UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

DEPARTMENT: ICT

Network Programming
Media-Sharing Chat Application - Report

Lecturer: *Mr. Huỳnh Vĩnh Nam*

Group Member:

1.	Trần Hải Đăng	23BI14085
2.	Nguyễn Minh Cương	23BI14467
3.	Bùi Tất Đạt	BA12042
4.	Lê Quang Vinh	23BI14457
5.	Đặng Quang Minh	23BI14304
6.	Lê Quý Tôn	23BI14425

1. Overview

- This report describes the design and implementation of a Media-Sharing Chat Application that enables users to communicate via text and share media files in real-time. The application integrates a range of functionalities including secure user authentication, public and private messaging, media file transfers, and history logging. Developed with Python, the system is divided into two main components—a server that handles communication and logic, and a client that provides a graphical user interface (GUI) for user interaction.

2. System Architecture

- The application adopts a client-server architecture. The server is responsible for managing user sessions, authenticating login credentials, handling public and private messaging, processing file uploads and downloads, and maintaining chat logs and credentials. The client is built using Python's **tkinter** module for GUI and enables users to connect to the server, send messages, and share media files. The client also supports caching and previewing downloaded media using the system's default viewer.

a. Client-server architecture overview

```
          |         |              |         |         |
     |         |              |         |         |           |

     |  Client A  |     <=>      |  Server   |   <=>   |  Client B   |

     |         |              |         |         |         |

     (Send Message/Files)          (Authentication)     (Receive Message/File)

   (Preview Downloaded Files)       (Broadcast)           (GUI Display)

                                (Handle File Transfer)    (Request Download)
```

b. Server-side functionality

- Accepting client connections
    - Listen on interface 0.0.0.0, port 3000
    - Create new thread for each client
- User authentication
    - Password is SHA-256-hashed
    - Credentials are stored in creds.txt
- Message handling
    - Public (for everyone)
    - Private (for specific person, format: @username)
- File-related problem
    - Receives files in chunks
    - Saves in server_files directory

