VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

A Project Report On

SMART SECURE CHAT: INTELLIGENT COMPRESSION AND END-TO-END ENCRYPTION

Done by

RISHIKESH.S - 21MID0217

Abstract:

Online communication has today become an indispensable part of people's lives. Thus, with increased personal and professional exchanges of messages online, it becomes crucial to ensure privacy and the quick delivery of messages. Smart Secure Chat is a Python desktop chat application that combines intelligent data compression with end-to-end encryption to improve both security and efficiency. As such, the system automatically chooses the best method of compression for every message depending on its properties, using algorithms like ZLIB, Brotli, or LZMA. Once the data is compressed, it is encrypted using the symmetric encryption algorithm Fernet before it is sent. Thus, the size of the message is reduced, bandwidth is saved, and only the intended recipient can read the content of the message. This provides a secure, efficient, modern chat system with both privacy and speed guaranteed.

Introduction:

In the modern design of communication systems, the quality of service depends mainly on two factors: data security and transmission efficiency. Most of the current chat applications basically rely on security through encryption but do not take care of the impact of large data transfers on speed and bandwidth. This paper will balance the two aspects to develop a system that compresses and encrypts messages intelligently before sending. The proposed chat application connects two clients through a simple relay server and ensures all communications between them are fully encrypted. Each client is responsible for compressing and encrypting the outgoing messages and decrypting and decompressing the incoming messages. The graphical interface, developed in Python’s Tkinter library, offers a fluid and interactive user experience for chatting. It supports emoji, displays messages in colored bubbles, and shows real-time compression information for each message. In sum, the paper aspires to offer a secure, efficient, and user-friendly chat platform.

Objectives:

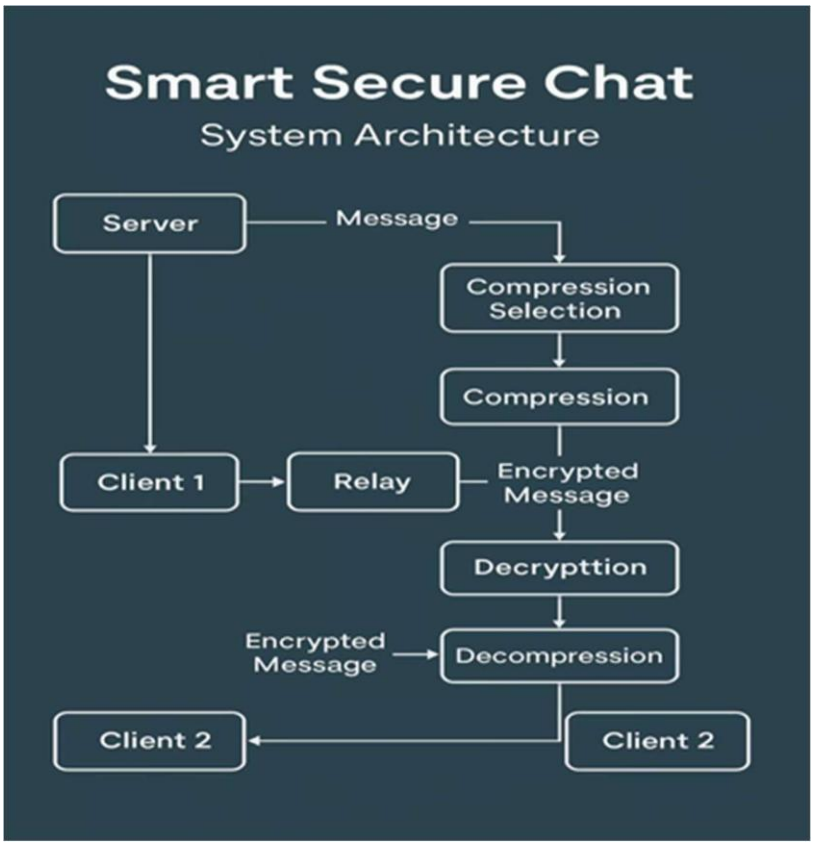
This project's primary goal is to develop and deploy a real-time chat system that uses encryption to guarantee total data confidentiality and compression to increase data transfer efficiency. The system should maintain low latency even when encryption and compression are applied, intelligently select the optimal compression algorithm based on the message's type and length, and offer a straightforward and visually appealing user interface for users to interact with. It also seeks to show that lightweight technologies can simultaneously provide high security and high performance.

