Politechnika Poznańska
Wydział Informatyki i Telekomunikacji

Telefonia IP

Dokumentacja projektu - aplikacja myTurn

Autor	Indeks	E-mail
Kornelia Maik	135465	kornelia.maik@student.put.poznan.pl
Patryk Wenz	136325	patryk.wenz@student.put.poznan.pl

Kwiecień 2020

Spis treści

1 Charakterystyka projektu

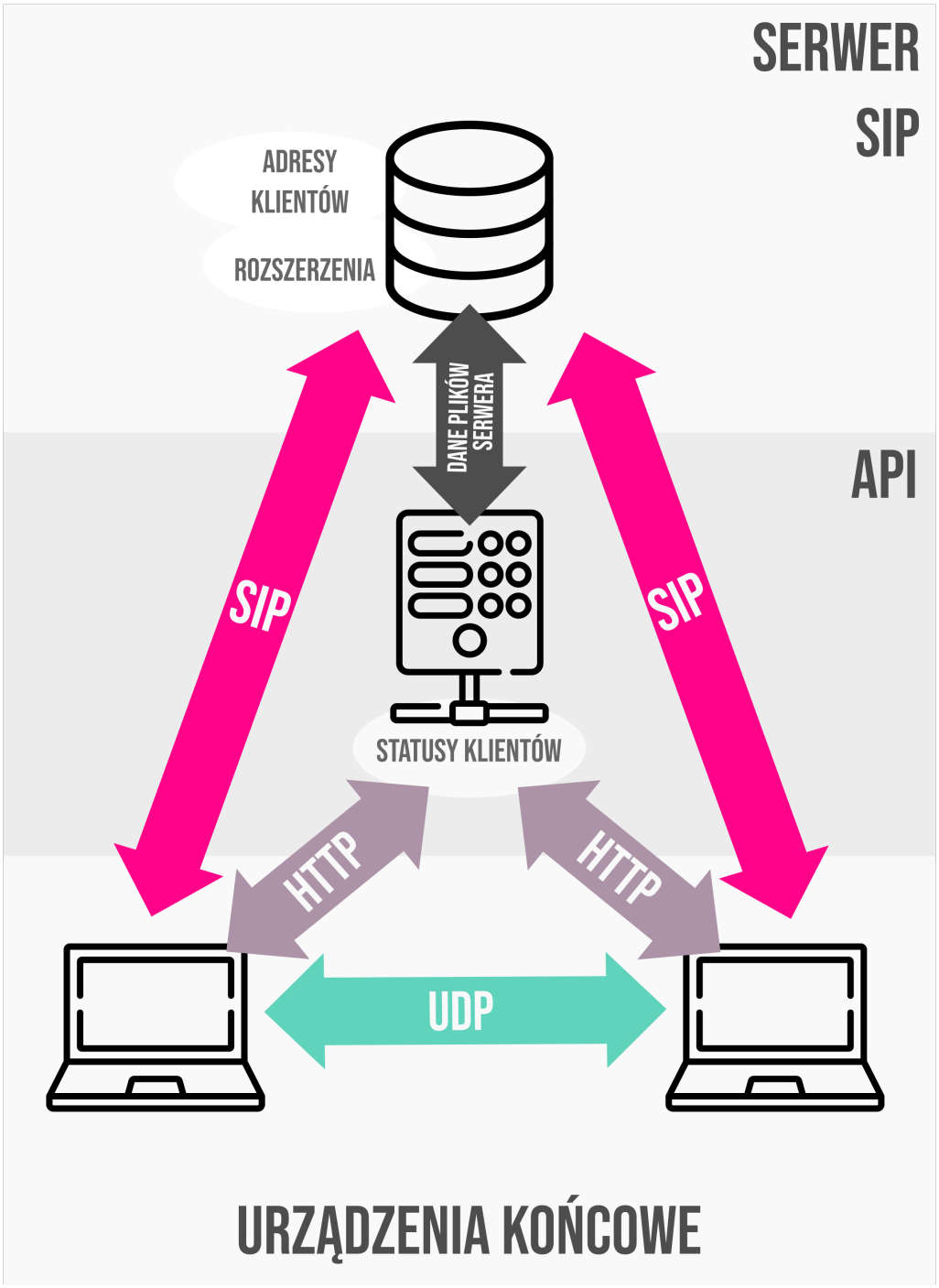
Celem projektu jest stworzenie systemu telefonii IP umożliwiającego rozmowę głosową i przesyłanie tekstu pomiędzy użytkownikami końcowymi. Aplikacja myTurn ma przypominać ideę rozmów w ciemno, tzn. zalogowany użytkownik łączony będzie losowo z innym zalogowanym użytkownikiem, a po zakończeniu sesji użytkownik może wyjść z systemu lub wylosować kolejną rozmowę.

2 Architektura systemu

2.1 Ogólna budowa

Używany w projekcie protokół SIP ma architekturę klient-serwer. Do komunikacji tekstowej i głosowej używany jest protokół UDP i komunikacja odbywa się pomiędzy urządzeniami końcowymi. Ogólną budowę systemu przedstawia Rysunek 1. Topologię logiczną systemu tworzą następujące elementy składowe:

- **oprogramowanie klienta** - odpowiedzialne za wysyłanie żądań SIP oraz pakietów związanych z komunikacją tekstową i głosową,
- **serwer SIP** - realizuje żądania SIP otrzymywane od klientów, zawiera bazę danych z aktualnymi adresami klientów i odwzorowaniami adresów SIP na adresy IP,
- **API** - zmienia zawartość plików serwera SIP przy rejestracji i wylogowaniu klienta, przechowuje statusy klientów, losuje rozmówców i pośredniczy w uzyskaniu przez klientów danych o swoich rozmówcach.



Rysunek 1: Ogólna budowa systemu (źródło: rysunek wykonany przez autorkę)

2.2 Identyfikatory i komunikaty

2.2.1 SIP

Zasobami rezerwowanymi w sesji SIP są zasoby sieciowe, które wymagają identyfikacji. Identyfikatorem tych zasobów jest SIP URI (SIP Uniform Resource Indicator) składający się z nazwy użytkownika i hosta.

W projekcie wykorzystane są cztery typy **żądań**:

- REGISTER służy do zarejestrowania adresu w serwerze SIP.
- INVITE wskazuje, że użytkownik/usługa jest zaproszona do nawiązania sesji. Zawiera opis sesji, do której uczestnik jest zapraszany.
- ACK potwierdza, że klient odebrał finalną odpowiedź na żądanie INVITE.
- BYE wysyłane przez klienta UAC w celu zasygnalizowania serwerowi żądania zakończenia komunikacji.

W **odpowiedziach** wyróżnia się sześć typów kodów statusu:

- 1xx: Provisional – żądanie odebrane i przekazane do dalszej realizacji.
- 2xx: Success – potwierdzenie ACK, wskazuje, że żądanie zostało przyjęte, prawidłowo odczytane i zaakceptowane.
- 3xx: Redirection – realizacja żądania wymaga dalszy działań.
- 4xx: Client Error – błąd po stronie klienta.
- 5xx: Server Error – serwer nie był w stanie zrealizować żądania.
- 6xx: Global Failure – żądanie nie może być zrealizowane przez serwer.

2.2.2 API

W celu zapewnienia komunikacji między klientem a serwerem wykorzystujemy zapytania **HTTP** do *API - GET, POST, PUT, DELETE*. Odpowiedzi *API* uzależnione są od zdarzeń na danym punkcie końcowym:

- '/add-user-to-sip-exten-conf/<string:user_name>', methods=['POST']

  Odpowiedzi:

  - **200** - jeśli utworzono poprwanie użytkownika.
  - **404** - jeśli użytkownik o danej nazwie już istnieje.

- '/sip-file-lookup', methods=['GET']

  Odpowiedzi:

  - **200, <dane_z_pliku>** - potwierdzenie wysłania pliku.

- '/user-status', methods=['GET'

  Odpowiedzi:

  - **200, <status_uzytkownika>** - potwierdzenie wysłania statusu użytkownika.

- '/pair-status', methods=['GET']

  Odpowiedzi:

  - **200, <list_par>** - potwierdzenie wysłania par.

- '/update-status-rdy/<string:user_name>'

  Odpowiedzi:

  - **200, "Status Changed" : "Rdy"** - potwierdzenie zmiany statusu użytkownika.

- '/update-status-busy/<string:user_name>'

  Odpowiedzi:

  - **200, "Status Changed" : "BUSY"** - potwierdzenie zmiany statusu użytkownika.

- '/update-status-rdy-to-talk/<string:user_name>'

  Odpowiedzi:

  - **200, "Status Changed" : "Rdy TO TALK"** - potwierdzenie zmiany statusu użytkownika.

- '/get-peer/<string:user_name>', methods=['GET']

  Odpowiedzi:

  - **226, "You are not rdy yet": ""** - konieczność zmiany statusu.
  - **208, "Queue is empty": ""** - brak klientów chętnych do rozmowy.
  - **202, "call": <user_name>, "exten": number, "ip_to_send_data": ip_addr** - wylosowano partnera, poczekaj na połączenie.
  - **239, "call": <user_name>, "exten": number, "ip_to_send_data": ip_addr** - wylosowano partnera, zadzwoń.

- '/delete-user/<string:user_name>', methods=['DELETE']

  Odpowiedzi:

  - **200, Deleted": <user_name>, "PAIRS": <PAIRS>, "DIAL": <DIAL_NUMBERS>, "USERS": <USERS>** - potwierdzenie usunięcia użytkownika.

2.3 Parametry sesji i urządzeń

Plik *sip.conf* serweru API określa **parametry konfiguracyjne sesji** zarządzanych przez serwer. Zespół określił parametry w pliku tak, aby były zgodne z działaniem biblioteki *most-voip*.

- context=public - domyślny kontekst połączeń przychodzących
- allowoverlap=no - wyłącza wsparcie dla nakładających się połączeń
- udpbindaddr=0.0.0.0 - adres IP dla nasłuchujących socketów UDP
- tcpenable=no - nie pozwala serwerowi na przychodzące połączenia TCP
- tcpbindaddr=0.0.0.0 - adres IP dla serwera TCP
- transport=udp - domyślny transport
- srvlookup=yes - włącza lookup DNS SRV dla połączeń wychodzących
- qualify=yes - oznacza to, że serwer SIP wysyła co 2 sekundy komendę opcji SIP by sprawdzić, czy urządzenie jest nadal online
- callcounter=yes - umożliwia poznanie większej liczby stanów urządzenia, niż domyślne bezczynny i niedostępny

Dodatkowo, plik określa **parametry urządzeń** znanych serwerowi.