- Notification for other users

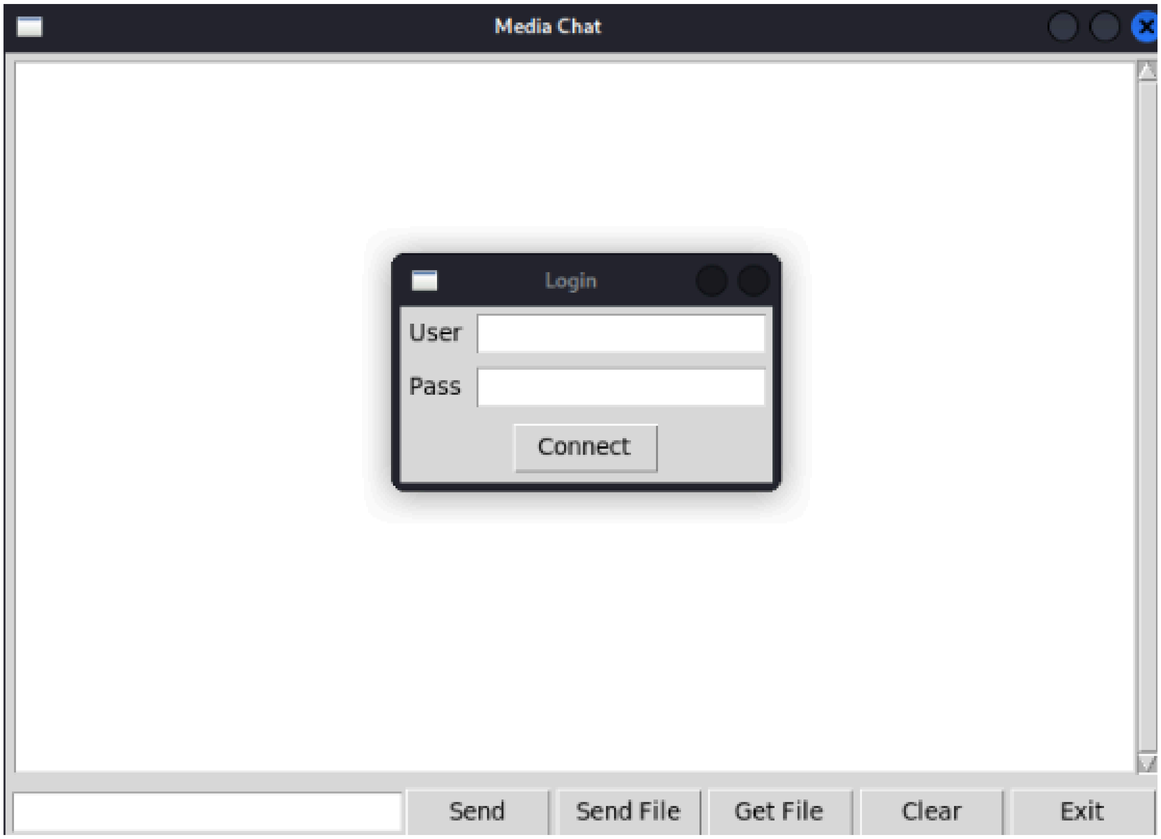
c. Client-side functionality

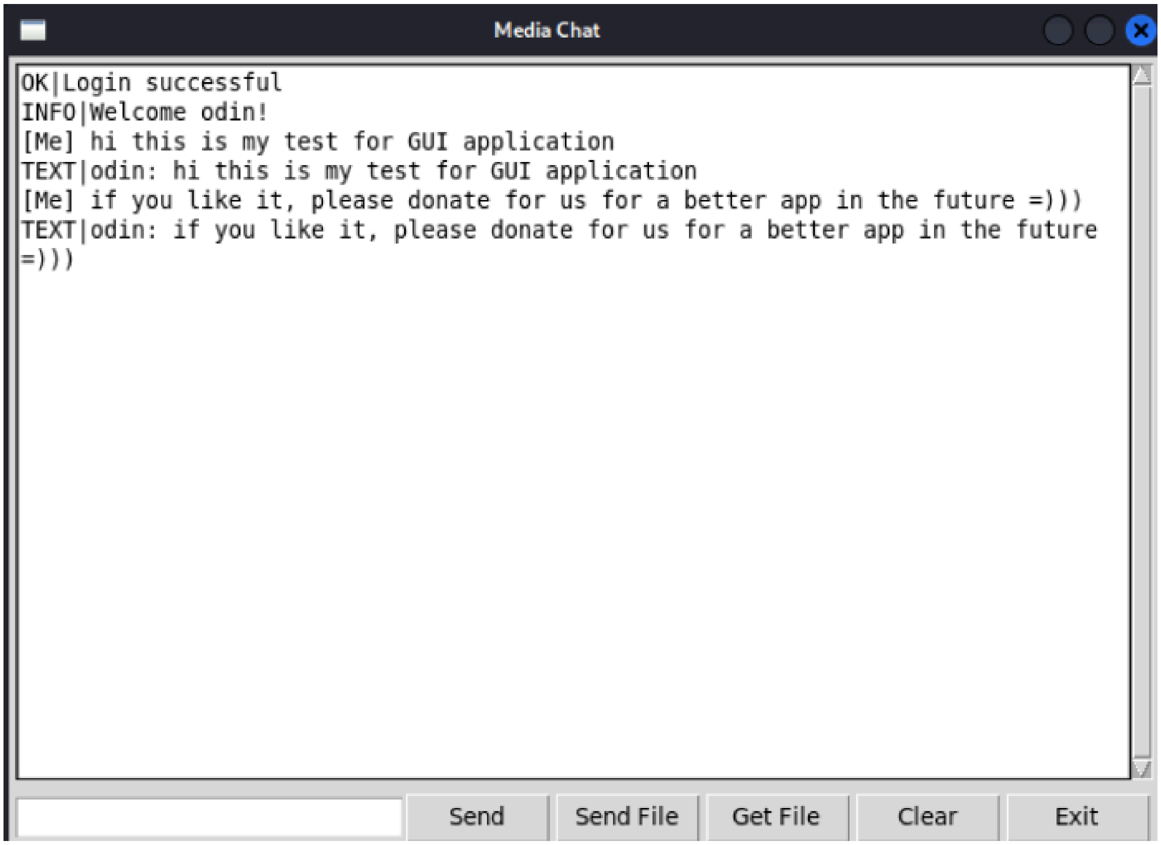
- Connecting to the server (host:no1usth.zapto.org, port: 5000)
- GUI layout (using Tkinter)
- Sending message
    - Use **@username** to send private message
    - Or send public message
- Upload/Download files
    - Use SHA-256 for integrity
    - Files are transmitted in chunks
- Message handling
    - Run in separate thread
    - Display message in GUI

3. Key Features

a. User authentication

- Users log in via username and password, with passwords hashed using SHA-256 for security. New users are auto-registered, while existing users must provide valid credentials.

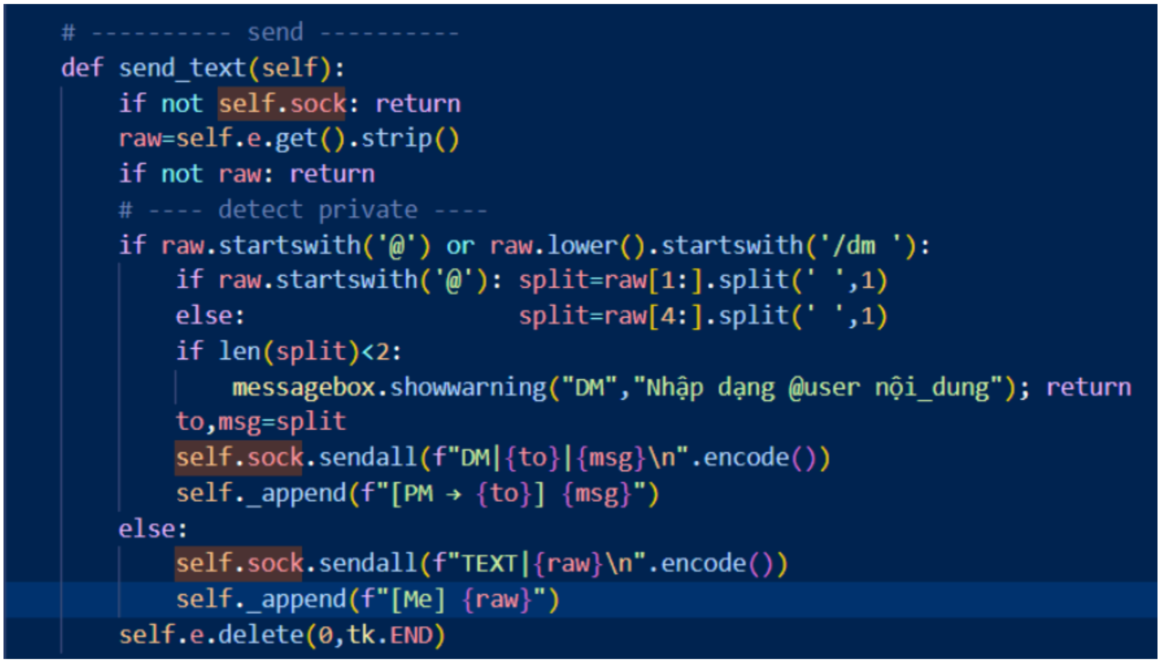




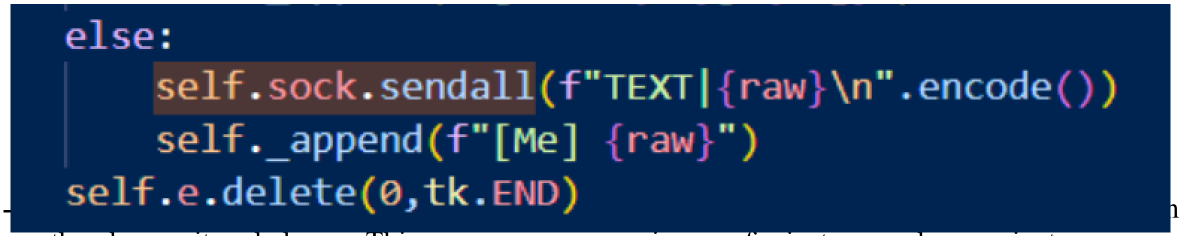
```
OK|Login successful
INFO|Welcome odin!
[Me] hi this is my test for GUI application
TEXT|odin: hi this is my test for GUI application
[Me] if you like it, please donate for us for a better app in the future =)))
TEXT|odin: if you like it, please donate for us for a better app in the future
=)))
```

b. Text Messaging

- The app supports public messages (visible to all) and private messages (sent via @username). Object-oriented programming (OOP) principles ensure secure message handling and visibility control.

