorzystano bibliotekę tkinter dla języka programowania Python. Na rysunkach 6, 7 i 8 widać okna połączenia w różnych stanach działania aplikacji.

Opis poszczególnych elementów:

- Pola:
 - Pole do wpisywania - "CLIENT NAME" - służy do nadania nazwy kontaktu, który chcemy dodać do książki adresowej, oraz alternatywnie do zadeklarowania portu na którym powinna nasłuchiwać aplikacja
 - Pole do wpisywania - "IP" - służy do wprowadzenia adresu IP osoby z którą chcemy się połączyć, oraz alternatywnie do nadania adresu kontaktu, który chcemy dodać do książki adresowej
 - Pole do wpisywania - "DESTINATED PORT" - służy do wpisania numeru portu na którym ma nasłuchiwać klient
- Przyciski:
 - Przycisk - "Join network" - służy do uruchomienia nasłuchiwania
 - Przycisk - "Connect" - służy do zainicjowania połączenia
 - Przycisk - "Disconnect" - służy do przerywania połączenia
- Pola statusu:
 - Pole - "SERVER STATUS" - Pokazuje obecny status nasłuchiwania
 - Pole - "CLIENT STATUS" - Pokazuje obecny status połączenia
 - Pole - "Server connected to" - Pokazuje adres osoby z którą obecnie nawiązane jest połączenie.



Rys. 6 - ekran główny aplikacji

CLIENT NAME	6000
IP	192.168.8.119
DESTINATED PORT	6000

Join network

Connect Disconnect

SERVER STATUS: Connected CLIENT STATUS: Client Stopped

Server connected to: Not connected

Add contact

Rys. 7 - ekran po rozpoczęciu nasłuchiwania

CLIENT NAME	6000
IP	192.168.0.21
DESTINATED PORT	6000

Join network

Connect Disconnect

SERVER STATUS: Connected CLIENT STATUS: Client connected to 19

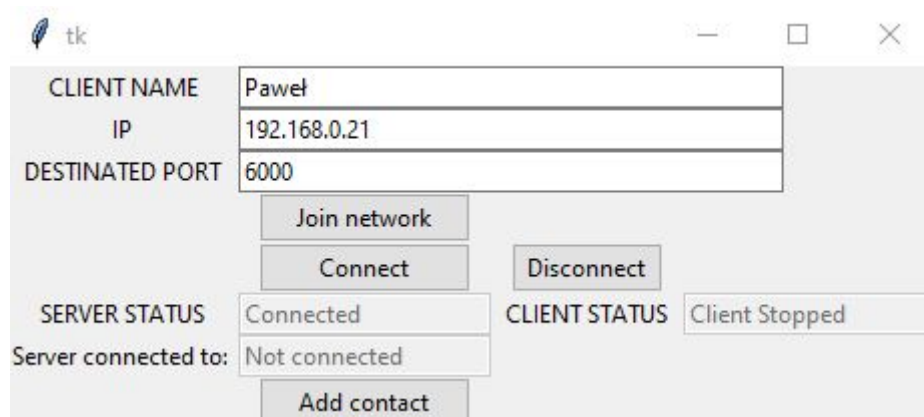
Server connected to: Not connected

Add contact

Rys. 8 - ekran po nawiązaniu połączenia

Na rys. 9. pokazano przykład dodawania kontaktu do książki adresowej znajdującej się na serwerze. W polu "CLIENT NAME" została wpisana nazwa pod jaką zapisany zostanie kontakt o IP wpisanym w polu "IP".

Pole "CLIENT NAME" zostało wykorzystane do tego celu ze względu na zachowanie kompaktowości aplikacji, próbę ułatwienia użytkownikowi zapisanie informacji oraz ze względu na fakt, że dodawanie kontaktów docelowo powinno zostać przeprowadzone przez osobę zarządzającą częścią serwerową aplikacji.



Rys. 9 - ekran przy dodawaniu kontaktu

9. Najważniejsze metody i fragmenty kodu aplikacji

Aplikacja składa się z dwóch podstawowych klas:

- server
- client

Najważniejsze metody w tych klasach to metody odpowiedzialne za nawiązanie i przebieg połączenia.