- type=friend - oznacza to, że urządzenie może zarówno przyjmować jak i wykonywać połączenia
- secret=12345678abcd - hasło potrzebne do zarejestrowania urządzenia
- host=dynamic - urządzenie musi się zarejestrować w celu pozania jego adresu IP, bo ten nie jest podany statycznie
- context=phones - kontekst, do którego należy klient. To tam należy szukać jego rozszerzenia

3 Wymagania funkcjonalne i niefunkcjonalne

W aplikacji wyróżnia się cztery **typy aktorów**:

- użytkownik
  - niezarejestrowany,
  - zarejestrowany,
- API pośredniczące między użytkownikami i serwerem SIP,
- serwer SIP.

Podział wymagań funkcjonalnych ze względu na aktorów:

Aktor: użytkownik niezarejestrowany

- Zarejestrowanie się przez podanie nazwy.

Aktor: użytkownik zarejestrowany

- Wyrażenie gotowości do rozmowy.
- Wysłanie wiadomości tekstowej.
- Udostępnienie dźwięku z mikrofonu.
- Wyłączenie mikrofonu.
- Zakończenie rozmowy.
- Wylogowanie się.

Aktor: API

- Zarejestrowanie użytkownika.
- Poproszenie użytkownika o inny nick.
- Wylosowanie uczestników do rozmowy.
- Powiadomienie uczestników o danych rozmówcy.
- Zmiana statusu użytkownika.
- Usunięcie użytkownika z serwera SIP.
- Odświeżanie plików serwera SIP.

Aktor: serwer SIP

- Zarejestrowanie użytkownika.
- Prowadzenie listy użytkowników zawierających skojarzenia nicku z adresem IP oraz rozszerzeniem służącym do dzwonienia.

Na aplikacje są nałożone następujące **wymagania niefunkcjonalne**:

- system operacyjny Ubuntu 14 i nowsze,
- użytkownik rozmawia jednocześnie tylko z jednym użytkownikiem,
- ochrona przed podsłuchem sesji za pomocą szyfrowania AES CBC kluczem uzyskanym przez protokół wymiany kluczy Diffiego-Hellmana,
- interfejs użytkownika jest graficzny i pozwala na łatwe posługiwanie się aplikacją. Składa się z okna służącego do zalogowania się/menu głównego/wyszukiwania partnera do rozmowy/prowadzenia rozmowy,
- serwer i moduły obsługujące komunikację SIP wyświetlają wszelkie istotne informacje w konsoli,
- użytkownik posiada sprzęt I/O, tzn. głośnik/słuchawki i mikrofon,

4 Narzędzia, środowisko, biblioteki, kodeki

Język	python2.7, python3
Środowisko	PyCharm, Asterisk
Biblioteki	pjSIP, most-voip, requests, pyaudio, Crypto (Random, AES), base64, hashlib, socket, threading, json, ast, flask
Kodeki	PCM signed 16-bit little-endian

5 Opis najważniejszych protokołów

5.1 Sygnalizacja

Aby implementowany w ramach projektu system nie był zwyczajnym czatem tekstowo-głosowym, zespół zdecydował się na wykorzystanie protokołu sygnalizacyjnego SIP. **SIP** (Session Initiation Protocol) to protokół służący do sygnalizacji w systemach transmisji głosu i obrazu w czasie rzeczywistym. Jego istotną cechą wykorzystywaną w systemie jest określenie lokalizacji drugiej strony komunikacji. Po zalogowaniu przez klienta, w plikach serwera SIP przypisane jest mu rozszerzenie, które służy później do wykonywania do niego połączeń. Klient pragnący zainicjować połączenie z innym klientem musi zadzwonić pod jego rozszerzenie.

Dzięki wykorzystaniu protokołu SIP zespół uzyskał panowanie nad połączeniami prowadzonymi pomiędzy klientami, a konieczność modyfikacji i pobierania danych z pliku serwera SIP oraz obsługiwanie wysyłanych przez niego komunikatów zmusiło zespół do nauki nowych aspektów programowania i utrwaliło wiedzę o projektowaniu API i wielowątkowości.

Protokół SIP współpracuje z protokółem **SDP** (Session Description Protocol) w celu opisu sesji, czyli określeniem informacji, którymi muszą się wymienić strony połączenia na początku sesji,

5.2 Przesył mediów

Transmisja głosu/obrazu odbywa się z użyciem protokołu **UDP**. Po przeanalizowaniu zalet i wad protokołów używanych do przesyłania danych w systemach VoIP zespół zdecydował się na ten protokół z powodu stosunkowo wysokiej jakości przesyłu danych w porównaniu do łatwości jego użycia. Klienci połączeni do portów przesyłają i odbierają od siebie umieszczane na nich porcje danych. Główne zmartwienie towarzyszące przy stosowaniu protokołu UDP, czyli utrata pakietów, nie ma dużego znaczenia w systemie, ponieważ w większości przypadków utrata pakietów nie jest na tyle duża, by wpływać doświadczalnie na jakość przesyłania mediów.

6 Schemat baz danych

W związku z tym, że aplikacja służy do kontaktu losowych osób, których rozmowa powinna pozostać anonimowa żadne dane na temat uzytkowników nie są trwale przechowywane w bazie danych (billingi, dane o użytkownikach np. adres IP, nazwa). Wszelkie informacje usuwane są z serwera (dokładniej plików sip.conf, extensions.conf) po zakończeniu pracy aplikacji klienta. Dane ulotne takie jak obecny status klienta oraz wybrany rozmówca, przechowywane są w zmiennych na serwerze, które mają postać **JSON**. Struktury prezentują się następująco:

- Struktura odpowiadająca za przechowywanie informacji o statusie użytkowników:

```
"""
{user_name : <name>,
    status: <status>}
"""
```

Rysunek 2: Status użytkowników.

- Struktura odpowiadająca za przechowywanie informacji o wylosowanym rozmówcy:

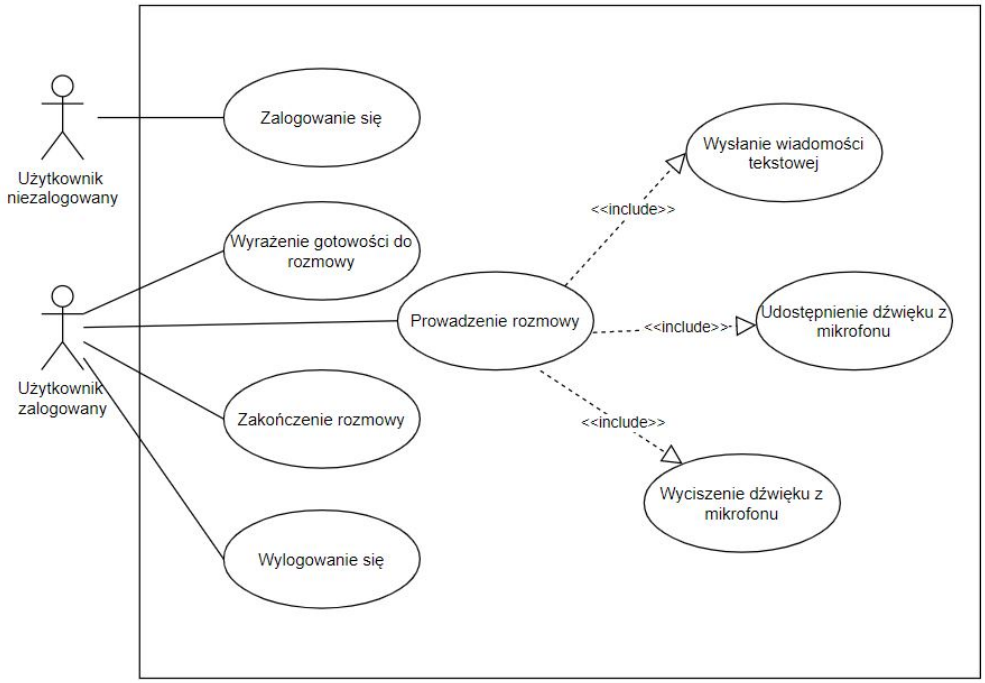
```
"""
{<name> : <name>}
"""
```

Rysunek 3: Pary rozmówców.

Wszelkie inne informacje takie jak adres IP użytkownika pobierane są za pomocą komend, a nastepnie filtrowane i zwracane przez **API** na danym punkcie końcowym.