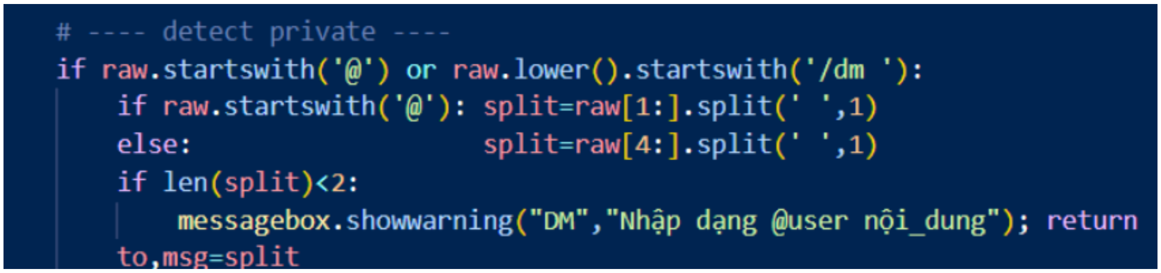


```python
# ---------- send ----------
def send_text(self):
    if not self.sock: return
    raw=self.e.get().strip()
    if not raw: return
    # ---- detect private ----
    if raw.startswith('@') or raw.lower().startswith('/dm '):
        if raw.startswith('@'): split=raw[1:].split(' ',1)
        else:                   split=raw[4:].split(' ',1)
        if len(split)<2:
            messagebox.showwarning("DM","Nhập dạng @user nội_dung"); return
        to,msg=split
        self.sock.sendall(f"DM|{to}|{msg}\n".encode())
        self._append(f"[PM → {to}] {msg}")
    else:
        self.sock.sendall(f"TEXT|{raw}\n".encode())
        self._append(f"[Me] {raw}")
    self.e.delete(0,tk.END)
```

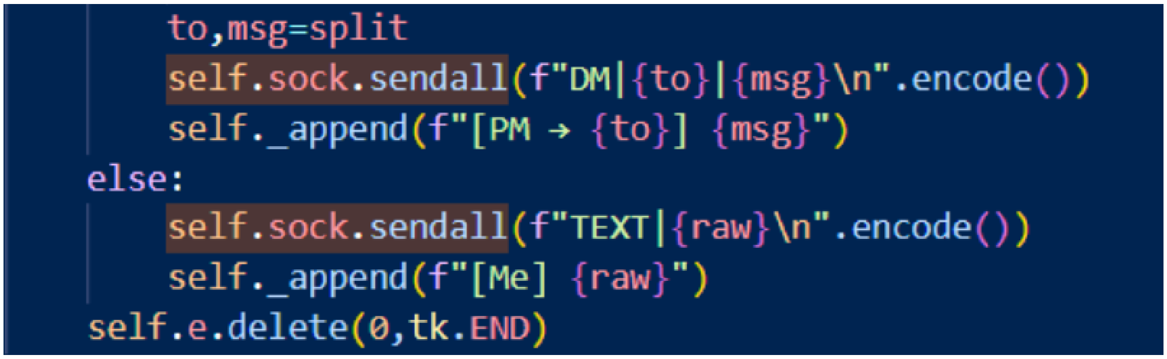


```
else:
    self.sock.sendall(f"TEXT|{raw}\n".encode())
    self._append(f"[Me] {raw}")
self.e.delete(0,tk.END)
```

- hin the class or its subclasses. This ensures message privacy—for instance, when a private message is sent, only the sender and intended recipient can access it, and the system confirms this by showing a notification to the sender only.



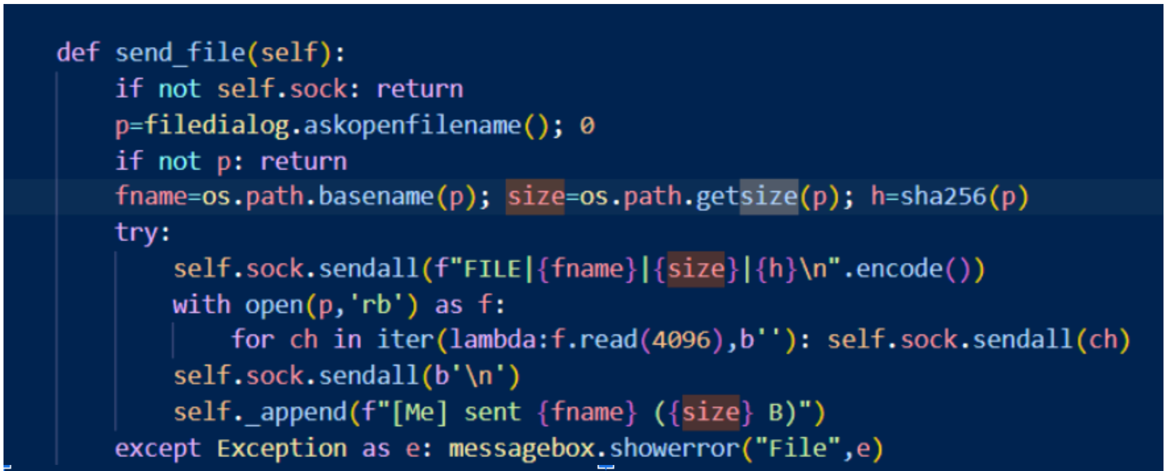
```
# ---- detect private ----
if raw.startswith('@') or raw.lower().startswith('/dm '):
    if raw.startswith('@'): split=raw[1:].split(' ',1)
    else:                   split=raw[4:].split(' ',1)
    if len(split)<2:
        messagebox.showwarning("DM","Nhập dạng @user nội_dung"); return
    to,msg=split
```



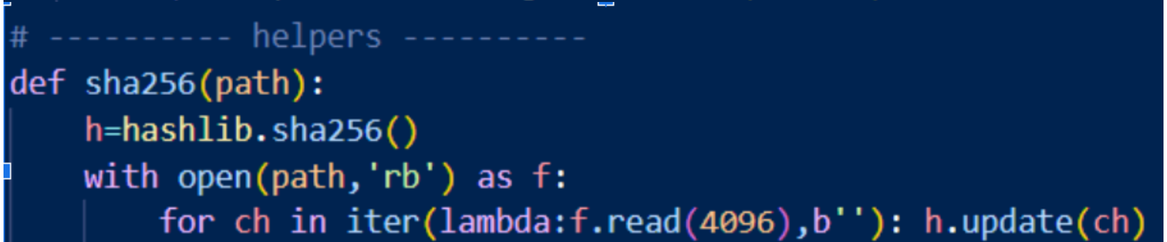
```
        to,msg=split
        self.sock.sendall(f"DM|{to}|{msg}\n".encode())
        self._append(f"[PM → {to}] {msg}")
    else:
        self.sock.sendall(f"TEXT|{raw}\n".encode())
        self._append(f"[Me] {raw}")
    self.e.delete(0,tk.END)
```