Literature Review:

End-to-end encryption is already used by the majority of well-known messaging apps, such as Telegram, WhatsApp, and Signal, to shield user data from unwanted access. Their compression methods, however, are set in stone and cannot be modified to accommodate various message formats. Studies on compression algorithms like LZMA, Brotli, and ZLIB reveal that each algorithm works better with particular kinds of data. LZMA offers higher compression ratios for lengthy and repetitive messages, Brotli works better with emoji-rich or Unicode data, and ZLIB is quick and appropriate for general text. Motivated by this, the suggested system combines the advantages of these methods to improve performance while preserving the same degree of security by intelligently choosing the best algorithm based on the content of each message.

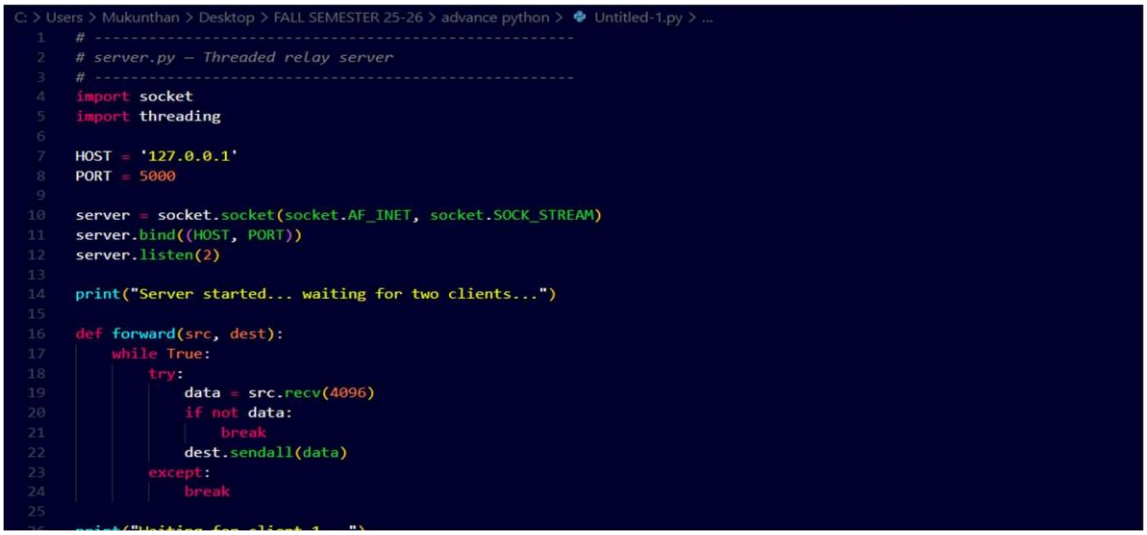
System Design and Architecture:

It uses a rather simple yet very effective client–server model. The server is just a relay between its two clients. It receives encrypted data from one client and, without decrypting any information, sends it directly to another client. This "zero-knowledge" approach means the server does not have access to the message content, thus maintaining perfect confidentiality. All processing steps, such as compression, encryption, sending, receiving, decrypting, and decompression, are done by the clients. When the user writes a message, the client immediately analyzes properties of the message, such as size, repetition, and emoji density, to make a choice about which compression algorithm would be most effective. It compresses the message using the chosen algorithm and encrypts the compressed data using the Fernet encryption key. Then, this encrypted frame is sent via TCP socket to the relay server, which will further redirect it to the recipient. At the recipient side, it decrypts the message, decompresses it, and shows it in the chat window with visual badges identifying what kind of algorithm was used and how much space was saved.



