Università degli Studi di Napoli "Parthenope"

Reti di Calcolatori

Progetto "Connect"

Docente
Emanuel Di Nardo

Candidato
Luca D'Oriano 0124002499

Anno Accademico 2023-2024

Indice

1 Descrizione **3**

2 Descrizione e schema dell'architettura **3**

2.1 Panoramica dell'architettura del progetto 3

2.2 Architettura WebRTC 5

3 Dettagli implementativi dei client/server **7**

3.1 Implementazione protocollo WebSocket 7

3.2 Logica server WebSocket 10

3.3 Funzionamento 13

4 Manuale **15**

1 Descrizione

Il progetto, ideato per il corso di Tecnologie Web, si concentra sulla creazione di un portale web per l'apprendimento da remoto attraverso lezioni e condivisione di appunti. L'idea di "Connect" nasce appunto per mettere in contatto gli studenti tra loro con i tutors. Grazie a questo portale sarà infatti molto più semplice trovare la persona adatta alle proprie esigenze, per poter colmare le proprie lacune e affrontare al meglio lo studio di una materia o la preparazione di un esame. È a disposizione dello studente la possibilità di contattare, infatti, il tutor in maniera privata e organizzarsi quindi per la lezione direttamente sul portale, tramite videochiamata in P2P (peer-to-peer).

La parte relativa al progetto di Reti di Calcolatori si concentra principalmente sull'implementazione del signaling per l'implementazione delle chiamate peer-to-peer (P2P) utilizzando il protocollo WebSocket. Questo protocollo offre una comunicazione bidirezionale e in tempo reale, ideale per la gestione del signaling in WebRTC.

Nello specifico il progetto prevede l'implementazione del protocollo WebSocket in Python e la successiva creazione di un server WebSocket che funga da punto di comunicazione per gli utenti connessi, consentendo loro di negoziare e stabilire le connessioni in modo efficiente e sicuro.

2 Descrizione e schema dell'architettura

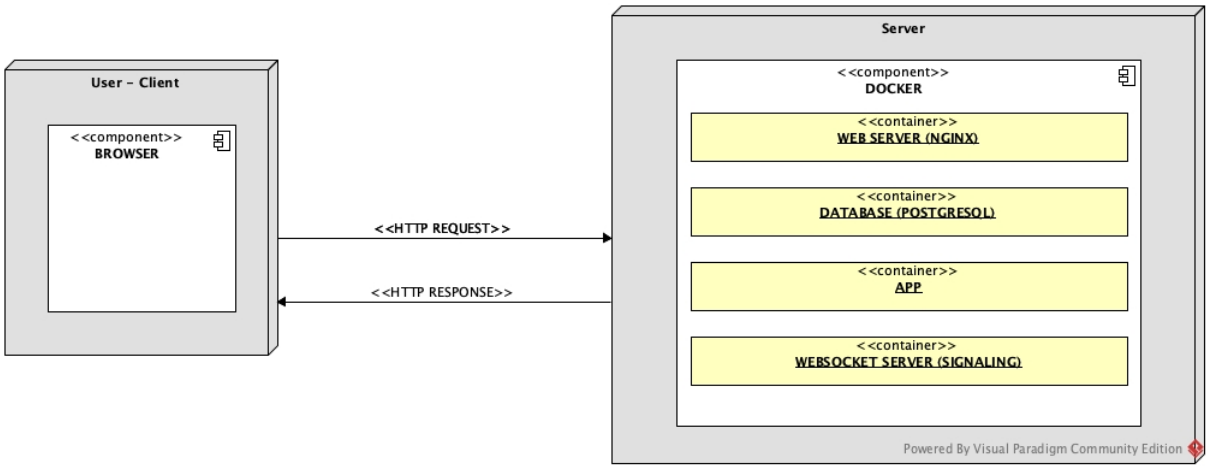
2.1 Panoramica dell'architettura del progetto

"Connect" utilizza un'architettura di tipo Client/Server. Il progetto è stato implementato utilizzando Docker per la containerizzazione, e per semplificare l'orchestrazione e la gestione dei container, è stato adottato Docker Compose. In particolare, è stato creato un file Docker Compose che definisce e configura tutti i servizi necessari per il sistema.