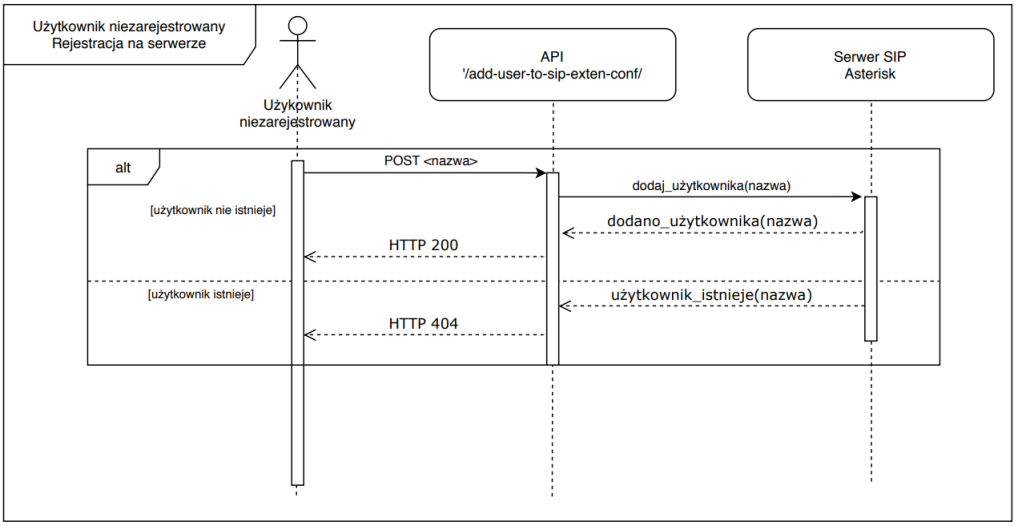
7 Diagramy UML

7.1 Przypadków użycia

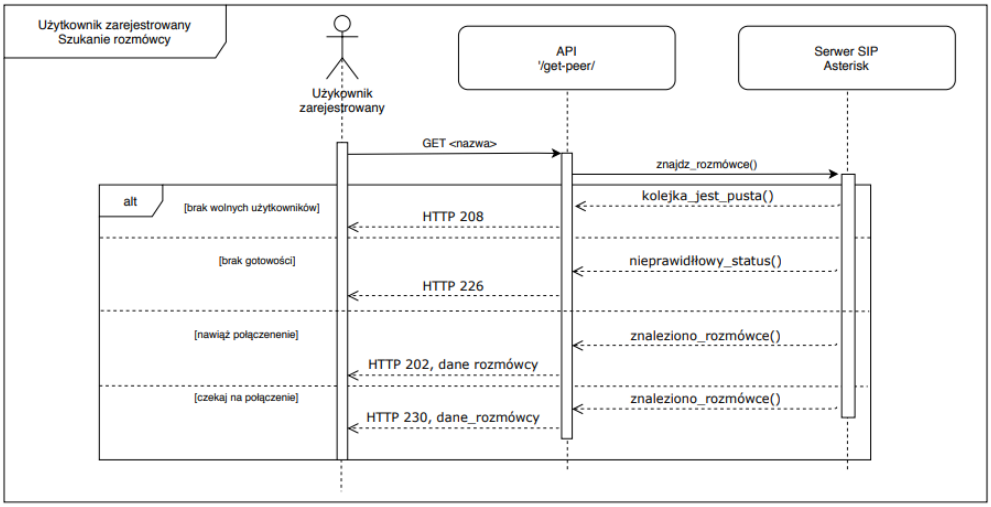


Rysunek 4: Diagram przypadków użycia.

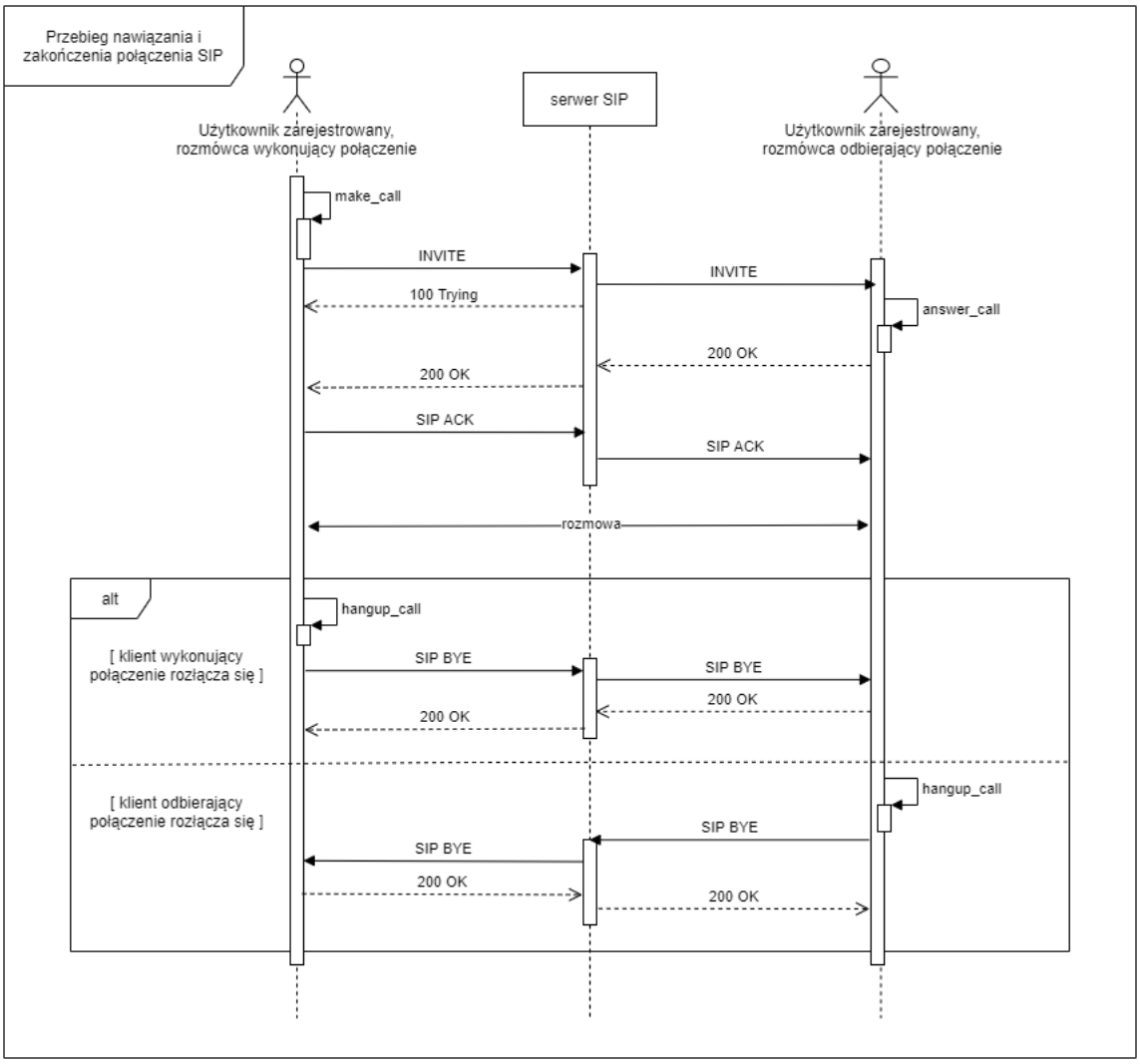
7.2 Przebiegów



Rysunek 5: Diagram przebiegu rejestracji.

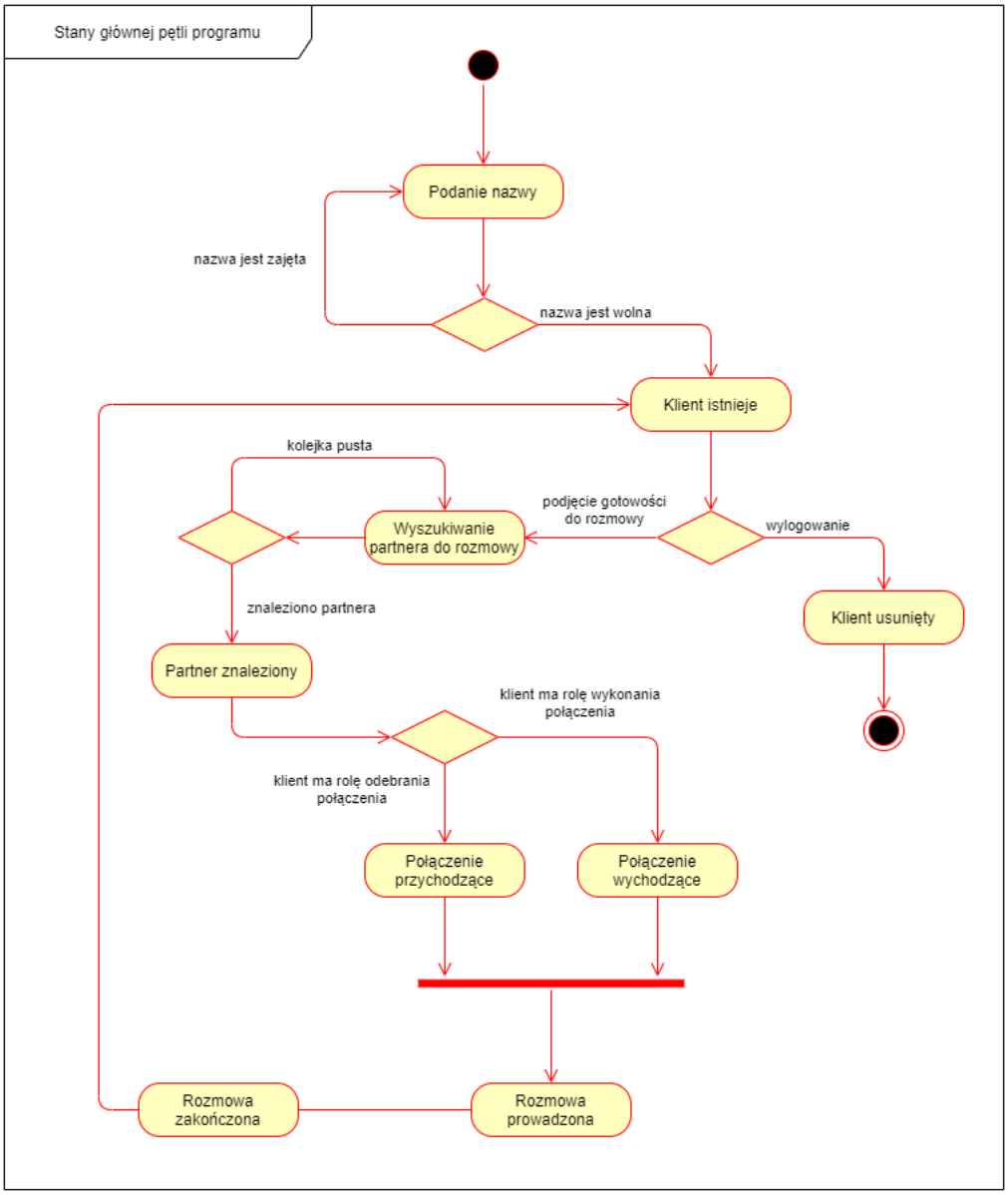


Rysunek 6: Diagram przebiegu szukania rozmówcy.

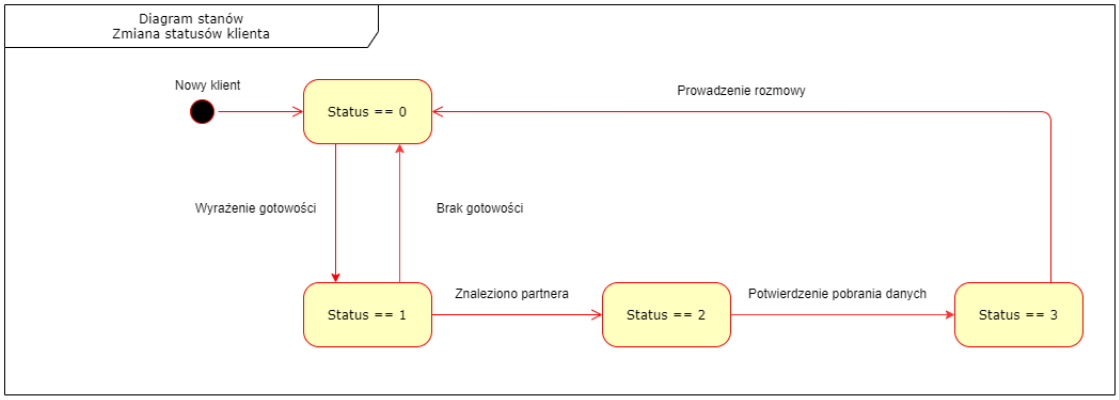


Rysunek 7: Diagram przebiegu nawiązania i zakończenia połączenia SIP.

