

LAN Messenger

Aayush Bajaj

July 10, 2026

Contents

1 architecture	1
1.1 the socket lifecycle	2
2 authentication and logout	3
3 message framing, or: tcp is not a message pipe	3
4 commands and messaging	4
5 the udp side channel	5
6 concurrency audit	5
7 artefacts	6
8 see also	6

“tessenger” is my unsw networks assignment: a multi-client chat server in raw c — `server.c` (583 lines) and `client.c` (253), no libraries beyond pthreads and the bsd socket api (Kernighan, Brian W. and Ritchie, Dennis M., 1988). it does authentication with timed logout, private and group messaging, presence, audit logs, and a udp side channel for peer-to-peer file transfer. this page walks the architecture, weaves in the socket theory the assignment was designed to teach, and then does something the assignment never required: an honest concurrency audit of my own submission, with the bugs demonstrated live.

everything quoted below — compiles, session transcripts, log files, even the crash — is a real run of this code on this machine, december-2023 sources untouched except where noted.

1 architecture

one process, $1 + n$ threads. the main thread owns the listening socket and does nothing but `accept()`; every accepted connection gets its own detached `client_handler` thread that blocks on `recv()` for that one client. shared state — the client table and group table — lives in globals guarded (in theory) by two mutexes:

```
1 // from messenger/server.c (trimmed)
2 #define BUF_LEN 1024
3 #define MAX_CLIENTS 50
4 #define MAX_GROUPS 20
5 #define LOCKOUT_TIME 10 // lockout time in seconds
6
7 typedef struct {
8     int sock;
9     char username[256];
10    char udp[256]; // client's udp port, as a string
```

```

11  char ip[INET_ADDRSTRLEN];
12  int failed_attempts;
13  time_t last_failed_time;
14  int active;
15  } client_t;
16
17  typedef struct {
18      int id;
19      char name[256];
20      client_t *members[MAX_CLIENTS]; // pointers into the global client table
21      int member_count;
22  } group_t;
23
24  client_t clients[MAX_CLIENTS]; int client_count = 0;
25  pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
26  group_t groups[MAX_GROUPS]; int group_count = 0;
27  pthread_mutex_t groups_mutex = PTHREAD_MUTEX_INITIALIZER;

```

the **thread-per-client** model is the pedagogically honest one: each handler does blocking io on exactly one socket, so there is no event loop, no `select()`~/~`epoll` bookkeeping, and the cost is one stack per connection — fine for `MAX_CLIENTS 50` on a lan, ruinous at internet scale (this trade-off is the `c10k` story) (Tanenbaum, Andrew S., 2008).

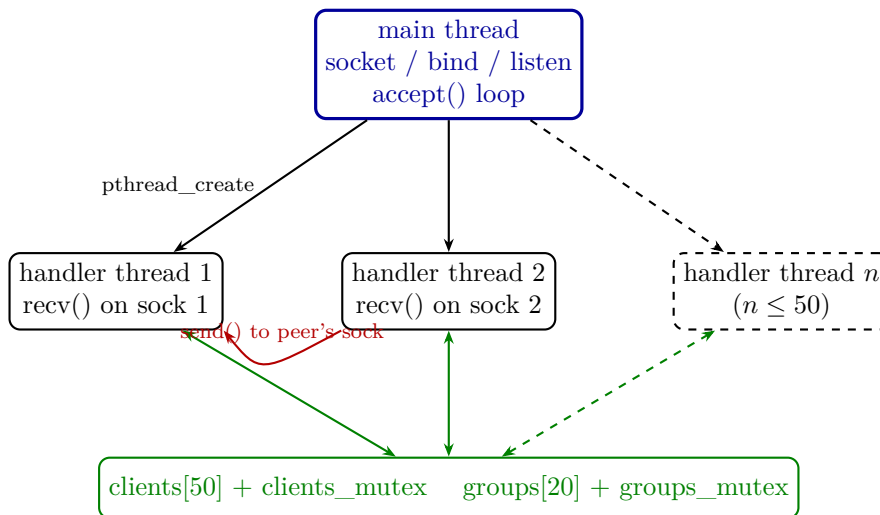


Figure 1: tessenger’s thread model: the main thread blocks in `accept()`, spawning one handler thread per client; handlers block in `recv()` on their own socket and reach into mutex-guarded global tables to route messages to other clients’ sockets.

