

# Project Deliverables

-z5522906

Github Repo -

<https://github.com/jatinSharma-create/6441---Project---E2E-Chat>

## **1. Aim**

In this project I built a small end-to-end encrypted chat system to *show* the core security principles—confidentiality, integrity, and a (limited but still demo-able) availability—plus a manual authentication step, instead of only talking theory. The design uses X25519 for key agreement, HKDF to derive session keys, and NaCl's XChaCha20-Poly1305 for authenticated encryption. I also plugged a kind of Double-Ratchet style layer into a Python TUI client and a minimal asyncio TCP relay. During testing I captured PCAP traces, deliberately triggered MAC verification fails, and measured how the relay behaves when one peer suddenly drops out. This report summarises the motivation and method, the evidence collected, issues and ethics I ran into, and what I actually learnt over ~30 hours (approx) of work.

## **2. Motivation & Need**

The main motivation of this project was basically two-fold:

- (1) I wanted to actually see and understand the moving pieces inside “secure chat”, not just repeat the buzzwords; and
- (2) I needed to demonstrate those parts clearly for the course criteria. Rather than trying to ship a full production-grade system (which would've been overkill and there was time constraint as well),

I targeted a minimal proof-of-concept that lines up directly with CIA plus authentication. So I kept the server - relay super simple and spent more time thinking through the crypto logic/reasoning. That was the whole goal, honestly.

### **3. Background, Crypto Choices & Quick Definitions**

For the crypto stack I looked at a few modern primitives and picked ones that are both well-regarded and (kinda) straightforward to derive from Python. Briefly:

- **X25519 (Diffie–Hellman on Curve25519)**: Fast scalar mult to derive a shared secret without exposing private keys. It's pretty much the de-facto standard in Signal/Noise etc, and PyNaCl gives a clean API, so I chose it over older stuff like RSA.
- **HKDF (HMAC-based Key Derivation Function)**: takes that single shared secret and stretches it into lots of separate keys (root, chain, message...). The Double-Ratchet burns through keys a lot, so HKDF was necessary; otherwise I'd be reusing entropy which was bad news.
- **XChaCha20-Poly1305 (via NaCl SecretBox)**: an AEAD cipher: ChaCha20 handles confidentiality, Poly1305 adds integrity (a MAC). The huge nonce space means I'm less likely to screw up nonce reuse, and the API is honestly hard to misuse (though not impossible).
- **Double-Ratchet (Signal-style)**: after the first X25519 exchange, every message "ratchets" forward (basically another DH + HKDF hop) so we get forward secrecy and avoid key reuse. I only did a trimmed version—no skipped-message queue, minimal state—but it still rotates per-message keys.

I briefly thought about just using TLS on the socket or even using RSA, but that hides the exact pieces I wanted to learn about. MLS looked good too, but it's way too heavy for ~30 hours and a 2-person chat demo.

### **4. Problem Statement & Objectives**

The core problem was to build a tiny chat system where only the two endpoints can actually read the messages, any tampering gets caught, and the middle relay can't be trusted at all. On top of that, I had to *prove* each security property with real evidence (PCAP dumps, logs, screenshots), not just hand-wavy claims.

**Objectives:**

- **Confidentiality:** No plaintext should appear on the network trace. I showed this in Wireshark PCAPs.
- **Integrity:** If I flip bits in a ciphertext, the MAC check must fail and the client refuses it.
- **Availability:** The relay process keeps running even when clients join/leave. It might drop a few packets, but it doesn't just die.
- **Authentication:** Users manually verify each other's public-key fingerprints.
- **Forward Secrecy:** Implemented via the mini Double-Ratchet. I didn't finish a proper replay/compromise demo though—I explained why later and what was missing.

(There's probably still a couple rough edges, but that was the target set of goals I aimed for.)

## 5. Design Overview (high level)

There are basically three moving parts:

1. **Client (Alice/Bob):** A Python TUI built with Rich. Each client makes a static X25519 keypair, prints its own + the peer's fingerprint, and then uses a kind of Double-Ratchet to encrypt every outgoing message.
2. **Relay (server.py):** An untrusted asyncio TCP forwarder. It just keeps a tiny buffer of pubkeys and pipes framed messages around. No database, no disk writes, nothing complicated.
3. **Attacker (conceptual):** Can sniff packets, drop/replay them, or even flip bits. But they shouldn't be able to decrypt or forge legit ciphertexts (unless I seriously messed up).