7.3 Stanów

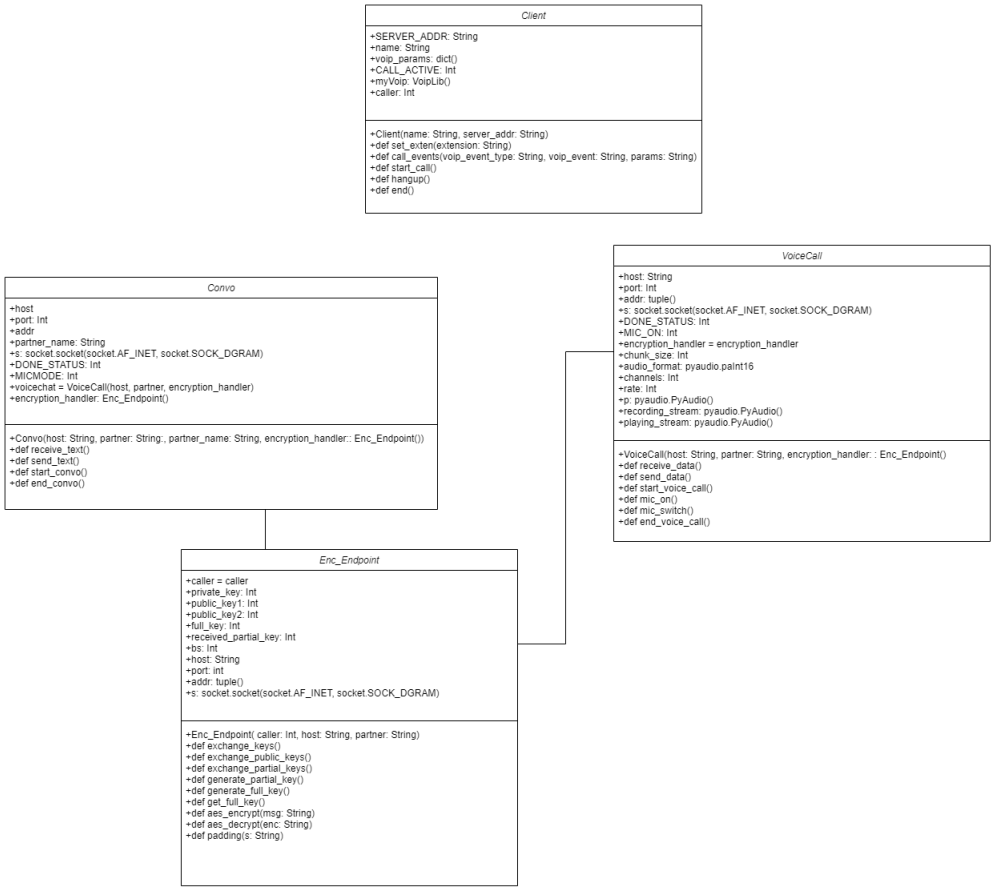


Rysunek 8: Diagram stanów głównej pętli programu.



Rysunek 9: Diagram stanów zmiany statusu klienta.

7.4 Klas

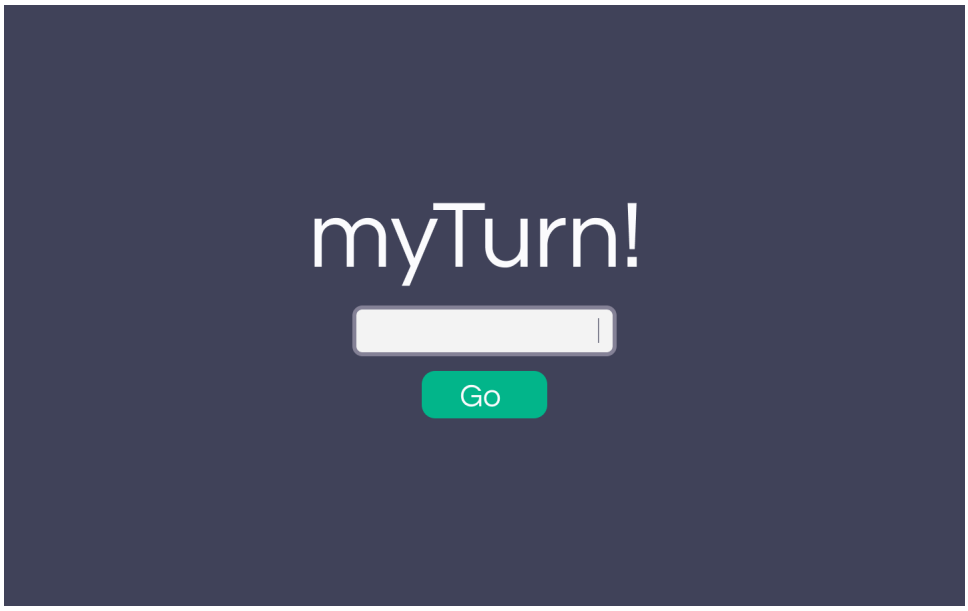


Rysunek 10: Diagram klas.

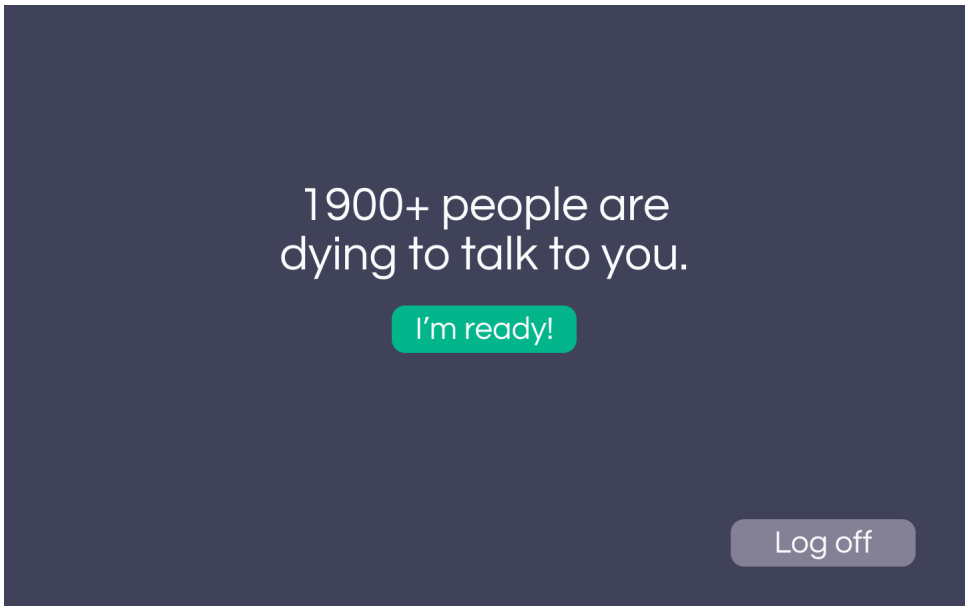
8 Projekt interfejsu graficznego

Na interfejs graficzny składają się cztery strony:

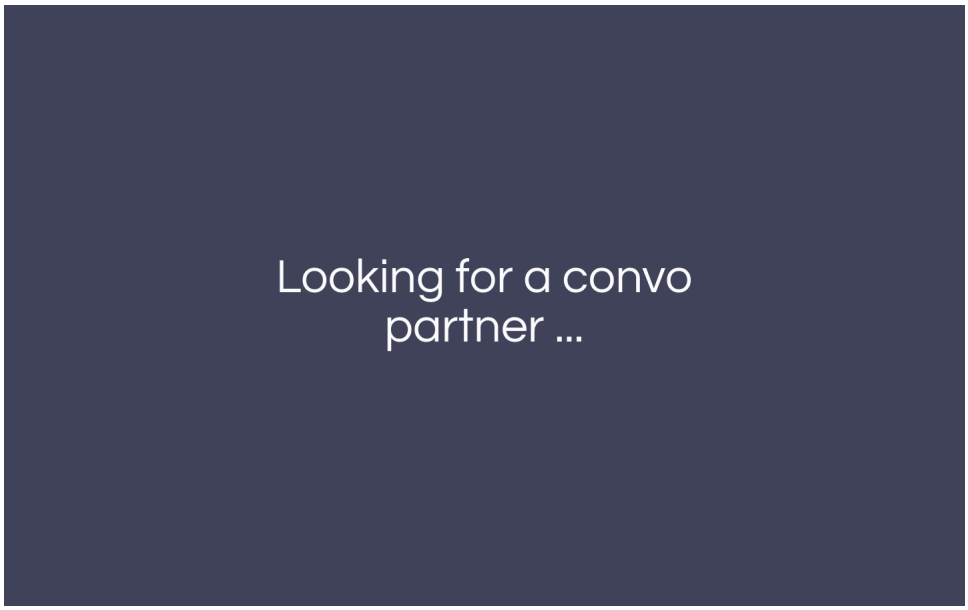
- **login** - strona umożliwiająca użytkownikowi zalogowanie się. Wyświeltana po wejściu w aplikację i po wylogowaniu się.
- **main** - strona umożliwiająca użytkownikowi wyrażenie gotowości do rozmowy. Wyświeltana po zalogowaniu się i po zakończeniu rozmowy.
- **wait** - strona wyświetlana po wyrażeniu przez użytkownika gotowości do rozmowy, sygnalizuje to, że trwa wyszukiwanie partnera do rozmowy.
- **convo** - strona wyświetlana podczas prowadzenia rozmowy. Umożliwia wysyłanie wiadomości tekstowych do partnera, udostępnienia dźwięku z mikrofonu i zakończenie rozmowy.