Metoda odpowiedzialna za połączenie po stronie klienta została przedstawiona na rys. 10. Rozpoczynamy od wysłania prośby o połączenie czym zajmuje się metoda "handshake". Po otrzymaniu odpowiedzi sprawdzamy czy jest to akceptacja czy odrzucenie i zgodnie z tym zwracamy True lub False.

Jeśli połączenie zostało ustanowione wyświetlamy również komunikat o udanym połączeniu.

```
def connect(self, target_ip):
    answer = self.handshake(target_ip)
    if answer == constants.ACCEPTED:
        self.variables.client_status.set("Client connected to " + target_ip)
        return True
    else:
        return False
```

Rys. 10 - metoda connect

Metoda odpowiadająca za przebieg połączenia po stronie klienta została przedstawiona na rys. 11.

Na starcie deklarujemy wątki, które będą odpowiedzialne za obsługę nagrywania i wysyłania dźwięku do serwera, oraz odbierania i odtwarzanie dźwięku z serwera. Następnie wyświetlany jest stosowny komunikat i połączenie rozpoczynamy wystartowaniem wątków.

Na rys. 11. pokazana jest część z odbieraniem danych z serwera a na rys. 12 część z wysyłaniem danych.

Zasadniczo działanie funkcji jest bardzo proste. Wątek działa w pętli, dopóki aplikacja nie zamknie połączenia (co wiąże się z ustawieniem zmiennej `stop_threads` na `True`). W jednym przebiegu pętli aplikacja odbiera 20 000 bajtów danych i zapisuje je do `playing_stream`. `playing_stream` jest to obiekt utworzony przy pomocy biblioteki `PyAudio` który umożliwia odtwarzanie dźwięku z głośników.

```
def start_communication(self):
    self.receive_thread = threading.Thread(target=self.receive_server_data)
    self.send_thread = threading.Thread(target=self.send_data_to_server)

    print(bcolors.OKBLUE + "Connection established" + bcolors.ENDC)

    self.receive_thread.start()
    self.send_thread.start()

def receive_server_data(self):
    while not self.stop_threads:
        self.receive_and_play()

def receive_and_play(self):
    data = self.client_socket.recv(20000)
    self.playing_stream.write(data)
```

Rys. 11 - metody obsługujące przebieg połączenia po stronie klienta

```
def send_data_to_server(self):
    while not self.stop_threads:
        self.send_and_record()

def send_and_record(self):
    data = self.recording_stream.read(20000)
    self.client_socket.send(data)
    time.sleep(0.01)
```

Rys. 12 - metody obsługujące wysyłanie danych na serwer

Metoda odpowiedzialna za rozpoczęcie połączenia po stronie serwera została przedstawiona na rys. 13.

Zaczynamy od wyświetlenia statusu, tj. port i IP na którym nasłuchuje serwer, np. żeby można było go łatwo dodać do książki adresowej.

Następnie czekamy na połączenie i jeśli jakieś nadejdzie:

- odbieramy powitanie
- jeśli powitanie jest zgodne z protokołem wyświetlamy popup pozwalający zdecydować, czy chcemy przyjąć/odrzuć połączenie
- jeśli postanowiliśmy przyjąć połączenie rozpoczynamy obsługę klienta
- jeśli zdecydujemy się nie przyjmować połączenia serwer wraca do stanu początkowego, tj. oczekiwania na połączenie.

```
def accept_connection(self):
    self.server_socket.listen(100)

    print(bcolors.OKGREEN + "Accepting new connections on:" + bcolors.ENDC)
    print(bcolors.BOLD + "IP: " + self.ip + bcolors.ENDC)
    print(bcolors.BOLD + "Port: " + str(self.port) + bcolors.ENDC)

    self.client, addr = self.server_socket.accept()
    if addr:
        inviting_message = self.client.recv(100)
        if inviting_message == constants.INVITE:
            self.raise_popup(f"{addr} is calling")

    if self.server_started:
        self.variables.connected_to.set(addr)
        self.handle_client_connection()
    else:
        self.accept_connection()
```

Rys. 13 - metoda accept_connection

Na rys. 14. przedstawiono sposób obsługi klienta przez serwer po ustanowieniu połączenia.

Podobnie jak w przypadku modułu klienta zaczynamy od deklaracji wątków oraz wyświetlenia komunikatu. Następnie startujemy wątki. Największą różnicą między serwerem a klientem jest to, że tylko klient może się rozłączyć. Jest to spowodowane zmianami w koncepcji przedstawionymi w podsumowaniu.