c. Media sharing

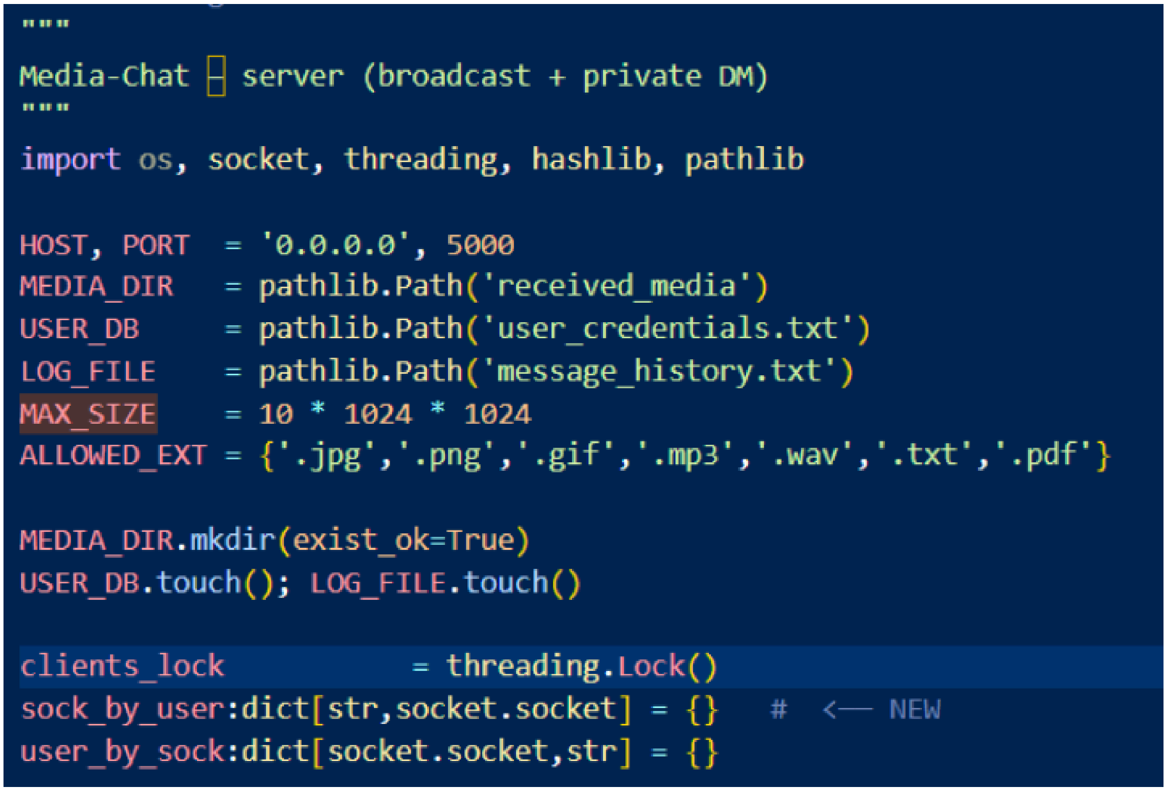
- Users can share media files via a file selection dialog. Metadata (filename, size—capped at 10MB, and SHA-256 hash) is sent first, followed by the file content. Notifications alert users when files are shared, and others can download them by filename.



```
def send_file(self):
    if not self.sock: return
    p=filedialog.askopenfilename(); 0
    if not p: return
    fname=os.path.basename(p); size=os.path.getsize(p); h=sha256(p)
    try:
        self.sock.sendall(f"FILE|{fname}|{size}|{h}\n".encode())
        with open(p,'rb') as f:
            for ch in iter(lambda:f.read(4096),b''): self.sock.sendall(ch)
        self.sock.sendall(b'\n')
        self._append(f"[Me] sent {fname} ({size} B)")
    except Exception as e: messagebox.showerror("File",e)
```



```
# ---------- helpers ----------
def sha256(path):
    h=hashlib.sha256()
    with open(path,'rb') as f:
        for ch in iter(lambda:f.read(4096),b''): h.update(ch)
```



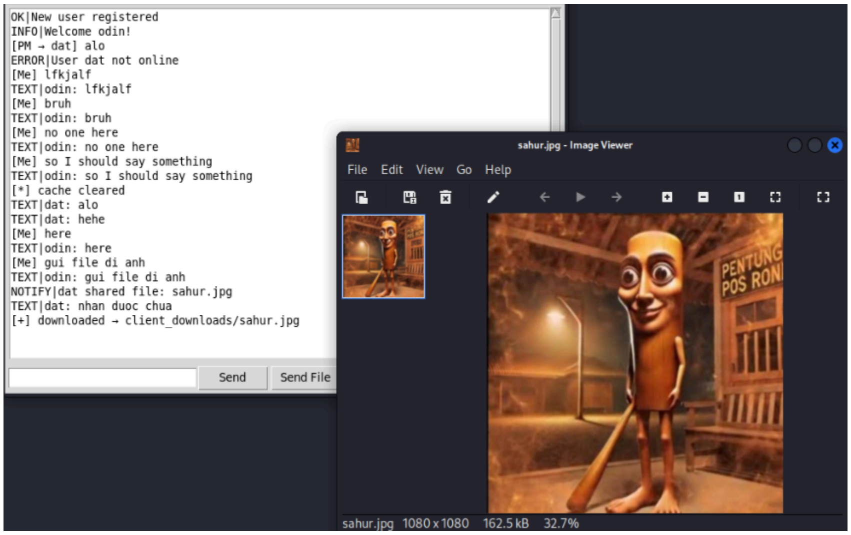
```python
"""
Media-Chat □ server (broadcast + private DM)
"""
import os, socket, threading, hashlib, pathlib

HOST, PORT   = '0.0.0.0', 5000
MEDIA_DIR    = pathlib.Path('received_media')
USER_DB      = pathlib.Path('user_credentials.txt')
LOG_FILE     = pathlib.Path('message_history.txt')
MAX_SIZE     = 10 * 1024 * 1024
ALLOWED_EXT = {'.jpg','.png','.gif','.mp3','.wav','.txt','.pdf'}

MEDIA_DIR.mkdir(exist_ok=True)
USER_DB.touch(); LOG_FILE.touch()

clients_lock           = threading.Lock()
sock_by_user:dict[str,socket.socket] = {}   # <── NEW
user_by_sock:dict[socket.socket,str] = {}
```

- The metadata is structured as FILE|filename|size|hash, followed by the file's binary content and a newline delimiter. When a user shares a file, a NOTIFY|<user> shared file: <filename> message appears. To download, click **Get File**, enter the filename, and the file will be saved locally.