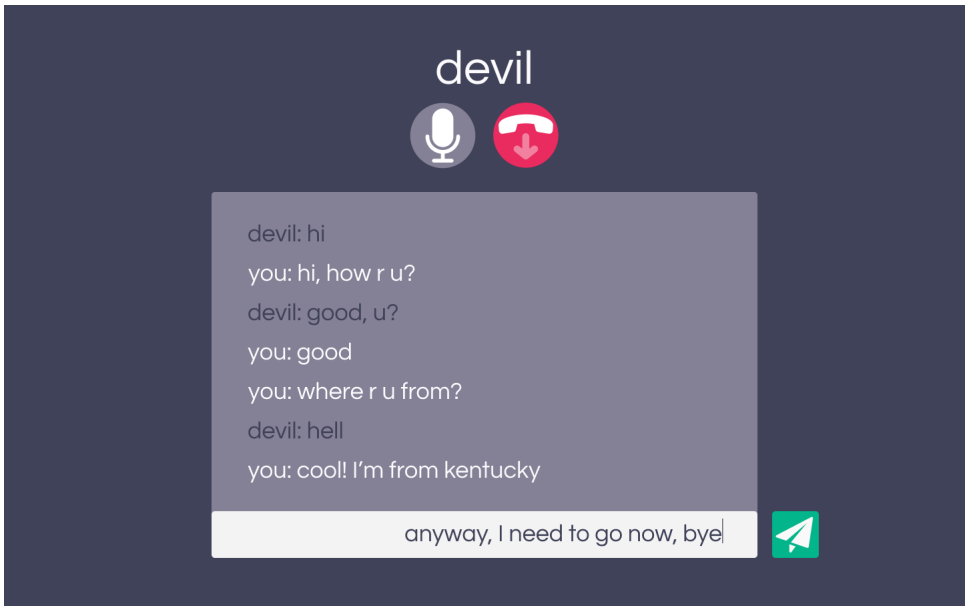
Rysunek 11: Strona login (źródło: rysunek wykonany przez autorkę)



Rysunek 12: Strona main (źródło: rysunek wykonany przez autorkę)



Rysunek 13: Strona wait (źródło: rysunek wykonany przez autorkę)



Rysunek 14: Strona convo (źródło: rysunek wykonany przez autorkę)

9 Najważniejsze metody i fragmenty kodu aplikacji

9.1 Fragment API serwera

Najważniejszą rolę w sterowaniu serwerem odgrywa **API** napisane przy pomocy biblioteki *flask*. Poszczgólne punkty końcowe odpowiadają za następujące funkcjonalności:

- '/add-user-to-sip-exten-conf/<string:user_name>', methods=['POST']

  Sprawdza, czy użytownik o danej nazwie istnieje już na serwerze, jeśli nie to tworzy użytkownikowi o danej nazwie - *user_name* unikalny numer oraz dodaje wszelkie wymagane z nim informacje do plików konfiguracyjnch serwera,

- '/sip-file-lookup', methods=['GET']

  Zapewnia podgląd pliku konfiguracyjnego *sip.conf* serwera **Asterisk**.

- '/user-status', methods=['GET'

  Sprawdza status każdego klienta na serwerze.

- '/pair-status', methods=['GET']

  Sprawdza wszystkie pary wylosowane do rozmowy.

- '/update-status-rdy/<string:user_name>'

  Zmienia wartość statusu użytkownika - *user_name* do wartości *1*, czyli gotowy do przeprowadzenia rozmowy.

- '/update-status-busy/<string:user_name>'

  Zmienia wartość statusu użytkownika - *user_name* do wartości *0*, czyli zajęty.

- '/update-status-rdy-to-talk/<string:user_name>'

  Zmienia wartość statusu użytkownika - *user_name* do wartości *3*, czyli potwierdzenie gotowości do rozmowy.

- '/get-peer/<string:user_name>', methods=['GET']

  Paruje użytkownika - *user_name* z innym, który wyraził chęć do rozmowy. Zwraca kod błędu lub wszystkie wymagane informacje do przeprowadzenia rozmowy - numer telefonu, adres IP, nazwa.

- '/delete-user/<string:user_name>', methods=['DELETE']

  Usuwa wszelkie informacje na temat użytkownika - *user_name* z plików serwera.

```
@app.route('/add-user-to-sip-exten-conf/<string:user_name>',
↪  methods=['POST'])
def add_user_to_sip_conf(user_name):
    if api_logic.if_user_exists(user_name):
        return jsonify({"User exists": "Exten NOT Created"}), 404
    else:
        exten_string_to_append =
        ↪  api_logic.add_user_to_exten_conf(user_name)
        sip_string_to_append =
        ↪  api_logic.add_user_to_sip_conf(user_name)
        return jsonify({"User added": "Exten Created"}), 200


@app.route('/sip-file-lookup', methods=['GET'])
def sip_file_lookup():
    data = api_logic.read_conf_file()
    return data, 200


@app.route('/user-status', methods=['GET'])
def status():
    return jsonify(USERS), 200


@app.route('/pair-status', methods=['GET'])
```

```
def status2():
    return jsonify(PAIRS), 200


@app.route('/update-status-rdy/<string:user_name>',
↪  methods=['PUT'])
def update_user_status_rdy(user_name):
    api_logic.set_user_flag(str(user_name), 1)
    return jsonify({"Status changed": "READY"}), 200


@app.route('/update-status-rdy-to-talk/<string:user_name>',
↪  methods=['PUT'])
def update_user_status_rdy_to_talk(user_name):
    api_logic.set_user_flag(str(user_name), 3)
    return jsonify({"Status changed": "READY TO TALK"}), 200


@app.route('/update-status-busy/<string:user_name>',
↪  methods=['PUT'])
def update_user_status_busy(user_name):
    api_logic.set_user_flag(str(user_name), 0)
    return jsonify({"Status changed": "BUSY"}), 200


@app.route('/user-status/<string:user_name>', methods=['GET'])
def get_user_status(user_name):
    status = api_logic.get_user_status(user_name)
    return jsonify(status), 200


@app.route('/user-ip/<string:user_name>', methods=['GET'])
def get_peer_ip(user_name):
    ip = api_logic.get_peer_ip_address(user_name)
    return jsonify(ip), 200


@app.route('/get-peer/<string:user_name>', methods=['GET'])
def get_peer_exten(user_name):
    user_flag = api_logic.get_user_status(str(user_name))
    users_pool = api_logic.get_all_user_except_with_status(str(u⌋
    ↪  ser_name),
    ↪  1)
    if user_flag == 0:
        return jsonify({"You are not rdy yet": ""}), 226
```

```
    if user_flag == 1:
        if len(users_pool) == 0:
            return jsonify({"Queue is empty": ""}), 208
    if user_flag == 2:
        if str(user_name) in PAIRS.keys():
            number =
            ↪  api_logic.get_dial_number(PAIRS[str(user_name)])
            ip_addr = api_logic.get_peer_ip_address(PAIRS[str(us
            ↪  er_name)])
            return jsonify({"call": PAIRS[str(user_name)],
                            "exten": str(number),
                            "ip_to_send_data": str(ip_addr)}),
                            ↪  202
        if str(user_name) in PAIRS.values():
            swapped_pairs = {val: key for key, val in
            ↪  PAIRS.items()}
            name_ip = swapped_pairs[str(user_name)]
            ip_addr = api_logic.get_peer_ip_address(str(name_ip))
            # u_check = swapped_phone_book[str(user_name)]
            # number = api_logic.get_dial_number(u_check)
            return jsonify({"call": name_ip,
                            "exten": None,
                            "ip_to_send_data": str(ip_addr)}),
                            ↪  230
    else:
        # choose random one
        peer = random.choice(users_pool)
        # set both status to 2
        api_logic.set_user_flag(str(user_name), 2)
        api_logic.set_user_flag(str(peer), 2)
        # add to pairs
        PAIRS[str(user_name)] = str(peer)
        number = api_logic.get_dial_number(PAIRS[str(user_name)])
        ip_addr =
        ↪  api_logic.get_peer_ip_address(PAIRS[str(user_name)])
        return jsonify({"call": PAIRS[str(user_name)],
                        "exten": str(number),
                        "ip_to_send_data": str(ip_addr)}), 202


@app.route('/delete-user/<string:user_name>', methods=['DELETE'])
def delete_user(user_name):
    api_logic.delete_user_from_exten_conf(user_name)
    api_logic.delete_user_from_sip_conf(user_name)
    api_logic.delete_from_list_dict(USERS, user_name)
    api_logic.delete_from_list_dict(DIAL_NUMBERS, user_name)
```

```
    if user_name in PAIRS.keys():
        del PAIRS[str(user_name)]

    if user_name in PAIRS.values():
        swapped_pairs = {val: key for key, val in PAIRS.items()}
        peer_name = swapped_pairs[user_name]
        del PAIRS[str(peer_name)]

    return jsonify({"Deleted": user_name,
                    "PAIRS": PAIRS,
                    "DIAL": DIAL_NUMBERS,
                    "USERS": USERS}), 200
```

9.2 Klient SIP

Odpowiednie inicjowanie i zakończenie komunikacji SIP wymagało utworzenia funkcji reagującej na zdarzenia wysyłane przez serwer SIP. Przykładowo, czynność taka jak zadzwonienie pod dane rozszerzenie SIP wymaga otrzymania od serwera informacji o pomyślnym zarejestrowaniu użytkownika, a potem musi zostać potwierdzone komunikatem o dzwonieniu do naszego rozmówcy. Metoda *call_ events()* reaguje na wszystkie istotne komunikaty serwera.

```
def call_events(self, voip_event_type, voip_event, params):
        print "Received event type:%s Event:%s -> Params: %s" %
        ↪  (voip_event_type, voip_event, params)

        # event triggered when the account registration has been
        ↪  confirmed by the SIP server
        if (voip_event == VoipEvent.ACCOUNT_REGISTERED):
            print "Account registered"

        # event triggered when a new call is incoming
        elif (voip_event == VoipEvent.CALL_INCOMING):
            print "INCOMING CALL From %s" % params["from"]
            time.sleep(2)
            print "Answering..."
            self.myVoip.answer_call()

        # event  triggered when a call has been established
        elif (voip_event == VoipEvent.CALL_ACTIVE):
            print "The call with %s has been established" %
            ↪  self.myVoip.get_call().get_remote_uri()
            self.CALL_ACTIVE=1

        # events triggered when the call ends for some reasons
```

```
        elif (voip_event in
        ↪  [VoipEvent.CALL_REMOTE_DISCONNECTION_HANGUP,
        ↪  VoipEvent.CALL_REMOTE_HANGUP,
                            VoipEvent.CALL_HANGUP]):
            print "End of call."
            self.CALL_ACTIVE=0

        # event triggered when the library was destroyed
        elif (voip_event == VoipEvent.LIB_DEINITIALIZED):
            print "Lib Destroyed. Exiting from the app."
            return

        # print information about other events triggered by the
        ↪  library
        else:
            print "Received unhandled event type:%s --> %s" %
            ↪  (voip_event_type, voip_event)
```