1.1 the socket lifecycle

the server’s `main()` is the canonical tcp passive-open sequence (Kurose, James F. and Ross, Keith W., 2020):

1. `socket(AF_INET, SOCK_STREAM, 0)` — ask the kernel for a tcp endpoint.
2. `bind()` to `INADDR_ANY` and the chosen port — claim the local address.
3. `listen(fd, 5)` — flip the socket passive; the 5 is the **backlog**, the queue of completed-but-unaccepted connections.
4. `accept()` in a loop — each call blocks until the kernel hands over a fresh *connected* socket, distinct from the listening one.

one omission worth knowing about: the code never sets `SO_REUSEADDR`. kill the server and restart it within a couple of minutes and `bind()` fails with “address already in use”, because the old connections sit in `TIME_WAIT` — the one-line `setsockopt()` every server tutorial adds is genuinely load-bearing during development. the client side is the mirror ritual: `socket()`, `connect()`, then a thread of its own for the udp listener (below).

2 authentication and logout

usage is `./server <port> <max_failed_attempts>` with the attempt limit validated to 1.5. credentials live in a plaintext `credentials.txt` (the assignment’s spec, not my security posture — the accounts are star wars characters). on connect, the client sends “username password”, the server linearly scans the file, and per-client failure counters implement a **timed logout**: after `max_attempts` consecutive failures inside the window, the account blocks for `LOCKOUT_TIME` (10 s), enforced by comparing `time(NULL)` against `last_failed_time` with `difftime()`.

that is the design. the implementation of the retry path has two real bugs, and they compound:

- the retry loop never calls `recv()` — `while ((auth_result = authenticate(client_message)) != 1)` re-tests the *same* buffer, so one wrong password means the loop spins at full speed re-failing and spamming `RETRY` down the socket.
- `client_blocked()` returns `-1` for “locked out”, but the caller tests `== 1` — the `BLOCKED` branch (and its clean `close()`) is unreachable. worse, that `-1` path returns *without releasing* `clients_mutex`.

demonstrated, not asserted — a probe client that sends a wrong password and disconnects takes the whole server down, because the still-spinning retry loop `send()~s` into the closed socket and macos delivers `~SIGPIPE` (default action: terminate):

```
$ python3 probe.py    # wrong password, then close()
delay=0.0: TIMEOUT waiting for auth reply (sends coalesced)
$ python3 probe.py    # second run, same server
ConnectionRefusedError: [Errno 61] Connection refused    # server is dead
```

(robust servers set `signal(SIGPIPE, SIG_IGN)` or pass `MSG_NOSIGNAL` and handle the `EPIPE` error return instead.)

3 message framing, or: tcp is not a message pipe

the probe output above smuggled in a second lesson. the client authenticates with two back-to-back `send()` calls — “username password”, then its udp port — and the server pairs them with two `recv()` calls. but **tcp is a byte stream**: `send()` boundaries are not preserved, and on the loopback interface the two writes routinely coalesce into one segment (Kurose, James F. and Ross, Keith W., 2020). when that happens the server’s first `recv()` reads “Yoda wise@!manbaby4002”, authentication fails, and we are back in the death-loop above. pacing the two sends 300 ms apart “fixes” it:

```
delay=0.0: TIMEOUT waiting for auth reply (sends coalesced)
delay=0.3: reply: Welcome to Tessenger!
```

in 2023 this worked in testing because nagle’s algorithm on the linux lab machines happened to hold the second small write until the first was acknowledged — a scheduling accident doing the job of a protocol. the correct fix is **framing**: delimit messages (newlines, which the command loop already half-uses) or length-prefix them, and buffer partial reads until a full frame arrives. every real wire protocol — http’s content-length, the referee protocol in **ultimate tic tac toe** with its `.\n` terminators — is answering exactly this question.

4 commands and messaging

after **WELCOME**, the handler loop reads a line, classifies it with a `strncmp` ladder folded into an enum (`get_command_type()` — “didn’t want 300 strcmps”, says the comment), and switches:

command	behaviour (as implemented)
<code>/msgto user text</code>	look up recipient, <code>send()</code> text to their socket, log to <code>messagelog.txt</code>
<code>/activeuser</code>	list logged-in usernames
<code>/creategroup name u1 u2..</code>	create group (max 20), enrol listed users
<code>/joingroup name</code>	add self to group
<code>/groupmsg name text</code>	<code>send()</code> text to every member’s socket
<code>/listgroups</code>	list group names
<code>/groupmembers name</code>	list a group’s members
<code>/p2pvideo user file</code>	reply with recipient’s ip + udp port (transfer happens client-side)
<code>/logout</code>	reply <code>exit</code> , remove from client table