- **Web Server (Nginx):** Nginx è utilizzato per gestire le richieste HTTP/HTTPS in entrata e per dirigere il traffico verso l'applicazione.
- **Database (PostgreSQL):** Un altro servizio è dedicato a PostgreSQL, il nostro database relazionale. Questo servizio gestisce la persistenza dei dati e fornisce un'interfaccia per l'applicazione per interagire con il database.
- **Web App:** L'applicazione stessa è containerizzata e composta da due parti principali: il backend e il frontend.
- **WebSocket Server (Signaling):** Un servizio dedicato è configurato per gestire il server WebSocket. Questo componente è essenziale per consentire la comunicazione bidirezionale in tempo reale tra il server e il frontend, ad esempio durante le videochiamate per la parte di signaling, nonché parte relativa al progetto in questione di Reti di Calcolatori.

La containerizzazione offre numerosi vantaggi, tra cui la facilità di distribuzione, la coerenza dell'ambiente di sviluppo e produzione, e la gestione semplificata delle dipendenze. Inoltre, ogni componente è isolato in un container, garantendo che le dipendenze e le configurazioni non entrino in conflitto con altri servizi o ambienti.

L'utilizzo di Docker e Docker Compose semplifica notevolmente il processo di sviluppo, distribuzione e manutenzione del sistema, garantendo che tutte le componenti siano configurate in modo uniforme e possano essere facilmente gestite in qualsiasi ambiente compatibile con Docker.



2.2 Architettura WebRTC

WebRTC è un tecnologia open-source ideata per connettere in maniera diretta utenti tramite web browsers. E' basata sull'architettura peer-to-peer e permette la comunicazione audio e video.

Il funzionamento è piuttosto semplice, ed è raffigurato nella **figura 2.0**. I due utenti (peers) A e B, prima di poter comunicare tra di loro, devono ottenere il proprio IP pubblico. WebRTC permette di definire un server **STUN** (Session Traversal Utilities for NAT) per ottenere IP e porta dei peers.

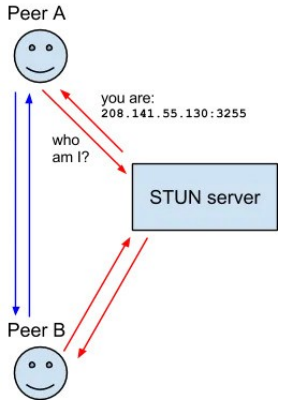


Figura 1.0

Ottenute le informazioni necessarie, i due peers per potersi scambiare l'offerta e la risposta, comunicheranno per mezzo di un server signaling, in questo caso utilizzando le WebSocket.

Lo scopo del server di signaling, è solo quello di stabilire una connessione iniziale tra i due peers. Una volta che la connessione è stata stabilita, non sarà utilizzato per la comunicazione tra i due utenti, che sarà diretta tramite ICE (Interactive Connectivity Establishment), bensì solo per notificare i peer dei vari eventi (entrata di un peer nella stanza, disconnessione, etc)

Per consentire il funzionamento ottimale della maggior parte delle applicazioni basate su WebRTC, è essenziale disporre di un server che agevoli l'instradamento del traffico tra i vari nodi, poiché spesso non è fattibile stabilire una connessione diretta tra i client, soprattutto se non si trovano sulla stessa rete locale. La soluzione più diffusa a questo problema è l'utilizzo di un server **TURN**, che sta per Traversal Using Relays around NAT, un protocollo progettato per instradare il traffico di rete in situazioni complesse come quelle generate dalle reti NAT.