9.3 Komunikacja tekstowa

Komunikacja tekstowa pomiędzy klientami rozpoczyna się po nawiązaniu między nimi połączenia SIP. Klient, w oknie czatu tekstowego, ma możliwość wysłania wiadomości lub wybrania jednej z opcji: zakończenia rozmowy przez wpisanie znaku q lub włączenie/wyłączenie mikrofonu przez wpisanie znaku m.

```
def receive_text(self):
        while self.DONE_STATUS == 0:
            try:
                data_encrypted, _ = self.s.recvfrom(1024)
                # print "message before decryption:",
                ↪  data_encrypted
                data_deciphered = self.encryption_handler.aes_de
                ↪  crypt(data_encrypted)
                print(self.partner_name+": " + data_deciphered)
            except:
                pass

def send_text(self):
    while self.DONE_STATUS == 0:
        try:
            text = raw_input()
            if text == 'q':
                sure = raw_input("Are you sure you want to
                ↪  hangup? y/n ")
                if sure == 'y':
                    self.end_convo()
```

```
                break
            elif text == 'm':
                self.voicechat.mic_switch()
            elif len(text) > 0:
                text_encrypted =
                ↪  self.encryption_handler.aes_encrypt(text)
                # print "message after encryption:",
                ↪  text_encrypted
                self.s.sendto(text_encrypted, self.addr)
                continue
        except:
            pass
```

9.4 Komunikacja głosowa

Klienci mają możliwość komunikacji głosowej - dźwięk pobrany z mikrofonu jest wysyłany do rozmówcy w pakietach UDP. Decyzja o udostępnieniu dźwięku z mikrofonu jest jedną z opcji w czacie tekstowym opisanym w poprzedniej sekcji. Poniżej umieszczono kod metod odbierania i wysyłania pakietów w komunikacji głosowej.

```
def receive_data(self):
        self.playing_stream =
        ↪  self.p.open(format=self.audio_format,
        ↪  channels=self.channels, rate=self.rate, output=True,
        ↪  frames_per_buffer=self.chunk_size)

        while self.DONE_STATUS == 0:
            try:
                encrypted_data = self.s.recvfrom(16)
                decrypted_data = self.encryption_handler.aes_dec
                ↪  rypt(encrypted_data)
                self.playing_stream.write(decrypted_data)
            except:
                pass

def send_data(self):
    self.recording_stream =
    ↪  self.p.open(format=self.audio_format, channels=1,
    ↪  rate=self.rate, input=True,
    ↪  frames_per_buffer=self.chunk_size)
    while self.MIC_ON == 1:
        try:
            data = self.recording_stream.read(16)
            encrypted_data =
            ↪  self.encryption_handler.aes_encrypt(data)
```

```
            self.s.sendto(encrypted_data, self.addr)
        except:
            pass
```

9.5 Synchronizacja typów komunikacji

Poprawne przeprowadzenie komunikacji pomiędzy klientami wymaga ustalenia połączenia SIP i kontrolowania, czy nie zostało one przerwane - oznaczałoby to zakończenie również komunikacji tekstowej i głosowej pomiędzy klientami. Wykonane czynności zależą również od tego, czy klientowi zostaje wylosowana opcja wykonania połączenia lub odebrania go. Przedstawiona poniżej główna pętla programu pozwala użytkownikowi na zalogowanie się przez podanie nazwy oraz na wyrażenie chęci do przeprowadzenia rozmowy, a następnie kontroluje zachowania klienta i jego rozmówcy, a w przypadku zerwania połączenia SIP zamyka pozostałe kanały komunikacji pomiędzy nimi.

```
# get login from user
while True:
    name = raw_input("Hi! What's your name? ")
    if len(name) > 0:
        final_endpoint = URL + ENDPOINT_NEW_USER + str(name)
        req = requests.post(final_endpoint)
        if req.status_code != 200:
            print "Sorry, that name is already taken."
        else:
            break

client = Client(name, server_ip)

while True:
    want = raw_input("Press Enter when you're ready to talk, " +
    ↪  name + '!' + '\n Type "quit" if you want to quit.')
    if want == 'quit':
        print "Bye!"
        break

    # let server know you're ready to talk.
    final_endpoint = URL + ENDPOINT_USER_STATUS_RDY + str(name)
    req = requests.put(final_endpoint)
    if req.status_code != 200:
        print "Sorry, couldn't let the server know you're ready."
        break

    # find a partner to talk to
    final_endpoint = URL + ENDPOINT_PEER + str(name)
    while True:
```

```
req = requests.get(final_endpoint, verify=False)

if req.status_code == 226:
    time.sleep(1)
    continue

if req.status_code == 208:
    time.sleep(1)
    continue

# you'll be making SIP call
if req.status_code == 202:
    caller = 1
    info_dict = json.loads(req.text)
    partner_name = info_dict["call"]
    exten = info_dict["exten"]
    partner_ip = info_dict["ip_to_send_data"]
    enc_endpoint = Enc_Endpoint(1, my_ip, partner_ip)

    # see if your partner is waiting for your call
    while True:
        status = get_user_status_value(partner_name)
        if int(status) == 3:
            break
        time.sleep(1)

    # set Diffie-Hellman key for encrypted communication
    enc_endpoint.exchange_keys()
    break

# you'll be waiting for SIP call
if req.status_code == 230:
    info_dict = json.loads(req.text)
    partner_ip = info_dict["ip_to_send_data"]
    partner_name = info_dict["call"]

    # set Diffie-Hellman key for encrypted communication
    enc_endpoint = Enc_Endpoint(0, my_ip, partner_ip)
    key_exchange_thread = threading.Thread(target=enc_en⌋
    ↪  dpoint.exchange_keys)
    key_exchange_thread.start()

    # let server know you're waiting for the call
    final_endpoint = URL +
    ↪  ENDPOINT_USER_STATUS_RDY_TO_TALK + str(name)
    req = requests.put(final_endpoint)
```

```
            if req.status_code != 200:
                print "Sorry, couldn't communicate with server."
                break

            break

        time.sleep(1)

    client.set_exten(str(exten))
    client.set_caller(caller)

    # ************** CALL ********************
    call = threading.Thread(target=client.start_call())
    call.start()

    while client.CALL_ACTIVE == 0:
        time.sleep(1)
        print "Waiting for partner..."

    # start text chat
    chat = Convo(my_ip, partner_ip, partner_name, enc_endpoint)
    chat.start_convo()

    # wait for call to end
    while True:
        if client.CALL_ACTIVE == 0:  # someone hang up on us
            chat.end_convo()
            break
        elif chat.DONE_STATUS == 1:  # we want to hang up
            client.hangup()
            break

    final_endpoint = URL + ENDPOINT_USER_STATUS_BUSY + str(name)
    requests.put(final_endpoint)
    # main loop repeats

# kill the client
final_endpoint = URL + "/delete-user/" + str(name)
requests.delete(final_endpoint)
client.end()
```

9.6 Szyfrowanie danych

Zarówno dane przesyłane w ramach komunikacji tekstowej i głosowej są szyfrowane za pomocą klucza uzyskanego przez wymianę kluczy z protokółem Diffiego-Hellmana. Wymiana odbywa się pomiędzy klientami wylosowanymi

do rozmowy poprzez komunikację z protokołem UDP. Szyfrowanie odbywa się za pomocą szyfru AEC w trybie CBC z losowym wektorem wejściowym. Na potrzeby kontroli działania podsystemu, klucze prywatne, częściowe i publiczne to stosunkowo liczby całkowite.

```
class Enc_Endpoint:
    def __init__(self, caller, host, partner):
        self.caller = caller
        self.private_key = random.randint(50, 200)

        if caller == 1:
            self.public_key1 = random.randint(50, 200)
            self.public_key2 = None
            # print "{%s,%s}" % (self.public_key1,
            ↪  self.private_key)
        else:
            self.public_key1 = None
            self.public_key2 = random.randint(50, 200)
            # print "{%s,%s}" % (self.public_key2,
            ↪  self.private_key)

        self.full_key = None
        self.received_partial_key = None
        self.bs = AES.block_size

        self.host = host  # my IP
        self.port = 4015
        self.addr = (partner, 4015)  # partner IP
        self.s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.s.setsockopt(socket.SOL_SOCKET,
        ↪  socket.SO_REUSEADDR, 1)
        self.s.settimeout(1.0)
        self.s.bind((self.host, self.port))

    def exchange_keys(self):
        self.exchange_public_keys()
        self.exchange_partial_keys()

    def exchange_public_keys(self):
        if self.caller == 1:
            print "Sending public key 1:", self.public_key1
            self.s.sendto(str(self.public_key1).encode('utf-8'),
            ↪  self.addr)
            while True:
                try:
                    data, _ = self.s.recvfrom(1024)
```

```
                data = data.decode('utf-8')
                print 'Received public key 2:', data
                self.public_key2 = int(data)
                break
            except:
                pass
    else:
        while True:
            try:
                data, _ = self.s.recvfrom(1024)
                data = data.decode('utf-8')
                print 'Received public key 1:', data
                self.public_key1 = int(data)
                break
            except:
                pass
        print "Sending public key 2:", self.public_key2
        self.s.sendto(str(self.public_key2).encode('utf-8'),
        ↪  self.addr)

def exchange_partial_keys(self):
    partial_key = self.generate_partial_key()
    print "Sending partial key:", partial_key
    self.s.sendto(str(partial_key).encode('utf-8'),
    ↪  self.addr)
    while True:
        try:
            data, _ = self.s.recvfrom(1024)
            data = data.decode('utf-8')
            print 'Received partial key:', data
            self.received_partial_key = int(data)
            break
        except:
            pass
    self.s.close()
    self.generate_full_key()

def generate_partial_key(self):
    partial_key = self.public_key1**self.private_key
    partial_key = partial_key % self.public_key2
    return partial_key

def generate_full_key(self):
    full_key = self.received_partial_key**self.private_key
    full_key = full_key % self.public_key2
```