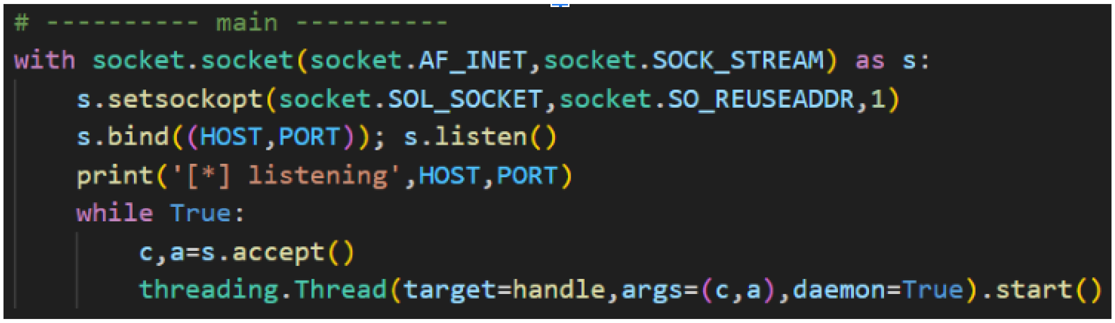


```
OK|New user registered
INFO|Welcome odin!
[PM → dat] alo
ERROR|User dat not online
[Me] lfkjalf
TEXT|odin: lfkjalf
[Me] bruh
TEXT|odin: bruh
[Me] no one here
TEXT|odin: no one here
[Me] so I should say something
TEXT|odin: so I should say something
[*] cache cleared
TEXT|dat: alo
TEXT|dat: hehe
[Me] here
TEXT|odin: here
[Me] gui file di anh
TEXT|odin: gui file di anh
NOTIFY|dat shared file: sahur.jpg
TEXT|dat: nhan duoc chua
[+] downloaded → client_downloads/sahur.jpg
```

4. Implementation Details

Server-Side Implementation

- Manages multiple client threads.
  - The server uses Python’s threading module to handle multiple clients simultaneously. When a client connects, the server spawns a dedicated thread to manage communication, preventing blocking and ensuring responsiveness.



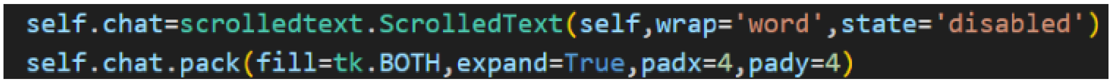
```
# ---------- main ----------
with socket.socket(socket.AF_INET,socket.SOCK_STREAM) as s:
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind((HOST,PORT)); s.listen()
    print('[*] listening',HOST,PORT)
    while True:
        c,a=s.accept()
        threading.Thread(target=handle,args=(c,a),daemon=True).start()
```

- Stores and verifies hashed credentials.

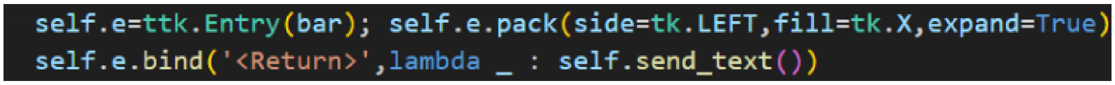
- Handles file storage and logging.

Client-Side Implementation

- GUI includes message display, input field, and file-sharing buttons.
  - Chat display area: Implemented using a tkinter.scrolledtext.ScrolledText widget; for example, the client initializes the chat view with it. The state='disabled' setting prevents manual editing, so messages are inserted programmatically via the _append method, which temporarily enables the widget, inserts the new line, then disables it again.

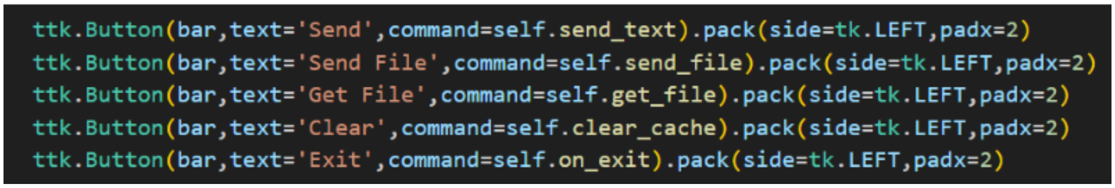


```
self.chat=scrolledtext.ScrolledText(self,wrap='word',state='disabled')
self.chat.pack(fill=tk.BOTH,expand=True,padx=4,pady=4)
```



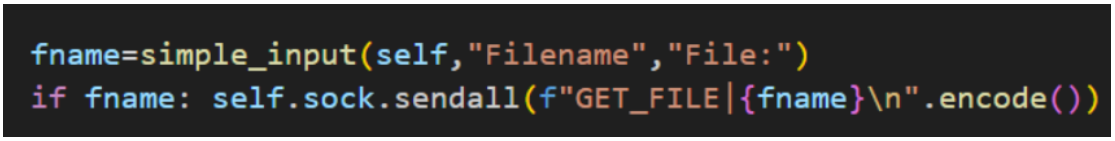
```
self.e=ttk.Entry(bar); self.e.pack(side=tk.LEFT,fill=tk.X,expand=True)
self.e.bind('<Return>',lambda _ : self.send_text())
```

  - Message input field: A single-line ttk.Entry widget lets the user type messages and expands horizontally. It is created as follows, with the Enter key (<Return>) bound to the send_text() method, allowing messages to be sent by pressing Enter.
  - Buttons for messaging and file sharing: The interface includes buttons for sending text and files. For example, the Send button triggers send_text(), the Send File button calls send_file() to open a file dialog and upload a file, and the Get File button calls get_file() to request a file from the server. All buttons, including Clear and Exit (for cache management and exiting), are packed into the same toolbar frame (bar) for a consistent layout.



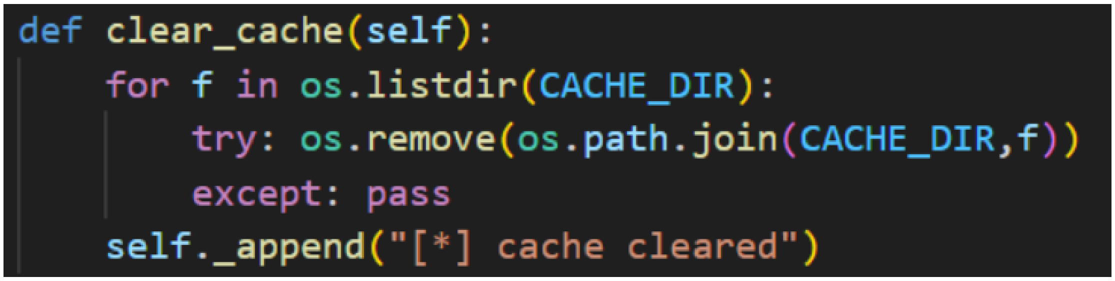
```
ttk.Button(bar,text='Send',command=self.send_text).pack(side=tk.LEFT,padx=2)
ttk.Button(bar,text='Send File',command=self.send_file).pack(side=tk.LEFT,padx=2)
ttk.Button(bar,text='Get File',command=self.get_file).pack(side=tk.LEFT,padx=2)
ttk.Button(bar,text='Clear',command=self.clear_cache).pack(side=tk.LEFT,padx=2)
ttk.Button(bar,text='Exit',command=self.on_exit).pack(side=tk.LEFT,padx=2)
```