The data-flow diagram (threat\_model.png) shows the trusted clients vs the untrusted relay/network, plus where an attacker would sit. Public keys + ciphertext go through the relay, but plaintext is meant to never leave the client processes. (If it does... that would be a bug.)

At the moment the model only spins-up a single 1-to-1 Double-Ratchet session between Alice and Bob. The relay server itself can already broadcast to heaps of sockets, but my client keeps just one active ratchet state, so talking to exactly one peer. If I want to scale to  $N$  users I'd pretty much need to hold  $N$  separate

ratchet objects (one per peer) **or** swap in a proper group protocol like MLS, which is very heavier and I haven't looked into it yet.

## **6. Implementation Summary**

### **crypto.py**

- generate\_keypair(), dh() (X25519), and a couple HKDF helpers (hkdf\_root, hkdf\_chain) to stretch secrets.
- RatchetSession: a trimmed Double-Ratchet—each message does a DH hop + HKDF to spit out fresh message keys. It's minimal (no skipped-msg queue etc.) but works for my aims.

### **server.py**

- Uses asyncio.start\_server; keeps an in-memory {cid: writer} dict to track clients.
- Stores 32-byte public keys briefly and just forwards framed packets to everyone else. No database, no files—relay is deliberately dumb/untrusted.

### **client.py**

- On connect it sends its static public key, then reads the peer key. Prints “Identity” (my fingerprint) and “Key Exchange” (their FP).
- After that, a send/recv loop encrypts/decrypts via the ratchet session. If MAC fails, it bails on that packet.

### **tests & helper scripts**

- test\_env.py, test\_e2ee.py: ~5 pytest cases to sanity-check crypto pieces (key sizes, MAC verify, ratchet advancing, etc).
- tamper\_relay.py flips one byte in each ciphertext, forcing MAC errors to prove integrity actually works.
- Forward-secrecy replay scripts were half-done; I parked that demo for later (see Section 10), mostly ran out of time and it got messy.

## **7. Results (Evidence tied to Appendices)**

## **Confidentiality**

Figure 1 (Appendix) shows a Wireshark dump on TCP port 9000 where everything on the wire is just ciphertext—basically hex junk, no readable strings. Evidence: screenshots/pcap\_ciphertext.pcap + screenshot(Appendix Figure 1).

## **Integrity**

Figure 2 (Appendix): when I flip a single byte in `tamper_relay.py`, the receiving client throws a NaCl MAC verification error (so the packet gets rejected). Evidence: Appendix Figure 2.

## **Availability**

Figure 3a (Appendix): The relay keeps running even with 0 peers connected—logs literally say “0 peers” but it doesn’t crash.  
Figure 3b (Appendix): After Bob reconnects, frames start forwarding again normally. Evidence: Screenshot Figure 3b.

## **Authentication**

Figure 4 (Appendix): Both clients print their own fingerprint and the peer’s. Alice’s view of Bob’s FP equals Bob’s self FP (and vice-versa), so the manual check works. Evidence: Figure 4 appendix.

(Forward secrecy live replay demo is missing; see Section 10 for why—I ran out of time and it got messy.)

# **8. Issues, Fixes & Professional / Ethical Bits**

## **Crypto correctness**

I initially messed up HKDF parameters (salt/info didn’t match between sides), which caused early decrypt fails. Wrote a couple of tiny unit tests and standardised those values, then it started working properly.

## **Asyncio messup**

The relay would sometimes crash when a client dropped halfway through a broadcast. I wrapped the writes in try/except and added an `asyncio.Lock` so the shared client map doesn’t get corrupted. After that, no more random tracebacks.

### **Tamper vs handshake frames**

My tamper script was flipping *every* frame, which nuked the handshake since those 32-byte pubkey frames must stay intact. I changed it to only mess with “application” ciphertext frames, not the initial key exchange.

### **Ethics & professional practice**

I avoided logging plaintext anywhere, and all crypto happens client-side (relay sees ciphertext only). I cited the libs/papers I relied on.