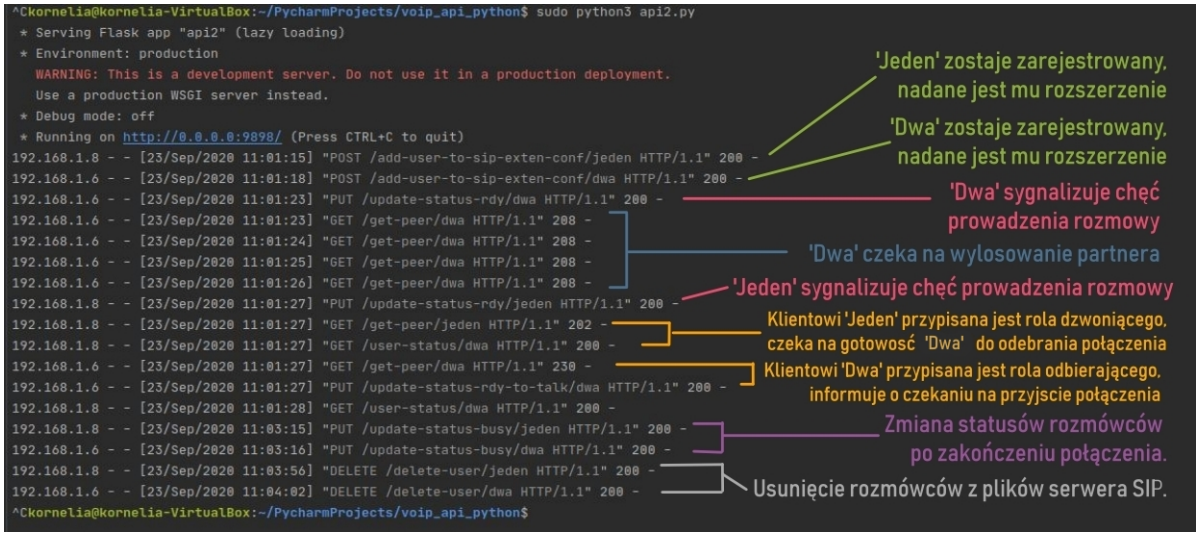
```
        full_key =
        ↪  hashlib.sha256(str(full_key).encode()).digest()
        self.full_key = full_key
        print "Generated full key."

    def get_full_key(self):
        return self.full_key

    def aes_encrypt(self, msg):
        raw = self.padding(msg)
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(self.full_key, AES.MODE_CBC, iv)
        return base64.b64encode(iv +
        ↪  cipher.encrypt(raw.encode()))

    def aes_decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.full_key, AES.MODE_CBC, iv)
        return self.remove_padding(cipher.decrypt(enc[AES.block_
        ↪  size:])).decode('utf-8')

    def padding(self, s):
        return s + (self.bs - len(s) % self.bs) * chr(self.bs -
        ↪  len(s) % self.bs)

    @staticmethod
    def remove_padding(s):
        return s[:-ord(s[len(s) - 1:])]
```

10 Testy i przebieg sesji

W celu przetestowania systemu i zobrazowania jego działania na rzecz dokumentacji wykonano testy dotyczące sześciu głównych aspektów składających się na działanie systemu. Są nimi kolejne etapy głównej pętli programu, czyli wymiana komunikatów z API podczas logowania klienta i losowania klientów, którzy mają przeprowadzić rozmowę, wprowadzenie zmian w plikach serwera po pomyślnym zalogowaniu klienta, inicjacja i zakończenie połączenia SIP z poziomu oprogramowania klienta, przeprowadzenie komunikacji tekstowej i głosowej oraz szyfrowanie przesyłanych danych.

10.1 Komunikaty API

Po podaniu przez klienta nazwy, którą chce się posługiwać, wysyłane jest do API zapytanie o dostępności tej nazwy. API odpowiada przez zezwolenie na używanie nazwy lub poproszenie o inną nazwę. Po uzyskaniu zezwolenia następuje rejestracja klienta - API wprowadza zmiany w plikach serwera, który nadaje klientowi rozszerzenie SIP, dzięki któremu możliwe będzie dzwonienie do klienta. Po wyrażeniu przez klienta chęci do prowadzenia rozmowy API zmienia status klienta i wybiera z listy chętnych do rozmowy klientów rozmówcę. Jednemu z dwóch rozmówców przypisana jest rola dzwoniącego, a drugiemu odbierającego połączenie. Informacje o adresie IP, nazwie i rozszerzeniu klienta są przesyłane do rozmówców i następuje nawiązanie połączenia SIP. Po zakończeniu rozmowy następuje zmiana statusów klientów, a po ich wylogowaniu dane są usuwane z plików serwera SIP.



Rysunek 15: Zrzut ekranu działania API w przykładowym scenariuszu.

10.2 Wprowadzanie zmian w plikach serwera

API ma możliwość wprowadzania zmian w plikach serwera SIP w celu zarejestrowania klientów do kontekstu serwera i umożliwienie im wykonywanie połączeń SIP. Poniżej przedstawiono zmiany wprowadzone w plikach po zarejestrowaniu dwóch klientów.

```
[ulaw-phone](!)                 ; and another one for ulaw-only
        disallow=all
        allow=ulaw
[jeden]
        type=friend
        context=phones
        allow=ulaw, alaw
        secret=12345678abcd
        host=dynamic
[dwa]
        type=friend
        context=phones
        allow=ulaw, alaw
        secret=12345678abcd
        host=dynamic
kornelia@kornelia-VirtualBox:/etc/asterisk$
```

Rysunek 16: Informacje o klientach dodane do pliku konfiguracyjnego serwera SIP po zarejestrowaniu klientów.

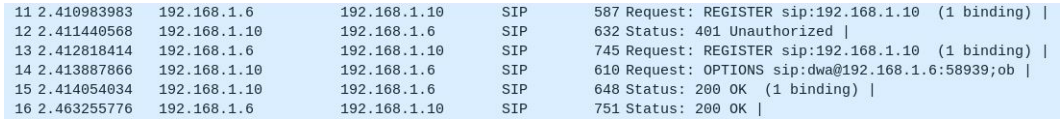
```
kornelia@kornelia-VirtualBox:/etc/asterisk$ sudo cat extensions.conf
[phones]




kornelia@kornelia-VirtualBox:/etc/asterisk$ sudo cat extensions.conf
[sudo] password for kornelia:
[phones]

exten => 111000, 1, NoOp(calling jeden)
exten => 111000, 2, Dial(sip/jeden)
exten => 111000. 3. HangUp

exten => 111001, 1, NoOp(calling dwa)
exten => 111001, 2, Dial(sip/dwa)
exten => 111001. 3. HangUp
kornelia@kornelia-VirtualBox:/etc/asterisk$
```

Rysunek 17: Plik serwera z rozszerzeniami SIP przed i po zarejestrowaniu klientów.

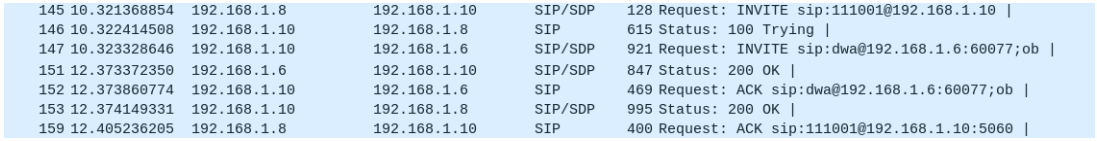
10.3 Połączenie SIP

Po zarejestrowaniu klientów w serwerze SIP i uzyskaniu przez rozmówców swoich danych, możliwe jest wykonanie przez nich połączenia SIP z poziomu oprogramowania klienta. Poniżej przedstawiono zrzuty ekranu działania serwera SIP po nawiązaniu i zerwaniu połączenia oraz odpowiednie logi Wireshark.

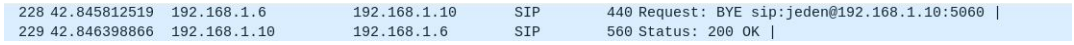
Rysunek 18: Zrzut ekranu działania serwera SIP.



Rysunek 19: Rejestracja klienta SIP.



Rysunek 20: Nawiązanie połączenia SIP.



Rysunek 21: Zakończenie połączenia SIP.

10.4 Komunikacja tekstowa

Po nawiązaniu połączenia SIP pomiędzy klientami rozpoczyna się komunikacja tekstowa. Poniżej przedstawiono wygląd przykładowej komunikacji tekstowej pomiędzy klientami. Podgląd pakietów przesyłanych w ramach komunikacji zostanie umieszczony w podsekcji Szyfrowanie przesyłanych danych.

Rysunek 22: Czat tekstowy pomiędzy klientami.

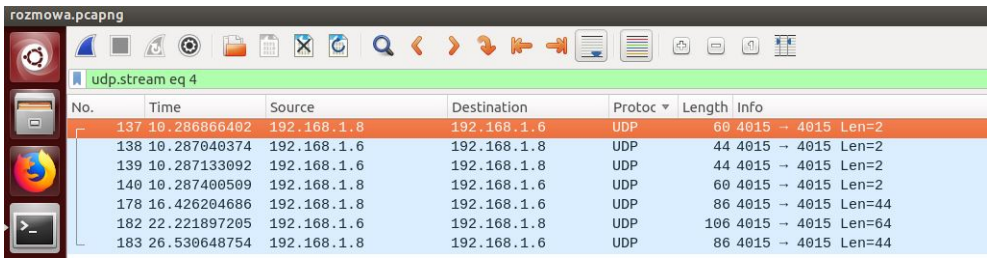
10.5 Komunikacja głosowa

Wraz z rozpoczęciem komunikacji tekstowej klienci mają możliwość udostępnienia dźwięku ze swojego mikrofonu w celu prowadzenia rozmowy głosowej z wylosowanym klientem. Aby rozpocząć rozmowę przez mikrofon, klient wpisuje literę m w czacie głosowym.

Rysunek 23: Włączenie przez klienta mikrofonu.

10.6 Szyfrowanie przesyłanych danych

Poniżej przedstawiono podgląd zawartości pakietów przesyłanych pomiędzy rozmówcami - pakietów przesyłanych w ramach wymiany kluczy i rozmowy tekstowej z Rysunku 18. W celu weryfikacji poprawności przesyłanych kluczy, w programie wypisano wysyłane i odbierane klucze publiczne i częściowe.