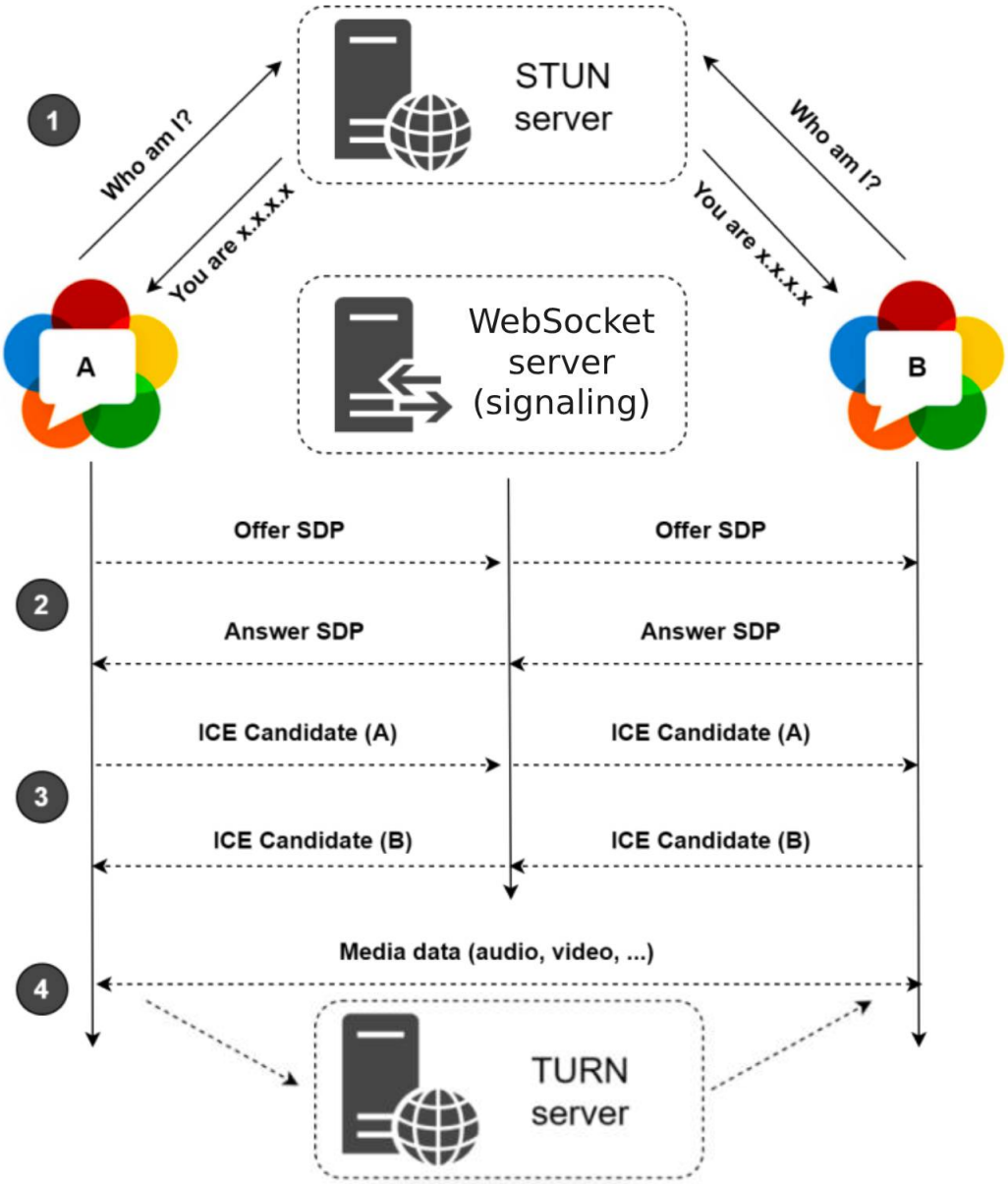

STUN
server
1
Who am I?
You are x.x.x.x
Who am I?
You are x.x.x.x
A
WebSocket
server
(signaling)
B
Offer SDP
Offer SDP
2
Answer SDP
Answer SDP
ICE Candidate (A)
ICE Candidate (A)
3
ICE Candidate (B)
ICE Candidate (B)
Media data (audio, video, ...)
4
TURN
server