I took AI help for debugging and for understanding the syntax and a few concepts.

## **9. Strengths, Weaknesses & Trade-offs**

### **Strengths**

- Small, pretty clear codebase but still shows CIA + manual auth properly.
- Each property is backed by concrete stuff (PCAP dump, MAC fail screenshot, relay logs), not just claims.
- Basic unit tests around the crypto bits and the ratchet so I knew it wasn't totally broken.

### **Weaknesses / Residual risks**

- No store-and-forward: if a peer is offline, those messages are just gone forever.
- Static keys aren't persisted, so smooth reconnects + a proper forward-secrecy replay demo was not possible.
- No GUI—Rich TUI could have been more interactable.
- Metadata (packet length/timing) still leaks, so an attacker can maybe infer patterns even if content stays secret.
- Only two clients can chat with each other at once adding extra clients would need fresh handshakes and new session management which I didn't implement.

### **Trade-offs**

I deliberately chose “keep it minimal and understandable” over “production features”. That means less complexity to reason about crypto, but also missing

stuff (e.g. message queues, nicer UX). which was in accordance with the 30 hour time frame.

## **10. Forward Secrecy Note (why the live replay demo is missing)**

I *did* wire in per-message key rotation. But proving it with a real replay attack would've needed the clients to stash their static keys on disk and then run a clean re-handshake after restart so I removed the code for that but I did write the code for that. That means extra state and complex code. So I just documented the limitation.

## **11. Personal Reflection – What Changed for Me**

I actually get how the ratchet works now, not just the documentation. I learnt a bunch of asyncio patterns the painful way (locks, handling half-dead sockets, etc.). Also realised “secure” isn't only math—making users compare fingerprints is super annoying but also essential. The hardest bit was balancing scope vs correctness: I had to say “no” to shiny stuff (better GUI, a polished FS replay demo) so the core stayed solid. Time wise I spent ~30 hours: reading/planning (~6h), crypto code + tests (~8h), relay/client (~7h), demos/screenshots (~5h), docs/report (~4h). And linking to CIA was also done. So I learnt how to code while keeping in mind the cybersecurity principles.

## **12. Conclusion & Future Work**

I ended up with a working E2EE chat that actually *shows* the main security props (CIA + auth) and backs them with concrete evidence. So the core goal was met, even if a few edges are rough.

If I keep going, I'd like to:

- **Persist static keys + clean re-handshakes:** so I can properly demo forward secrecy with a replay attack, not just claim it.

- **Encrypted store-and-forward:** to improve availability when peers are offline (right now they just miss stuff).
- **Better UX:** QR codes for fingerprint checks, maybe even a tiny GUI instead of the TUI—less pain for users.
- **Add multi-party support:** maintaining a one ratchet per one peer or switching to a group scheme like MLS

### 13. Note

Right now there is only 1 to 1 chat possible. The relay practically can forward to more people but my client file only tracks the one peer's public key and one double ratchet state. If another user joins I need to run an independent ratchet session for him and a new handshake UI or redesign to implement a group protocol.

There's more polish possible to this project but the time was the limitation.



## **References**

Perrin, T. (2018) *The Noise Protocol Framework*. Available at: <https://noiseprotocol.org/noise.pdf> (Accessed 19 July 2025).

Bernstein, D.J. and Lange, T. (n.d.) *NaCl: Networking and Cryptography Library*. Available at: <https://nacl.cr.yp.to/> (Accessed 16 July 2025).

Marlinspike, M. and Perrin, T. (2016) *The Double Ratchet Algorithm*. Available at: <https://signal.org/docs/specifications/doubleratchet/> (Accessed 17 July 2025).

PyNaCl Developers (2024) *PyNaCl Documentation (v1.5.0)*. Available at: <https://pynacl.readthedocs.io/en/latest/> (Accessed 19 July 2025).

Python Software Foundation (n.d.) *asyncio — Asynchronous I/O*. In: *Python 3.x Documentation*. Available at: <https://docs.python.org/3/library/asyncio.html> (Accessed 20 July 2025).