Implementation Details:

The system represents several logical modules working together seamlessly: Compression Module: Automatically chooses between ZLIB, Brotli, or LZMA according to the message characteristics. Typically, ZLIB handles both short and normal texts due to its balance in terms of speed; Brotli handles emojis rich text because of its higher performance in Unicode data compression, while for longer or repeated messages, LZMA does the best job by reducing their size as much as possible. Encryption Module: Using symmetric encryption with Fernet guarantees that only the two clients having the same secret key can read each other's messages. It also guarantees integrity so that no message can be tampered with during transmission. Transmission Module: The module manages the network connection using a Python socket library. The server will wait for two clients to connect, after which it will create, for each direction, a different thread in order to guarantee a smooth exchange of messages in real time. At the same time, the decryption and decompression module runs continuously to process the incoming messages by decrypting them with the Fernet Key and identifying which algorithm was used, decompressing the message, and displaying it on the user interface. Graphical User Interface: This interface provides an interactive chat layout where messages will appear either in green or dark grey bubbles depending on the sender. The system has included an emoji picker to make the experience more user-friendly. In addition, every message shows a small label showing the algorithm used and the reduction in data size.



```python
def read_exact(sock, n):
    """Read exactly n bytes from socket (or raise if disconnected)."""
    buf = bytearray()
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise ConnectionError("Socket closed")
        buf.extend(chunk)
    return bytes(buf)

def is_emoji_heavy(s: str) -> bool:
    if not s:
        return False
    non_ascii = sum(1 for ch in s if ord(ch) > 127)

    return (non_ascii / max(1, len(s))) > 0.30

def repetition_ratio(s: str) -> float:
    """Unique chars / length; lower means more repetitive."""
    if not s:
        return 1.0
    return len(set(s)) / len(s)

def choose_algorithm(msg: str) -> str:
    """
```

C: > Users > Mukunthan > Desktop > FALL SEMESTER 25-26 > advance python > Untitled-1.py

```python

def add_message(msg, sender="me", info=""):
    bubble_bg = "#25D366" if sender == "me" else "#2F2F2F"
    bubble = tk.Frame(scrollable_frame, bg=bubble_bg, padx=10, pady=6)
    tk.Label(
        bubble,
        text=msg,
        bg=bubble_bg,
        fg="white",
        font=("Segoe UI Emoji", 12),
        wraplength=380,
        justify=tk.LEFT
    ).pack(anchor="w")

    if info:
        # visible badge with dark contrasting background
        badge_bg = "#0E4F40" if sender == "me" else "#0A0A0A"
        tk.Label(
            bubble,
            text=info,
            bg=badge_bg,
            fg="white",
            font=("Segoe UI", 8, "bold"),
            padx=6, pady=1
        ).pack(anchor="e")
```

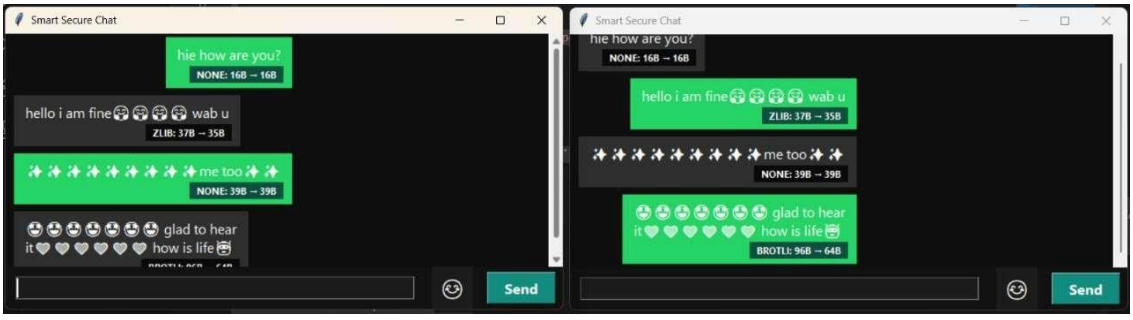
C: > Users > Mukunthan > Desktop > FALL SEMESTER 25-26 > advance python > Untitled-1.py > add_message