Figura 2.0

3 Dettagli implementativi dei client/server

3.1 Implementazione protocollo WebSocket

Parte essenziale di questo progetto, è stata implementare il protocollo WebSocket seguendo le caratteristiche del RFC 6455. Lo sviluppo prevedeva di non utilizzare framework o librerie preesistenti, si è optato per lo sviluppo in Python tramite il modulo built-in *socket.py*, che provvede accesso all'interfaccia low-level per Berkeley sockets.

```
import socket
import struct
import threading
import json

from hashlib import sha1
from base64 import b64encode

from typing import Callable

# Constants
FIN = 0x80
OPCODE_TEXT = 0x1
OPCODE_CLOSE = 0x8


class WebSocket(object):
    def __init__(self, host="localhost", port=9999, certs=None):
        self.host = host
        self.port = port
        self.certs = certs
        self.clients = set()


    def start(self):
        try:
            # Creating and setting up the socket
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                self.socket = sock

                # Allowing reuse of the socket address
                sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

                # Binding the socket to the specified host and port
                sock.bind((self.host, self.port))
                # Listening for incoming connections.
                # 5 is the numberof unaccepted connections that the system will allow
                                                        before refusing new
                                                        connections.
                sock.listen(5)

                print(
                    "WebSocket server started at "
                    f"{'wss' if self.certs else 'ws'}://{self.host}:{self.port}"
                )

                # Accepting and handling client connections
                while True:
                    client, address = sock.accept()
```

```
            print(f'Connection incoming from: {address}')
            threading.Thread(
                target=self._handler, args=(address, client), daemon=True
            ).start()

    except KeyboardInterrupt:
        # Closing all client connections and the server socket upon keyboard
                                                    interrupt
        [client.close() for client in self.clients]
        sock.close()

def _generate_accept_key(self, key: str):
    # Generating the accept key for the WebSocket handshake
    GUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
    hash = sha1((key + GUID).encode("utf-8")).digest()
    return b64encode(hash).decode("utf-8")

def _handshake(self, data: bytes, client: socket.socket):
    # Handling WebSocket handshake
    data = data.decode("utf-8")
    if data.startswith("GET"):
        headers = data.split("\r\n")
        for header in headers:
            if "Sec-WebSocket-Key" in header:
                key = header.replace("Sec-WebSocket-Key:", "").strip()
                accept_key = self._generate_accept_key(key)
                response = (
                    "HTTP/1.1 101 Switching Protocols\r\n"
                    "Upgrade: websocket\r\n"
                    "Connection: Upgrade\r\n"
                    f"Sec-WebSocket-Accept: {accept_key}\r\n\r\n"
                )
                client.send(response.encode("utf-8"))

def _receive(self, client: socket.socket, size=1024) -> bytes:
    # Receiving data from the client
    message = bytearray()
    while True:
        chunk = client.recv(size)
        message += chunk
        if not chunk or len(chunk) < size:
            break
    return message


def _handler(self, address, client: socket.socket):
 # Handling client connection
 while True:
     data = self._receive(client, 2048)
     if data:
         if client in self.clients:
             data, opcode = self._decode_frames(frame=data)

             if opcode == OPCODE_TEXT:
                 # Handling text data
                 self._onmessage(client, message=data.encode("utf8"))

             if opcode == OPCODE_CLOSE:
                 # Handling close frame
                 self.clients.remove(client)
```

```
                    self._onclose(client)
                    return

            else:
                # Performing handshake for new connections
                self._handshake(data, client)
                self.clients.add(client)
                # Notifying about successful handshake
                self.send(client, message=json.dumps({
                  "type": "handshake",
                  "status": "complete"
                }))



def send(self, client: socket.socket, message: str):
    # Sending data to the client
    length = len(message.encode("utf8"))
    header = struct.pack("!B", FIN + OPCODE_TEXT)

    if length <= 125:
        payload_length = struct.pack("!B", length)
    elif 126 <= length <= 65535:
        payload_length = struct.pack("!BH", 126, length)
    else:
        payload_length = struct.pack("!BQ", 127, length)

    response = header + payload_length + message.encode("utf8")
    client.send(response)


def _decode_frames(self, frame: bytearray):
    # Decoding WebSocket frames
    # check if fin is not 0 (if not 0 then is valid)
    fin = bool(frame[0] & 128)
    #check true or false to verify if data is unmasked
    masked = bool(frame[0] & 128)

    opcode = frame[0] & 127 #   (127=1111111)
    payload_length = frame[1] & 127 # unmasks data

    # if payload len is between 0 and 125
    if payload_length <= 125: # (125=1111101)
        mask = frame[2:6]
        payload = frame[6:]

    if payload_length == 126: # (126=1111110)
        mask = frame[4:8]
        payload = frame[8:]

    if payload_length == 127:
        mask = frame[10:4]
        payload = frame[14:]

    message = bytearray()
    if payload is not None:
        try:
            """ From RFC 6455 section 5.3:
            To convert masked data into unmasked data, or vice versa, the following
            algorithm is applied.
```

```
            Octet (byte) i of the transformed data ("transformed-octet-i") is the XOR
                                                                of
            octet i of the original data ("original-octet-i") with octet at index
            i modulo 4 of the masking key ("masking-key-octet-j"):
                                    j = i MOD 4
            transformed-octet-i = original-octet-i XOR masking-key-octet-j
            """
            for byte in range(len(payload)):
               message.append(payload[byte] ^ mask[byte % 4]) # xor to unmask the
                                                               data
            message = message.decode("utf-8")
         except UnicodeDecodeError as e:
            message = None

      return message, opcode

   def onmessage(self, callback: Callable[[socket.socket, str, str], None]):
      # Callback function for receiving messages
      self._onmessage = callback


   def onopen(self, callback: Callable[[socket.socket], None]):
      # Callback function for new client connections
      self._onopen = callback


   def onclose(self, callback: Callable[[socket.socket], None]):
      # Callback function for client disconnections
      self._onclose = callback


   def _onmessage(self, client: socket.socket, message: str):
      # Default message handling function
      pass


   def _onopen(self, client: socket.socket, message: str):
      # Default function for new connections
      pass


   def _onclose(self, client: socket.socket, message: str):
      # Default function for client disconnections
      pass
```