    - File download and cache controls:
        - Downloaded files are saved in a local cache directory (**client_downloads**). The **Get File** action uses:



```
fname=simple_input(self,"Filename","File:")
if fname: self.sock.sendall(f"GET_FILE|{fname}\n".encode())
```

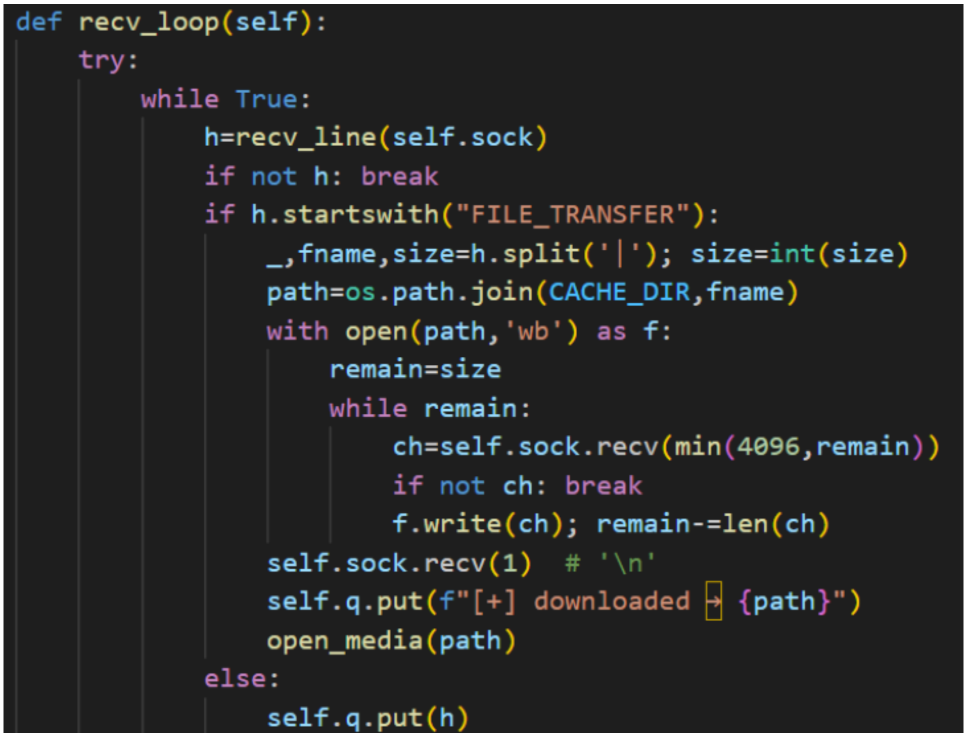
        - After receiving a file, the client writes it to **client_downloads** and opens it with the system viewer. A **Clear** button allows cache cleanup:



```
def clear_cache(self):
    for f in os.listdir(CACHE_DIR):
        try: os.remove(os.path.join(CACHE_DIR,f))
        except: pass
    self._append("[*] cache cleared")
```

        - Clicking **Clear** deletes all files in the cache directory and appends a "**[*] cache cleared**" message in the chat area to notify the user.
- Uses threads and queues for asynchronous communication.

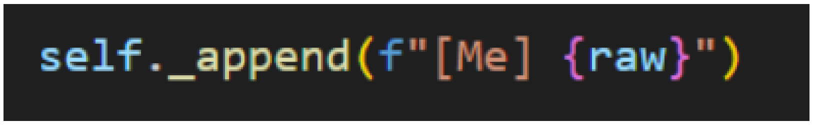
The client uses a queue.Queue() (self.q) to handle asynchronous data. After login, a background thread runs recv_loop(), reading from the socket and putting messages into the queue. Meanwhile, poll_q() (called via self.after(100, self.poll_q)) periodically checks the queue, retrieves pending data, and passes it to _append() for display. This mechanism decouples network I/O from GUI updates, ensuring thread-safe processing.



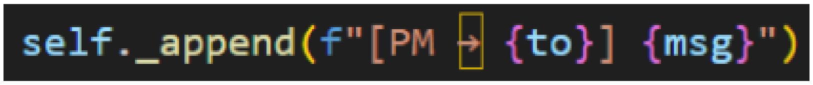
```
def recv_loop(self):
    try:
        while True:
            h=recv_line(self.sock)
            if not h: break
            if h.startswith("FILE_TRANSFER"):
                _,fname,size=h.split('|'); size=int(size)
                path=os.path.join(CACHE_DIR,fname)
                with open(path,'wb') as f:
                    remain=size
                    while remain:
                        ch=self.sock.recv(min(4096,remain))
                        if not ch: break
                        f.write(ch); remain-=len(ch)
                self.sock.recv(1)  # '\n'
                self.q.put(f"[+] downloaded → {path}")
                open_media(path)
            else:
                self.q.put(h)
```

- Automatically opens downloaded files post-transfer.

The client provides immediate feedback by prefixing messages in the chat. For example, public messages use the [Me] tag, private messages use [PM], downloaded files are marked with [+], clearing the cache shows [*], and connection errors display [!] disconnected. These visual tags clearly indicate the type and status of each action in the chat window.



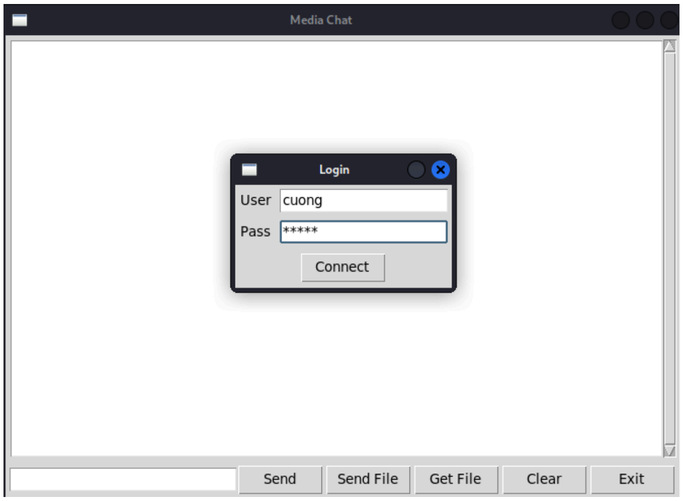
```
self._append(f"[Me] {raw}")
```



```
self._append(f"[PM → {to}] {msg}")
```