# Appendix

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	54	54921 → 9000 [PSH, ACK] Seq=1 Ack=1 Win=6380 Len=98 TSval=3334426775 TSecr=4110842407
2	0.000071	127.0.0.1	127.0.0.1	TCP	56	9000 → 54921 [ACK] Seq=1 Ack=99 Win=6379 Len=0 TSval=4110962283 TSecr=3334426775
3	0.000466	127.0.0.1	127.0.0.1	TCP	154	9000 → 54922 [PSH, ACK] Seq=1 Ack=1 Win=6380 Len=98 TSval=78910464 TSecr=549391323
4	0.000504	127.0.0.1	127.0.0.1	TCP	56	54922 → 9000 [ACK] Seq=1 Ack=99 Win=6379 Len=0 TSval=549511199 TSecr=78910464
5	19.901260	127.0.0.1	127.0.0.1	TCP	152	54922 → 9000 [PSH, ACK] Seq=1 Ack=99 Win=6379 Len=96 TSval=549531100 TSecr=78910464
6	19.901332	127.0.0.1	127.0.0.1	TCP	56	9000 → 54922 [ACK] Seq=99 Ack=97 Win=6379 Len=0 TSval=78930365 TSecr=549531100
7	19.901892	127.0.0.1	127.0.0.1	TCP	152	9000 → 54921 [PSH, ACK] Seq=1 Ack=99 Win=6379 Len=96 TSval=4110982184 TSecr=3334426775
8	19.901915	127.0.0.1	127.0.0.1	TCP	56	54921 → 9000 [ACK] Seq=99 Ack=97 Win=6379 Len=0 TSval=3334446676 TSecr=4110982184
9	23.483947	127.0.0.1	127.0.0.1	TCP	144	54922 → 9000 [PSH, ACK] Seq=97 Ack=99 Win=6379 Len=88 TSval=549534683 TSecr=78930365
10	23.484012	127.0.0.1	127.0.0.1	TCP	56	9000 → 54922 [ACK] Seq=99 Ack=185 Win=6378 Len=0 TSval=78933948 TSecr=549534683
11	23.484402	127.0.0.1	127.0.0.1	TCP	144	9000 → 54921 [PSH, ACK] Seq=97 Ack=99 Win=6379 Len=88 TSval=4110985767 TSecr=3334446676
12	23.484426	127.0.0.1	127.0.0.1	TCP	56	54921 → 9000 [ACK] Seq=99 Ack=185 Win=6378 Len=0 TSval=3334450259 TSecr=4110985767
13	30.623446	127.0.0.1	127.0.0.1	TCP	155	54921 → 9000 [PSH, ACK] Seq=99 Ack=185 Win=6378 Len=99 TSval=3334457398 TSecr=4110985767
14	30.623517	127.0.0.1	127.0.0.1	TCP	56	9000 → 54921 [ACK] Seq=185 Ack=198 Win=6378 Len=0 TSval=4110992906 TSecr=3334457398
15	30.623915	127.0.0.1	127.0.0.1	TCP	155	9000 → 54922 [PSH, ACK] Seq=99 Ack=185 Win=6378 Len=99 TSval=78941087 TSecr=549534683
16	30.623953	127.0.0.1	127.0.0.1	TCP	56	54922 → 9000 [ACK] Seq=185 Ack=198 Win=6378 Len=0 TSval=549541822 TSecr=78941087
17	42.469180	127.0.0.1	127.0.0.1	TCP	166	54922 → 9000 [PSH, ACK] Seq=185 Ack=198 Win=6378 Len=110 TSval=549553667 TSecr=78941087
18	42.469237	127.0.0.1	127.0.0.1	TCP	56	9000 → 54922 [ACK] Seq=198 Ack=295 Win=6377 Len=0 TSval=78952932 TSecr=549553667
19	42.469623	127.0.0.1	127.0.0.1	TCP	166	9000 → 54921 [PSH, ACK] Seq=185 Ack=198 Win=6378 Len=110 TSval=411004751 TSecr=3334457398
20	42.469644	127.0.0.1	127.0.0.1	TCP	56	54921 → 9000 [ACK] Seq=198 Ack=295 Win=6377 Len=0 TSval=3334469243 TSecr=411004751
21	47.283123	127.0.0.1	127.0.0.1	TCP	146	54921 → 9000 [PSH, ACK] Seq=198 Ack=295 Win=6377 Len=90 TSval=3334474057 TSecr=411004751
22	47.283191	127.0.0.1	127.0.0.1	TCP	56	9000 → 54921 [ACK] Seq=295 Ack=288 Win=6377 Len=0 TSval=411009565 TSecr=3334474057
23	47.283587	127.0.0.1	127.0.0.1	TCP	146	9000 → 54922 [PSH, ACK] Seq=198 Ack=295 Win=6377 Len=90 TSval=78957746 TSecr=549553667
24	47.283618	127.0.0.1	127.0.0.1	TCP	56	54922 → 9000 [ACK] Seq=295 Ack=288 Win=6377 Len=0 TSval=549558481 TSecr=78957746