3.2 Logica server WebSocket

Il server eredita le logiche di connessione, handshake e decoding dal protocollo precedentemente introdotto. In aggiunta è presente la logica per la gestione dei vari eventi WebRTC.

```
import socket
import json

from core.websocket import WebSocket

MAX_USERS = 2

rooms = {} # tracks rooms and users inside the rooms
```

```
clients = {} #stores username and socket


def send_to_room(roomid, message):
    if rooms[roomid]:
        for user in rooms[roomid]:
            ws.send(get_user_client(user), json.dumps(message))

def send_log(client, message):
    ws.send(client, json.dumps({
        "type": "log",
        "message": message
    }))

def get_user_client(user):
    """Return user socket given their username if its stored already"""
    for username, client in clients.items():
        if user == username:
            return client

def get_client_username(socket):
    """Retrieve username of user given their socket"""
    for username, client in clients.items():
        if client == socket:
            return username

def remove_from_room(username):
    """Remove a user from room given their username"""
    for room_id, users in rooms.items():
        if username in users:
            index = rooms[room_id].index(username)
            rooms[room_id].pop(index)

def remove_empty_room():
    """Removes room if its empty"""
    rooms_to_remove = []
    for room_id, users in rooms.items():
        if not users:
            rooms_to_remove.append(room_id)

    for room_id in rooms_to_remove:
        rooms.pop(room_id, None)


def send_to_user(username, message):
    """Given their username, send message only to that user"""
    client = get_user_client(username)
    ws.send(client, json.dumps(message))

def onmessage(client: socket.socket, message: str):
    try:
        # parses the values that are being sent from the frontend
        type, params = json.loads(message).values()

        if type == "join":
            username = params["username"]
            room_id  = params["room_id"]

            # if room is not stored
            if not room_id in rooms.keys():
```

```
                if room_id:
                    rooms[room_id] = []
                    send_log(client, "Room does not exist, created now")
                else:
                    send_log(client, "Room ID is undefined")

            if len(rooms[room_id]) >= MAX_USERS:
                send_log(client, "Room is full")
                send_log(client, rooms[room_id]) #sends users connected
                return

            if username not in rooms[room_id]:
                rooms[room_id].append(username)
                clients[username] = client
                send_log(client, f"{username} joined room {room_id}")

                if len(rooms[room_id]) >= MAX_USERS:
                    send_to_room(room_id, {
                        "type": "ready",
                        "callee": f"{username}",
                        "status": "All users have joined"
                    })
            else:
                send_log(client, "User already in room")

        if type == "offer":
            sender = params["sender"]
            target = params["target"]
            sdp = params["sdp"]

            send_to_user(target, {"type": "offer", "sender": sender, "sdp": sdp})

        if type == "answer":
            sender = params["sender"]
            target = params["target"]
            sdp = params["sdp"]

            send_to_user(target, {"type": "answer", "sender": sender, "sdp": sdp})

        if type == "close":
            room_id = params["room_id"]
            send_to_room(room_id, {
                "type": "close",
                "status": "Call has been ended"
            })
    except json.JSONDecodeError as e:
        print(f"Error decoding JSON: {e}")

def onopen(client: socket.socket):
    print(f"Client connected: {client.getpeername()}")


def onclose(client: socket.socket):
    username = get_client_username(client)
    remove_from_room(username=username)
    remove_empty_room()
    print(f"Client [{username}] disconnected - {client.getpeername()}")


if __name__ == '__main__':
```

```
ws = WebSocket("0.0.0.0", 6000)
ws.onmessage(onmessage)
ws.onopen(onopen)
ws.onclose(onclose)
ws.start()
```

3.3 Funzionamento

Un utente può creare una stanza (room) navigando sull'apposita sezione della navbar o entrare in una stanza già creata, se in possesso del link per unirsi.