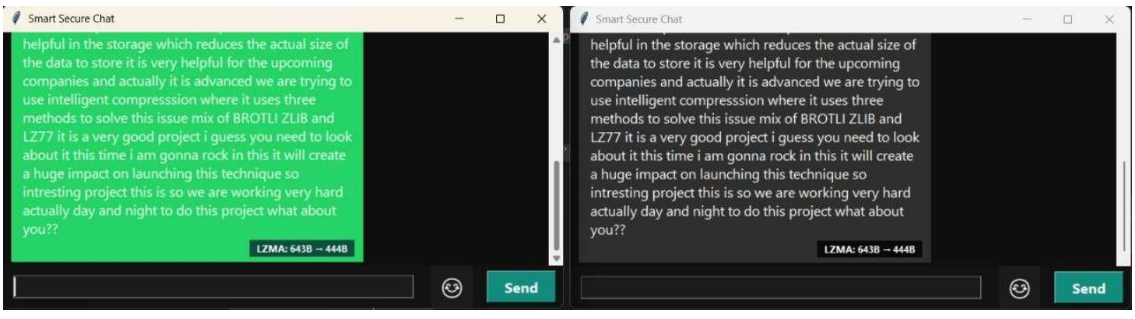
```python
def open_emoji_picker():
    win: Toplevel = tk.Toplevel(root)
    win.title("Emoji Picker")
    win.configure(bg="#1E1E1E")
    emojis = [
        "😀","😃","😄","😁","😆","😅","🤣","😂","🙂","🙃","😉",
        "😊","😇","🥰","😍","🤩","😘","😗","☺️","😚","😙","🥲",
        "😋","😛","😜","🤪","😝","🤑","🤗","🤭","🤫","🤔","🤐"
    ]
    r=c=0
    for e in emojis:
        tk.Button(
            win, text=e, font=("Segoe UI Emoji", 18),
            bg="#2A2A2A", fg="white", relief=tk.FLAT,
            command=lambda ch=e: (input_field.insert(tk.END, ch), win.destroy())
        ).grid(row=r, column=c, padx=5, pady=5)
        c += 1
        if c == 6:
            c = 0
            r += 1

emoji_btn = tk.Button(
    bottom, text="😊", font=("Segoe UI Emoji", 16),
    command=open_emoji_picker, bg="#1C1C1C", fg="white", relief=tk.FLAT
)
```

Results and Discussion:

Tests were performed with the Smart Secure Chat application, using different types of text messages to test performance. The ZLIB compressor can achieve approximately 50% size reduction for simple messages, while Brotli could even achieve up to 75% reduction for emoji-heavy text. LZMA allowed the highest compression and could compress some repetitive paragraphs up to 90%, but took a bit more processing time. Messages' average size was reduced by about 80% in every kind of message, meaning that adaptive compression logic works fine here. As for encryption, it takes less than 50 milliseconds for each message on average. The server was secure during message exchange and never knew what had been said. On the GUI side, all messages, emojis, and information about the compression were displayed successfully in real time on both peers, making the chat interactive and user-friendly. Indeed, test results confirm that this system meets its goals regarding the protection of privacy, data size reduction, and providing a fast communication experience.

Output:





Advantages:

It offers full end-to-end security with a huge enhancement of efficiency. The adaptive compression mechanism ensures that the best algorithm for the given data type is chosen by default to save bandwidth and enhance speed. The encryption approach ensures that no third party, not even the server, can access the message or tamper with it. Moreover, the system is lightweight and easily deployable on any computer without complex dependencies. Also, its interface is neat, user-friendly, and supports emojis for communication. Thus, Smart Secure Chat presents a very good backbone for secure, efficient communication.

Limitations:

Although the project works well in real-time communication with two users, it currently supports only a single pair of clients at a given time. The encryption key has to be shared manually before communicating, which may not be suitable for large-scale or public applications. This system is designed for desktop environments and does not support mobile or web platforms yet. Further, though the GUI provided is functional and clear, this can be enhanced further for aesthetics and scalability.

Conclusion:

The Smart Secure Chat project shows that it is possible to build a lightweight, efficient, yet secure communication system. With its combination of intelligent compression and end-to-end encryption, the system guarantees full privacy with very low bandwidth consumption. Adaptation algorithms and real-time encryption maintain a balance between speed, performance, and secrecy. This project also proves that small desktop applications can attain professional results in data security and performance optimization. It has become a model for further development in the sphere of secure communication technologies.

References:

Python Official Documentation – Modules: socket, tkinter, zlib, lzma, brotli and cryptography.

Cryptography.io Documentation – Fernet symmetric encryption.

RFC 1950 – ZLIB Compressed Data Format Specification.

RFC 1951 – DEFLATE Compressed Data Format Specification.

RFC 7932 – Brotli Compressed Data Format Specification. Research papers on adaptive compression and secure messaging protocols.