> Frame 17: 166 bytes on wire (1328 bits), 166 bytes captured (1328 bits) on 0  
> Null/Loopback  
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1  
> Transmission Control Protocol, Src Port: 54922, Dst Port: 9000, Seq: 185, Ack: 198, Len: 110  
> Data (110 bytes)

0000 02 00 00 00 45 00 00 a2 00 00 40 00 40 06 00 00 ...E...@...  
0010 7f 00 00 01 7f 00 00 01 d6 8a 23 28 c9 bf b5 ab ...@[...  
0020 ff f6 e7 a8 80 18 ea fe 96 00 00 01 01 08 0a ...  
0030 20 c1 86 03 04 b4 8b 9f 00 00 00 6a 5a 8c c3 ba ...}Z...  
0040 c9 69 dd a3 0e 53 ab f6 ae b0 89 2b 46 26 d9 f8 ...i..S...+F6...  
0050 2f b0 0e ad 0c 89 06 d5 6e 2b 62 61 a8 da ff 3c .../.....n+ba...<  
0060 51 76 82 d6 60 6c 8a 1e 7e 17 e6 a3 c2 b2 62 6d ...Qv...l...+...-bm  
0070 81 9d 58 e9 ee 3d a4 5f a3 73 62 59 c0 48 87 66 ...X...-...sBy..H..f  
0080 22 35 82 8b b8 4d 4a 8e 11 bc 56 be 06 61 e1 22 ...5...MJ...V...a...  
0090 f7 a2 e5 a5 66 46 6c b6 eb 54 6d f4 64 ca 91 a9 ...FF...Tm..d...  
00a0 b3 99 90 b4 d4 f5 ...

Figure 1

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS
(venv) jatinsharma@Jatins-MacBook-Pro-2 6	(venv) jatinsharma@Jatins-MacBook-Pro-2 6	(venv) jatinsharma@Jatins-MacBook-Pro-2 6	(venv) jatinsharma@Jatins-MacBook-Pro-2 6	project/6441---Project---E2E-Chat/crypto.p
441---Project---E2E-Chat % python server.py --host 127.0.0.1 --port 9000	441---Project---E2E-Chat % python scripts /tamper_relay.py	441---Project---E2E-Chat % python client.py --host 127.0.0.1 --port 9001 --name Alice	441---Project---E2E-Chat % python scripts /tamper_relay.py	y", line 151, in decrypt
14:59:44   INFO   Server listening on 127.0.0.1:9000	2025-07-25 14:59:49,213   Tamper-relay listening on ('127.0.0.1', 9001)	Connecting to 127.0.0.1:9001 ...	Connecting to 127.0.0.1:9001 ...	return box.decrypt(nonce + ct)
14:59:58   INFO   C01 connected from ('127.0.0.1', 51080)	2025-07-25 14:59:58,543   : New tamper session	Connected.	Connected.	File "/Users/jatinsharma/Desktop/6441 Project/6441---Project---E2E-Chat/venv/lib/python3.10/site-packages/nacl/bindings/crypto_secretbox.py", line 149, in decrypt
14:59:58   INFO   C01 sent 32 B	2025-07-25 15:00:07,192   : Bit-flipped first byte on application frame	Identity	Identity	plaintext = nacl.bindings.crypto_secretbox_open
14:59:58   INFO   stored pubkey for C01		Your public key fingerprint: 32c324495de7141f212f8b5435904087da30c299b0e623e81cb29da6ce76b926	Your public key fingerprint: 32c324495de7141f212f8b5435904087da30c299b0e623e81cb29da6ce76b926	File "/Users/jatinsharma/Desktop/6441 Project/6441---Project---E2E-Chat/venv/lib/python3.10/site-packages/nacl/bindings/crypto_secretbox.py", line 79, in crypto_secretbox_open
15:00:07   INFO   C02 connected from ('127.0.0.1', 51082)		Key Exchange	Key Exchange	ensure()
15:00:07   INFO   sent 1 buffered pubkey(s) to C02		Peer public key received: f2d58aa07749d5195ec753d368	Peer public key received: f2d58aa07749d5195ec753d368	File "/Users/jatinsharma/Desktop/6441 Project/6441---Project---E2E-Chat/venv/lib/python3.10/site-packages/nacl/exceptions.py", line 88, in ensure
15:00:07   INFO   C02 sent 32 B		You can now start chatting!	You can now start chatting!	raise raising(kargs)
15:00:07   INFO   stored pubkey for C02		hello	hello	nacl.exceptions.CryptoError: Decryption failed. Ciphertext failed verification
15:00:07   INFO   forwarded to C01 (32 B)		hello	hello	(venv) jatinsharma@Jatins-MacBook-Pro-2 6
15:00:11   INFO   C01 sent 77 B				441---Project---E2E-Chat %
15:00:11   INFO   forwarded to C02 (77 B)				