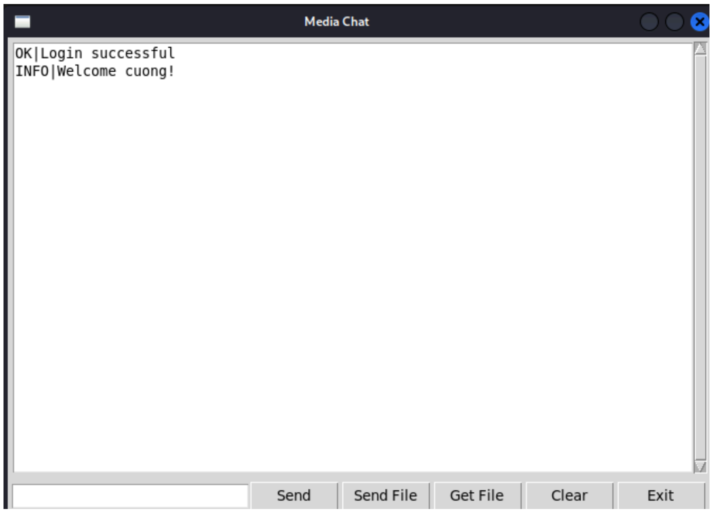
5. Example Usage

- **Login**: Enter credentials → "Login successful" confirmation.



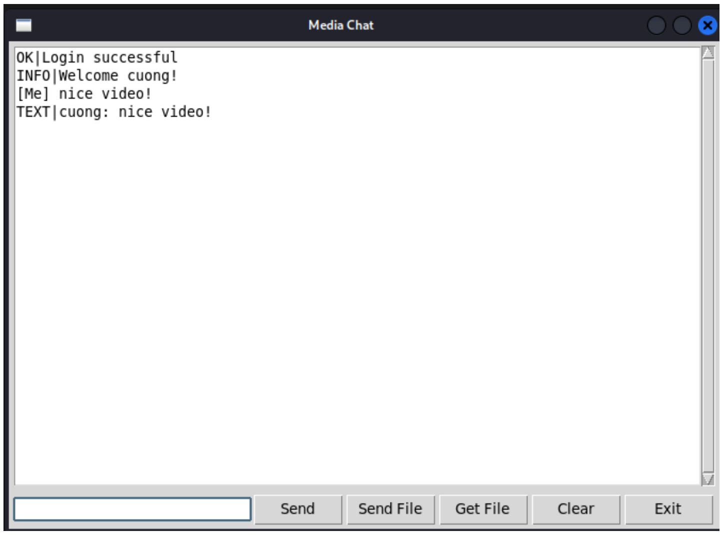
User cuong

Pass *****



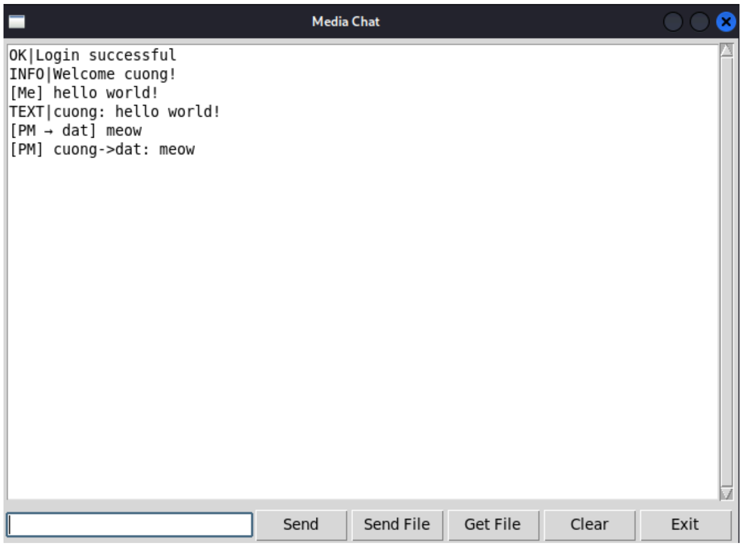
```
OK|Login successful
INFO|Welcome cuong!
```

- **Send Message**: Type text → appears in chat.



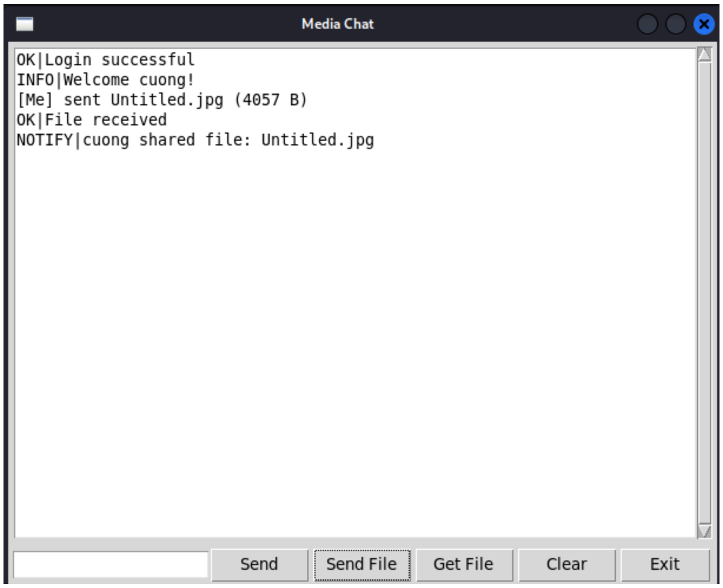
```
OK|Login successful
INFO|Welcome cuong!
[Me] nice video!
TEXT|cuong: nice video!
```

- **Private Message**: Use @username → only recipient sees it.



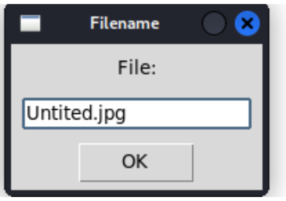
```
OK|Login successful
INFO|Welcome cuong!
[Me] hello world!
TEXT|cuong: hello world!
[PM → dat] meow
[PM] cuong->dat: meow
```

- **Share File**: Select file → notification appears.