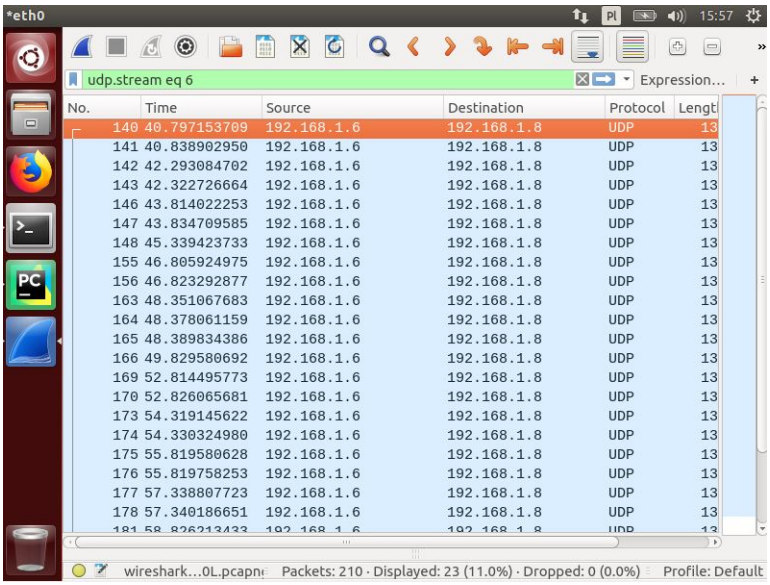


Rysunek 24: Seria pakietów UDP przesłanych podczas wymiany kluczy i rozmowy tekstowej.

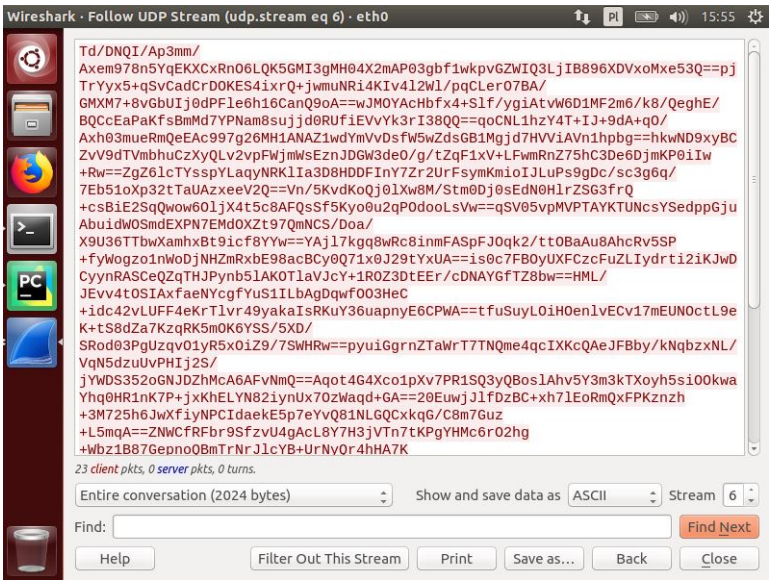
Rysunek 25: Podgląd zawartości pakietów z poprzedniego rysunku.

Rysunek 26: Podgląd zawartości pakietów z poprzedniego rysunku.

Szyfrowane są wszystkie pakiety, zarówno te dotyczące komunikacji tekstowej i głosowej. Poniżej przedstawiono podgląd zawartości pakietów przesyłanych po włączeniu przez klienta mikrofonu.



Rysunek 27: Logi Wireshark po włączeniu mikrofonu.



Rysunek 28: Podgląd zawartości pakietów przesyłanych po włączeniu przez klienta mikrofonu.

11 Analiza bezpieczeństwa

Metody i środki ochrony danych w systemie dzielą się na kategorie:

- **Poufność**

  Pakiety wysyłane pomiędzy klientami, związane z komunikacją tekstową i głosową są szyfrowane za pomocą klucza ustalonego przez protokół Diffiego-Hellmana. Przed nawiązaniem połączenia następuje wymiana między klientami kluczy publicznych i częściowych przez pakiety UDP. Po wymianie kluczy każdy z klientów generuje na ich podstawie identyczny klucz pełny, który służy do szyfrowania przesyłanych między klientami danych. Dzięki temu, że w generowaniu klucza pełnego używana jest operacja modulo związana z kluczem prywatnym unikanlnym dla każdej strony komunikacji, klucze publiczne i częściowe są przesyłane jawnie, ponieważ nie są one wystarczające do odszyfrowania przesyłanych później danych - komunikacja tekstowa i głosowa są zatem zabezpieczone przed atakiem takim jak Man-in-the-Middle.

  Szyfrowanie odbywa się za pomocą algorytmu AES. Klucz pełny generowany na podstawie wymienionych kluczy zostaje przekształcony na 32-bajtowy klucz, odpowiedni do szyfru AES. Blok szyfru wynosi 16 bajtów. Każda wiadomość tekstowa i pakiet z danymi dźwiękowymi są szyfrowane przed wysłaniem przez nadawcę i deszyfrowane po otrzymaniu przez odbiorcę. Szyfrowanie odbywa się metodą **AES CBC (tryb wiązania**

bloków zaszyfrowanych) z wektorem wyjściowym długości 16-bajtów generowanym losowo przed szyfrowaniem wiadomości. Przed zaszyfrowaniem wiadomości dodawany jest padding tak, by dane do zaszyfrowania miały rozmiar wielokrotności bloku szyfru.

- **Integralność**

  Pakiety wysyłane pomiędzy klientami są pakietami UDP. Protokół ten zapewnia weryfikację integralności przez sumę kontrolną.

- **Dostępność**

  Wszelkie dane dotyczące zarejestrowanych klientów są dostępne jedynie w plikach serwera SIP i w zasobach API przechowującego statusy klientów. Dostęp do adresu IP i rozszerzenia danego klienta jest możliwy jedynie przez otrzymanie takich danych od API w przypadku zostania wylosowanym do rozmowy z tym klientem - nie jest możliwe wysłanie zapytania do API, które zwróci klientowi dane wszystkich zarejestrowanych klientów, ponieważ jedyne, czego klient potrzebuje, to adres i rozszerzenie wylosowanego właśnie rozmówcy. Dodatkowym zabezpieczeniem jest fakt, że rozszerzenie klienta jest inne przy każdym jego zalogowaniu, dane są więc stale zmieniane. Dane klienta są usuwane z plików serwera SIP po jego wylogowaniu, dlatego nie są one szczególnie narażone na przechwycenie przez osoby nieupoważnione.

12 Podział prac

Autor	Podzadanie
Kornelia Maik	moduł klienta SIP, moduł łączący trzy typy komunikacji (SIP, tekstowa, głosowa) w wielowątkową całość, komunikacja tekstowa, komunikacja głosowa, szyfrowanie pakietów, prowadzenie komunikacji SIP z obsługą zdarzeń
Patryk Wenz	konfiguracja serwera Asterisk (instalacja, konfiguracja plików, dostosowanie parametrów), implementacja API umożliwiającego wykonywanie zdarzeń na serwerze z poziomou klienta, implementacja funkcji do edytowania ustawień serwera

13 Podsumowanie

13.1 Cele zrealizowane, cele niezrealizowane

Cele zrealizowane:

- poznanie działania serwera SIP w celu jego konfiguracji
- komunikacja SIP pomiędzy klientem i serwerem Asterisk

- nawiązywanie i zrywanie połączenia SIP pomiędzy klientami
- stworzenie API, które obsługuje żądania klientów i wprowadza konieczne do komunikacji SIP zmiany w plikach serwera Asterisk
- stworzenie podsystemu losowania klientów do prowadzonych rozmów
- komunikacja tekstowa UDP pomiędzy klientami
- komunikacja głosowa UDP pomiędzy klientami
- szyfrowanie pakietów tekstowych i głosowych z pomocą protokołu AES CBC i wymiany kluczy Diffiego-Hellmana
- połączenie wszystkich powyższych elementów w działający system obsługi połączeń SIP i komunikacji między klientami.

Cele niezrealizowane:

- wykonanie zaplanowanego interfejsu graficznego.

13.2 Napotkane problemy

Pierwszym problemem, jaki zespół napotkał na etapie planowania kolejnych etapów dążących do wykonania systemu, było znalezienie biblioteki, która pozwoliłaby na nawiązanie komunikacji z serwerem Asterisk. Zespół chciał jak najbardziej zrozumieć poszczególne kroki komunikacji SIP i szukał biblioteki, która jasno opisuje zaistniałe w komunikacji zdarzenia i pozwala na odpowiednie reagowanie na odpowiedzi serwera. Tą biblioteką okazała się oparta na bibliotece pjSIP biblioteka most-voip.

Kolejnym problemem była odpowiednia konfiguracja serwera Asterisk tak, by pozwolił on na komunikację klientów. Problem rozwiązało przeczytanie wielu dokumentacji i artykułów. Dzięki temu zespół mógł przetestować komunikację klientów SIP i przejść do tworzenia API wprowadzającego zmiany w plikach Asterisk.

Po pomyślnej implementacji nawiązywania połączeń SIP zespół musiał zaimplementować wielowątkową obsługę komunikacji tekstowej i głosowej pomiędzy klientami, co z natury wielowątkowości stwarzało problemy rozwiązywalne po dokładnej analizie kodu i działania wątków w języku python.

Niespodziewanym problemem okazały sie statusy HTTP. Przez początkową nieuwagę jeden z punktów końcowych **API** zwracał dla pewnego przypadku status *100* oznaczający nieustanne powtarzanie zapytań, co skutkowało zerwaniem połączenia między klientem a **API**. Po zidentyfikowaniu problemu zmieniono kody statusu na odpowiednie do danej sytuacji.

13.3 Perspektywa rozwoju

Najbardziej istotnym aspektem systemu, który wymaga rozwoju jest interfejs graficzny, który w bardziej przystępny i estetyczny sposób pomógłby użytkownikom orientować się w aplikacji i w przyjemny sposób prowadzić rozmowy.

Kolejnym pomysłem na rozwój jest wprowadzenie rozmów wideo - dzięki współgrającym między sobą podsystemów istniejących już typów komunikacji, wprowadzenie kolejnego środku przekazu do systemu byłoby łatwym do rozwiązania problemem, który wzamian dodałby do systemu przydatną i ekscytującą dla użytkowników funkcjonalność.