Rozłączanie przebiega w następujący sposób: jeżeli po odebraniu danych serwer stwierdzi, że klient wysłał zgodnie z protokołem prośbę o zamknięcie, serwer powinien przerwać połączenie.


```

def handle_client_connection(self):
    print(bcolors.OKBLUE + "Handling new connection" + bcolors.ENDC)
    self.receive_from_client_thread = threading.Thread(target=self.receive)
    self.send_to_client_thread = threading.Thread(target=self.send)

    self.receive_from_client_thread.start()
    self.send_to_client_thread.start()

def receive(self):
    while self.server_started:
        data = self.client.recv(20000)
        if data == constants.CLOSE:
            self.disconnect_server()
            self.playing_stream.write(data)

def send(self):
    while self.server_started:
        send_data = self.recording_stream.read(20000, exception_on_overflow=False)
        self.client.send(send_data)
        time.sleep(0.01)

```

Rys. 14 - obsługa połączenia z klientem

W przypadku aplikacji docelowego serwera najważniejsza część kodu to ta powiązana z jego główną funkcją: rezolucją adresów IP użytkowników. Widok obsługujący tę funkcję przedstawiony jest na rys. 15 - dzięki wzorcowi projektowemu Django jest to bardzo prosta metoda.

Zaczynamy od odseparowania adresu IP z query params przesłanych za pomocą metody GET protokołu HTTP. Następnie sprawdzamy, czy taki kontakt istnieje w naszej bazie danych przy pomocy ORM oferowanego przez Django. Jeśli istnieje - odsyłamy komunikat o znalezieniu adresu wraz z niezbędnymi informacjami. Jeśli nie istnieje - odsyłamy stosowny komunikat o błędzie.

```

class IP(View):
    def get(self, request, *args, **kwargs):
        query_string = request.environ["QUERY_STRING"]
        name = re.sub(".*=", "", query_string)
        try:
            contact = Contact.objects.get(name=name.strip())
            return JsonResponse({"name": contact.name, "ip": contact.ip})
        except Contact.DoesNotExist:
            return JsonResponse({"Not found": "Error"})

```

Rys. 15. - metoda obsługująca wysyłanie kontaktów

10. Testy i przebieg sesji

Wszystkie testy przebiegały w sieci lokalnej. Poniżej wyjaśnienie nazw użytych w opisie przebiegu sesji:

- serwer - aplikacja w trybie serwera
- klient - aplikacja w trybie klienta
- DB - aplikacja do przechowywania kontaktów

Przykładowy przebieg sesji:

- Start serwera - osoba, która chce udostępnić możliwość rozmowy ze sobą musi rozpocząć działanie w trybie serwer ("Join network").
- Start klienta - osoba która chce połączyć się z inną osobą musi wystartować aplikację w trybie klienta.
- Ustanowienie połączenia.
 - Odpytanie DB z książką adresową, o to czy zawiera wpis o osobie podanej przez klienta.
 - Jeśli zawiera - DB odsyła adres IP, do którego powinna odwoływać się aplikacja klienta
 - Jeśli nie zawiera: DB odsyła komunikat o błędzie
 - Przetworzenie odpowiedzi DB
 - Jeśli DB zwrócił IP - klient rozpoczyna połączenie
 - Jeśli DB nie zwrócił IP - klient próbuje połączyć się z osobą wpisaną w pole IP jako rzeczywiste IP, bez translacji
 - Rozpoczęcie połączenia przez wysłanie komunikatu INVITE klient - serwer
 - Przyjęcie ("Accept") lub odrzucenie ("Decline") połączenia przez serwer
 - Zapętłona, dwukierunkowa transmisja głosu pomiędzy klientem, a serwerem
 - Zakończenie połączenia przez klienta poprzez wysłanie komunikatu CLOSE

Testy pokazały, że widoczność serwera kontaktów (DB) jest na tym etapie rozwoju aplikacji konieczna do wykonania jakiegokolwiek połączenia. Z kolei on sam jest widoczny tylko dla lokalnej maszyny, na której jest włączony.