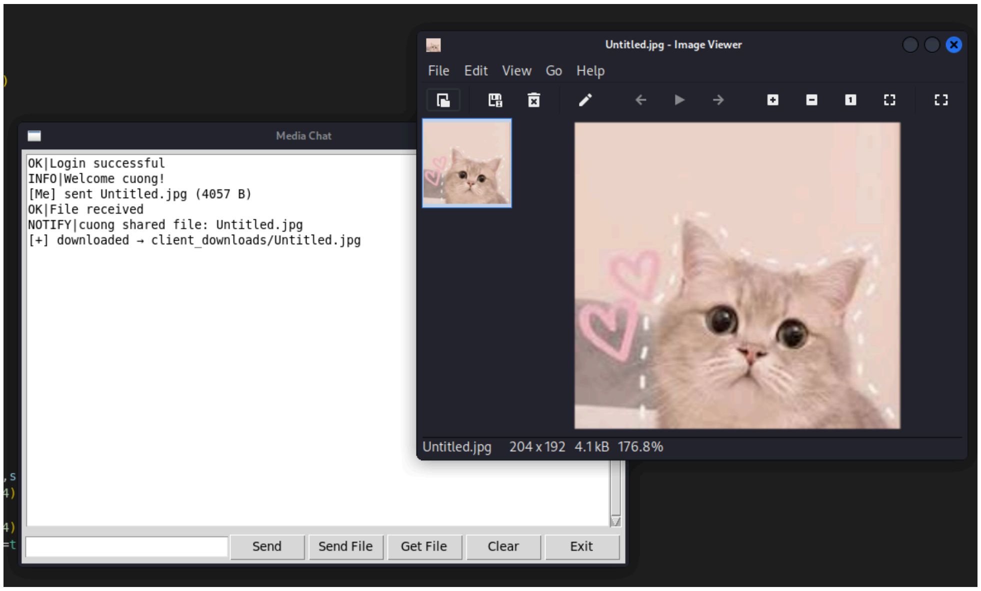


```
OK|Login successful
INFO|Welcome cuong!
[Me] sent Untitled.jpg (4057 B)
OK|File received
NOTIFY|cuong shared file: Untitled.jpg
```

- **Download File**: Enter filename → file downloads and opens.



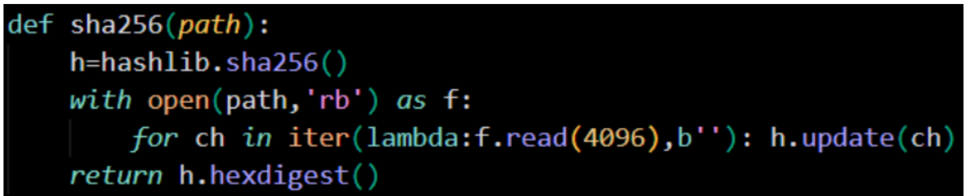
File: Untited.jpg



```
OK|Login successful
INFO|Welcome cuong!
[Me] sent Untitled.jpg (4057 B)
OK|File received
NOTIFY|cuong shared file: Untitled.jpg
[+] downloaded → client_downloads/Untitled.jpg
```

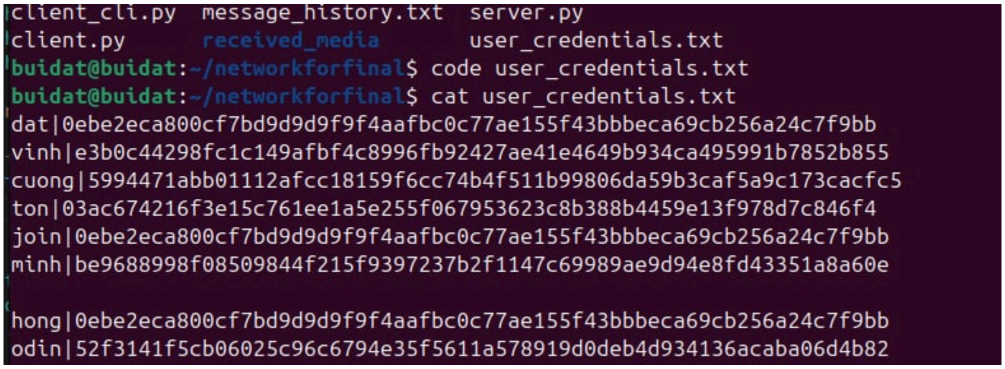
6. Security Considerations

- **Password Security**: SHA-256 hashing prevents plaintext exposure.



```python
def sha256(path):
    h=hashlib.sha256()
    with open(path,'rb') as f:
        for ch in iter(lambda:f.read(4096),b''): h.update(ch)
    return h.hexdigest()
```

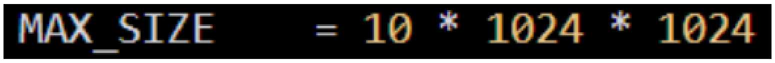
  - The hash is then stored in a file called **user_credentials.txt**
  - This approach ensures that even if the credentials file is compromised, the actual passwords remain undisclosed, significantly reducing the risk of credential leakage or replay attacks.
  - File encryption password:



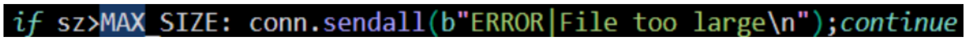
```
client_cli.py  message_history.txt  server.py
client.py      received_media       user_credentials.txt
buidat@buidat:~/networkforfinal$ code user_credentials.txt
buidat@buidat:~/networkforfinal$ cat user_credentials.txt
dat|0ebe2eca800cf7bd9d9d9f9f4aafbc0c77ae155f43bbbeca69cb256a24c7f9bb
vinh|e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
cuong|5994471abb01112afcc18159f6cc74b4f511b99806da59b3caf5a9c173cacfc5
ton|03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4
join|0ebe2eca800cf7bd9d9d9f9f4aafbc0c77ae155f43bbbeca69cb256a24c7f9bb
minh|be9688998f08509844f215f9397237b2f1147c69989ae9d94e8fd43351a8a60e

hong|0ebe2eca800cf7bd9d9d9f9f4aafbc0c77ae155f43bbbeca69cb256a24c7f9bb
odin|52f3141f5cb06025c96c6794e35f5611a578919d0deb4d934136acaba06d4b82
```

- **File Upload Restrictions**: 10MB limit mitigates denial-of-service risks.



```
MAX_SIZE    = 10 * 1024 * 1024
```



```
if sz>MAX_SIZE: conn.sendall(b"ERROR|File too large\n");continue
```

- **File Integrity Check**: SHA-256 comparison detects transmission errors.

7. Limitations and Future Work

Current Limitations

- No end-to-end encryption for messages.

- Basic authentication (no salting).

- Lacks advanced features like group chats or emojis.

Planned Enhancements

- Add end-to-end encryption and password salting.

- Introduce group chats, emojis, and message reactions.

- Improve media previews and file type support.

8. Conclusion

The Media-Sharing Chat Application delivers secure messaging and file-sharing in an intuitive GUI. Its modular design allows for future upgrades in security, usability, and user experience.