Alla creazione o entrata in una stanza, un oggetto WebSocket viene istanziato automaticamente e si connette al server signaling.

- L'utente A crea una stanza, l'istanza WebSocket crea una stanza temporanea che sarà chiusa alla disconnessione di entrambi gli utenti. L'utente che ha creato la stanza si collega automaticamente a quest'ultima. Un modale monitora lo status della connessione dell'altro peer (in attesa o connesso).
- L'utente B si collega alla stanza. Tramite WebSocket, viene notificato l'evento e il modale dell'utente A viene aggiornato e permette di avviare la chiamata.

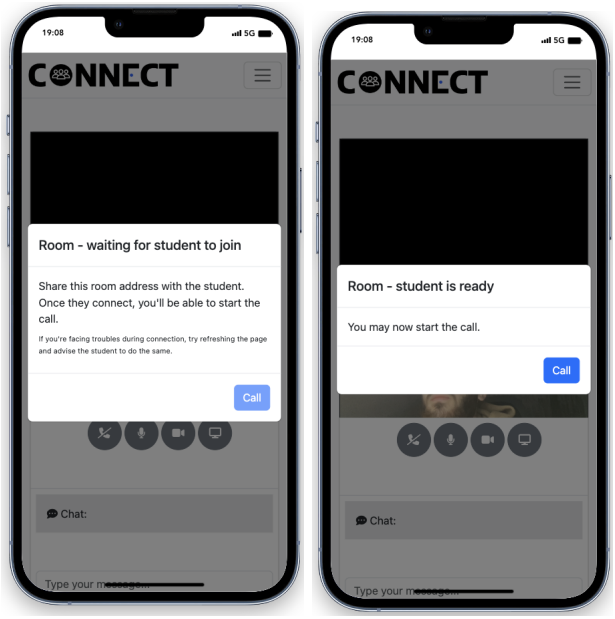


Figura 3.0

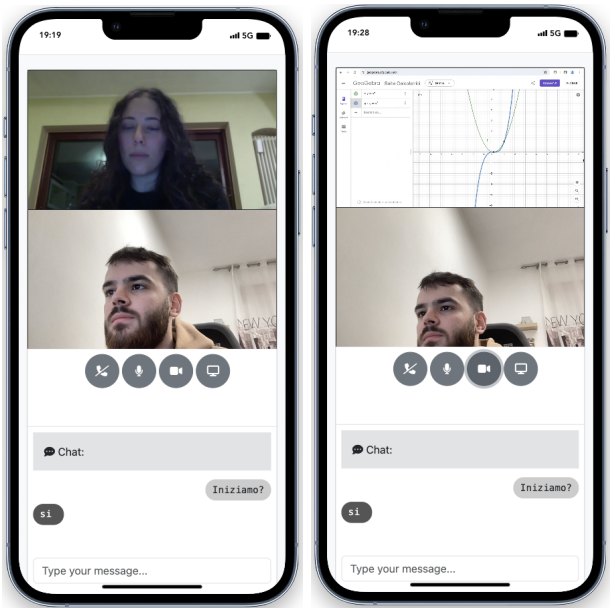

19:19
5G
Chat:
Iniziamo?
si
Type your message...
19:28
5G
Chat:
Iniziamo?
si
Type your message...

Figura 4.0

4 Manuale

1. Clonare il seguente repository e seguire le istruzioni presenti nel file README.md per l'installazione.
2. Navigare nella cartella *"deploy"*.
3. Avviare il docker-compose, eseguendo "docker-compose up -d".
4. Per monitorare i logs dell'applicazione eseguire "docker-compose logs -f –tail 100"
5. Connettersi a *127.0.0.1*. Per testare le funzionalità, è necessario creare due profili.
6. Connettersi a uno dei due profili in navigazione in incognito o tramite un altro browser.
7. Creare la stanza, inserendo nel campo apposito il nome del secondo profilo.
8. Una volta creata la stanza, accedere al link anche dal secondo profilo.