Podczas próby zakończenia połączenia przez klienta, jego aplikacja zawieszała się. Samo połączenie głosowe było jednak pomyślnie przerywane. Najprawdopodobniej wynika to z próby zamknięcia odpowiednich wątków, aby powrócić do stanu sprzed nawiązania połączenia i słabej obsługi tej operacji przez wykorzystaną bibliotekę.

11. Analiza bezpieczeństwa

Aplikacja do komunikacji głosowej powinna być względnie bezpieczna z powodu niebezpieczeństwa wynikającego z przechwycenia transmisji przez osoby postronne. Z tego powodu aplikacja JoinMe wykorzystuje kodowaną komunikację klient-serwer.

Ze względu na wykorzystany sposób kodowania transmisji, nie istnieje mechanizm szyfrowania wysyłanych danych, przez co w tej chwili nie zaleca się wykorzystania aplikacji do celów poufnych, przekazywania informacji wrażliwych, lub o dużym znaczeniu.

12. Podsumowanie

12.1. Podział prac

- Szymon Szczot - aplikacja kliencka, zbieranie i odtwarzanie audio
- Szymon Wasilewski - serwer, baza danych, testy aplikacji

12.2. Cele zrealizowane, cele niezrealizowane, napotkane problemy

Cele zrealizowane:

- napisanie aplikacji pozwalającej na komunikację głosową dwukierunkową za pomocą protokołu IP - w trakcie prac koncepcja ewoluowała, czego efektem jest aplikacja umożliwiająca (na przykład) firmie obsługującej zgłoszenia użytkowników łatwy, a przede wszystkim kontrolowany kontakt z klientem. Aplikacja wskazuje na takie rozwiązanie za pomocą:
 - braku możliwości rozłączenia się przez obsługującego - jest to częsta praktyka w takich rozwiązaniach,
 - Tryb aplikacji klient-serwer - serwerowi (osobie obsługującej zgłoszenia) nie jest potrzebna możliwość nawiązywania połączeń, a osobie dzwoniącej nie potrzeba możliwości odbierania połączeń.

Cele niezrealizowane:

- uwierzytelnianie użytkownika - utworzenie konta użytkownika aby połączyć się z konsultantem wydaje się niepotrzebnym utrudnieniem kontaktu, jednak w tej chwili nie istnieje żaden system uwierzytelniania klienta, przez co użytkownik przyjmujący połączenie jest w stanie wywnioskować tożsamość klienta tylko po adresie IP

- sygnalizowanie zajętego łącza - użytkownik, który nie może się połączyć powinien otrzymywać informację o zajętych łączach. Można to rozwiązać np. statusami użytkowników-serwerów na osobnej platformie, jak i w aplikacji
- rozłączanie przez serwer - powinna istnieć możliwość przerwania połączenia w przypadku braku odpowiedzi od klienta, jak i braku chęci kontynuowania połączenia
- połączenie przez Internet - w tej chwili aplikacja działa tylko na maszynach w jednej sieci lokalnej
- dostępność serwera kontaktów - serwer Django jest uruchamiany na maszynie lokalnej użytkownika i widziany jest również tylko przez niego. Można wystawić ten serwer jako osobną usługę sieciową, dzięki czemu każdy użytkownik sieci lokalnej miałby do niej dostęp

12.3. Perspektywa rozwoju

Aplikacja jest dobrym początkiem dla rozwoju projektu komunikacji pomiędzy konsultantem, a użytkownikiem. Z pomysłów na dalszy rozwój najważniejszym byłoby rozdzielenie aplikacji na dwie: klienta i serwera - w tym momencie użytkownik i tak może pełnić funkcję tylko jednej ze stron, a w przyszłości mógłby mieć tylko funkcje jemu potrzebne.

Przydatną funkcją godną zaimplementowania jest wybieranie kontaktu z listy przesłanej przez serwer. Każdy konsultant mógłby mieć też swój obecny status zapisany w bazie, który wyświetlany byłby użytkownikowi-klientowi, dzięki czemu mógłby łatwo wybrać, z kim chce się skontaktować.

Najwyższym priorytetem dla rozwoju aplikacji jest natomiast zaimplementowanie szyfrowania transmisji głosowej, aby uchronić użytkowników przed wyciekiem poufnych/wrażliwych informacji poprzez ataki typu Man-in-the-middle.