Figure 2

```
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % python server.py --host 127.
0.0.1 --port 9000
01:58:47 | INFO | Relay listening on 127.0.0.1:9
000
01:58:57 | INFO | ✓ C01 connected from ('127.0.
0.1', 63815)
01:58:57 | INFO | ↗from C01 | 32 bytes
01:58:57 | INFO | • buffered pubkey for C01
[]

(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % python client.py --host 127.
0.0.1 --port 9000 --name Alice
✂ Connecting to 127.0.0.1:9000 ...
✓ Connected.
hey
Waiting for peer...
how are you
Waiting for peer...
what are you doing
Waiting for peer...
[]

jatinsharma@Jatins-MacBook-Pro-2 6441---Project-
---E2E-Chat % source venv/bin/activate
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % []
```

Figure 3a

```
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % python server.py --host 127.
0.0.1 --port 9000
01:58:47 | INFO | Relay listening on 127.0.0.1:9
000
01:58:57 | INFO | ✓ C01 connected from ('127.0.
0.1', 63815)
01:58:57 | INFO | ↗from C01 | 32 bytes
01:58:57 | INFO | • buffered pubkey for C01
02:00:16 | INFO | ✓ C02 connected from ('127.0.
0.1', 63818)
02:00:16 | INFO | • forwarded 1 buffered pub
key(s) to C02
02:00:16 | INFO | ↗from C02 | 32 bytes
02:00:16 | INFO | • buffered pubkey for C02
02:00:16 | INFO | → to C01 | 32 bytes
02:00:32 | INFO | ↗from C02 | 77 bytes
02:00:32 | INFO | → to C01 | 77 bytes
02:00:35 | INFO | ↗from C01 | 74 bytes
02:00:35 | INFO | → to C02 | 74 bytes
02:00:39 | INFO | ↗from C02 | 83 bytes
02:00:39 | INFO | → to C01 | 83 bytes
[]

(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % python client.py --host 127.
0.0.1 --port 9000 --name Alice
✂ Connecting to 127.0.0.1:9000 ...
✓ Connected.
hey
Waiting for peer...
how are you
Waiting for peer...
what are you doing
Waiting for peer...
[]

Your public key fingerprint:
195d086c4d2c741dfbf9ebc0ba099e5197ba74efd96a
c0d9b0114dd383cfd538

Key Exchange
Peer public key received:
f22795038e736acde7816de8e728b5060554b99f3183
2cc9d30fa739790f4739

You can now start chatting!
Alice- hello
hi
You- hi
Alice- how are you
[]

jatinsharma@Jatins-MacBook-Pro-2 6441---Project-
---E2E-Chat % source venv/bin/activate
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---P
project---E2E-Chat % python client.py --host 127.
0.0.1 --port 9000 --name Bob
✂ Connecting to 127.0.0.1:9000 ...
✓ Connected.

Identity
Your public key fingerprint:
f22795038e736acde7816de8e728b5060554b99f3183
2cc9d30fa739790f4739

Key Exchange
Peer public key received:
195d086c4d2c741dfbf9ebc0ba099e5197ba74efd96a
c0d9b0114dd383cfd538

You can now start chatting!
hello
You- hello
Bob- hi
how are you
You- how are you
[]
```

Figure 3b

```
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python server.py --host 127.0.0.1 --port 9000
01:58:47 | INFO | Relay listening on 127.0.0.1:9000
01:58:57 | INFO | ✓ C01 connected from ('127.0.0.1', 63815)
01:58:57 | INFO | ↗ from C01 | 32 bytes
01:58:57 | INFO | • buffered pubkey for C01
02:00:16 | INFO | ✓ C02 connected from ('127.0.0.1', 63818)
02:00:16 | INFO | • forwarded 1 buffered pubkey(s) to C02
02:00:16 | INFO | ↗ from C02 | 32 bytes
02:00:16 | INFO | • buffered pubkey for C02
02:00:16 | INFO | → to C01 | 32 bytes
[]

(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python client.py --host 127.0.0.1 --port 9000 --name Alice
Connecting to 127.0.0.1:9000 ...
✓ Connected.
hey
Waiting for peer...
how are you
Waiting for peer...
what are you doing
Waiting for peer...

Identity
Your public key fingerprint:
195d086c4d2c741dfbf9ebc0ba099e5197ba74efd96a
c0d9b0114dd383cfd538

Key Exchange
Peer public key received:
f22795038e736acde7816de8e728b5060554b99f3183
2cc9d30fa739790f4739

You can now start chatting!
[]

jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % source venv/bin/activate
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python client.py --host 127.0.0.1 --port 9000 --name Bob
Connecting to 127.0.0.1:9000 ...
✓ Connected.

Identity
Your public key fingerprint:
f22795038e736acde7816de8e728b5060554b99f3183
2cc9d30fa739790f4739

Key Exchange
Peer public key received:
195d086c4d2c741dfbf9ebc0ba099e5197ba74efd96a
c0d9b0114dd383cfd538

You can now start chatting!
[]
```

Figure 4

```
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python server.py --host 127.0.0.1 --port 9000
15:03:30 | INFO | Server listening on 127.0.0.1:9000
15:03:52 | INFO | ✓ C01 connected from ('127.0.0.1', 51124)
15:03:52 | INFO | ↑ C01 sent 32 B
15:03:52 | INFO | • stored pubkey for C01
15:03:58 | INFO | ✓ C02 connected from ('127.0.0.1', 51125)
15:03:58 | INFO | • sent 1 buffered pubkey(s) to C02
15:03:58 | INFO | ↑ C02 sent 32 B
15:03:58 | INFO | • stored pubkey for C02
15:03:58 | INFO | → forwarded to C01 (32 B)
15:04:01 | INFO | ↑ C01 sent 75 B
15:04:01 | INFO | → forwarded to C02 (75 B)
15:04:06 | INFO | ↑ C02 sent 90 B
15:04:06 | INFO | → forwarded to C01 (90 B)
[]

(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python client.py --host 127.0.0.1 --port 9000 --name Alice
Connecting to 127.0.0.1:9000 ...
✓ Connected.

Identity
Your public key fingerprint:
023013a02b9aaf3bf732c89546c737845e16ac1a8e2dbbbe8f2
af47775c5f59

Key Exchange
Peer public key received:
0728e4bb5585897e97f0eda06d23876f8400c9e3d33e34206b8c
8fd812d9f174

You can now start chatting!
hey
You-> hey
Alice-> how are you doing?
[]

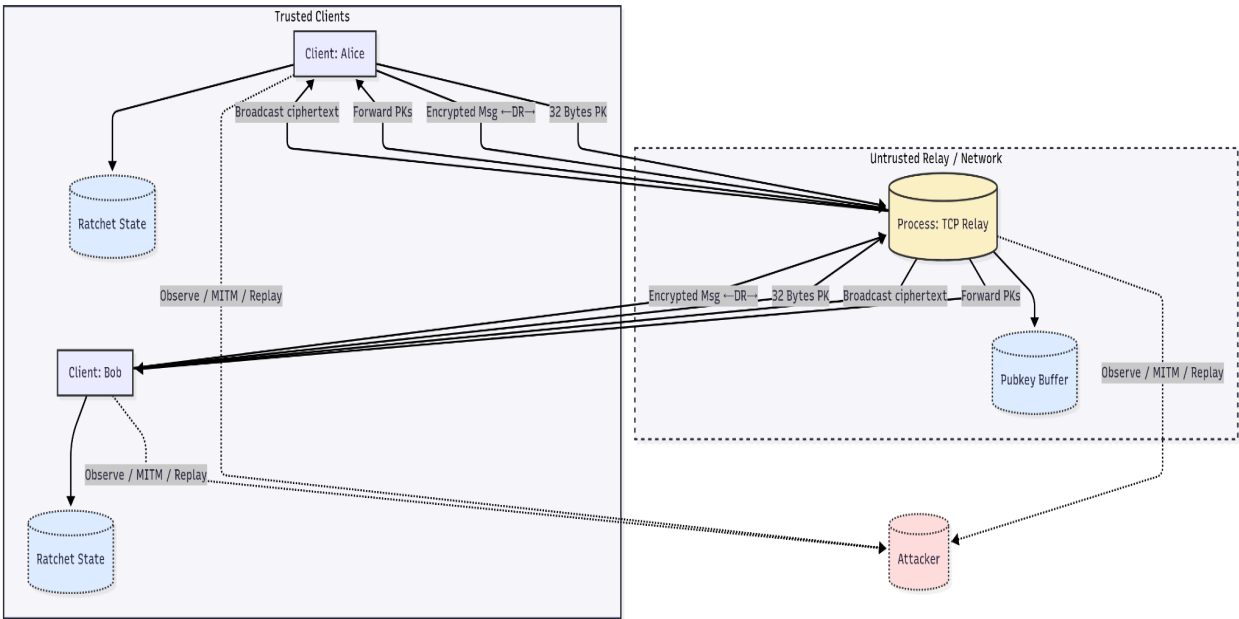
(venv) jatinsharma@Jatins-MacBook-Pro-2 6441---Project--E2E-Chat % python client.py --host 127.0.0.1 --port 9000 --name Bob
Connecting to 127.0.0.1:9000 ...
✓ Connected.

Identity
Your public key fingerprint:
0728e4bb5585897e97f0eda06d23876f8400c9e3d33e34206b8c
8fd812d9f174

Key Exchange
Peer public key received:
023013a02b9aaf3bf732c89546c737845e16ac1a8e2dbbbe8f2
af47775c5f59

You can now start chatting!
Bob-> hey
how are you doing?
You-> how are you doing?
[]
```

Figure 5 (working model of the E2EE chat)



This is threat\_model.png for the chat model

## Risk Table

(Quick map of what could go wrong, what I actually did about it, and what's still left as risk.)

Threat (what could go wrong)	Mitigation I put in	Residual / still a problem
MITM during key-exchange	X25519 DH + manual fingerprint compare.	If we skip or mis-read the hex, a MITM could sneak in. Human error, basically.
Ciphertext tampering (bit-flip etc.)	XChaCha20-Poly1305 (NaCl SecretBox) MAC on every msg	Attacker can still spam bad frames to trigger MAC errors (DoS-ish), no plaintext leaks tho.
Replay of old ciphertext	I implemented double Ratchet	Cannot write the full working usage as it required significant refactoring
Stolen device / state leakage	Ratchet state only in RAM, keys rotate per message	If someone grabs the device while it's active, they may see future msgs until next ratchet step. Past msgs safe.
Offline peer ⇒ message loss (availability)	Intentionally no store-and-forward in relay	Anything sent while peer is offline is gone forever. I accept that as a trade-off.
flooding	Super small asyncio server	No rate limit or auth, so a bored attacker could just flood messages/frames.
Only 1:1 session supported right now	Kept it simple: one ratchet per peer to focus on core ideas	Adding more users needs new handshakes & per-peer state (or MLS). Not done yet.

Risk\_table for my chat model