presence and messages are audited to `userlog.txt` and `messagelog.txt` with `strftime`-formatted timestamps. a real two-client session against the compiled server (driven by a scripted client that paces its auth sends):

```
$ gcc -Wall -pthread -o server server.c # only unused-variable warnings
$ ./server 5761 3 &
$ python3 session.py
[Chewy] Welcome to Tessenger!
[Yoda] Welcome to Tessenger!
[Yoda] /activeuser -> 'Chewy\nYoda\n'
[Yoda] /msgto -> Message sent
[Chewy] received: hello there general kenobi
[Yoda] /p2pvideo -> UDP: 127.0.0.1 4001 example1.mp4
[Yoda] /logout -> exit

$ cat userlog.txt
2; 09 Jul 2026 21:02:02; Chewy; 127.0.0.1; 4001
4; 09 Jul 2026 21:02:02; Yoda; 127.0.0.1; 4002
$ cat messagelog.txt
1; 09 Jul 2026 21:02:03; Chewy; hello there general kenobi
```

it works — private messaging, presence, the p2p handshake, the logs. (the `userlog` sequence numbers jumping 2, 4 instead of 1, 2 are a bug: `active_clients` is incremented in both `add_client()` and `userlog()`, so every login counts twice.)

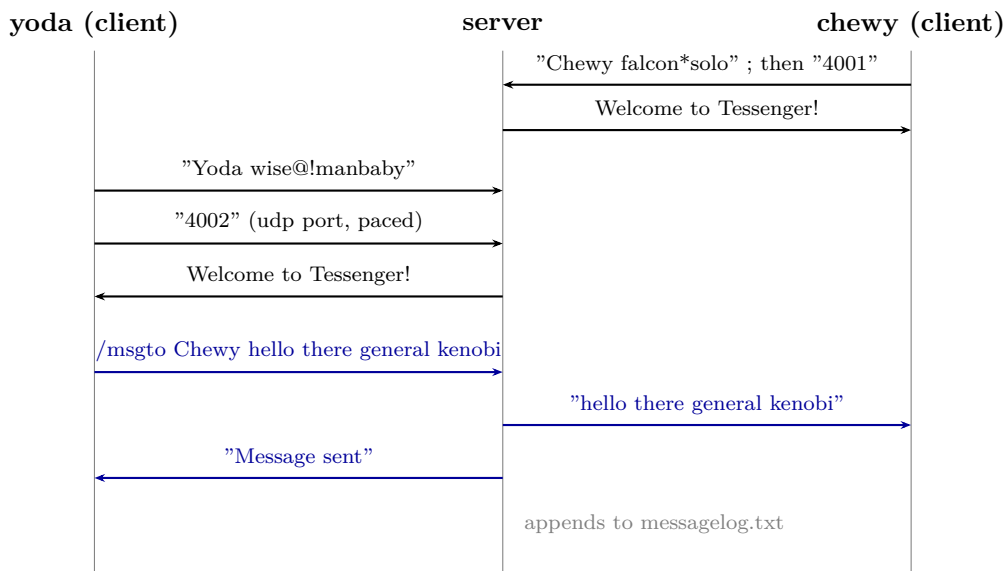


Figure 2: sequence diagram of the smoke test: paced authentication, then a private message. note /msgto produces two sends from the server — the delivery to chewy and the acknowledgement to yoda.

5 the udp side channel

/p2pvideo is the assignment’s excuse to make you touch both transport protocols. the tcp server never carries the file — it only brokers the rendezvous: it looks up the recipient and replies **UDP: <ip> <port> <filename>**. the *sending client* then opens a **SOCK_DGRAM** socket and fires the file at that address in 1024-byte **sendto()** chunks (**my_sendfile()**); every client has been running a dedicated **udp listener thread** since startup (its port is the third argv, registered with the server at login), which **recvfrom()~s chunks and appends them to ~output.dat**.

honest appraisal of the datagram half: udp gives no delivery, ordering, or duplication guarantees, and this code adds no sequence numbers, acks, or end-of-file marker — on a quiet lan loopback it works, across any real network the file arrives probabilistically (Kurose, James F. and Ross, Keith W., 2020). the listener also hardcodes **output.dat**, so a second transfer tramples the first, and it writes forever (no termination signal). a real design would either add tftp-style stop-and-wait over the datagrams or concede and open a second tcp connection. as a demonstration that the client can juggle a blocking tcp loop and a udp receiver concurrently, though, it does its job.

6 concurrency audit

the assignment asks “does it work?”; a systems programmer should ask “is it correct under contention?” reading my own submission two years later, with fresh eyes (Tanenbaum, Andrew S., 2008):

- **lock discipline is aspirational.** **get_client_by_sock()** *unlocks* **clients_mutex** on the found-path without ever locking it (undefined behaviour on a fast mutex); **client_blocked()**’s lockout path and **remove_client_from_group()**’s success path both return with mutexes still held — instant deadlock the next time anyone touches the table; **groups_string()** takes no lock at all.
- **pointers escape their critical sections.** **get_client_by_username()** un-

locks, then returns a pointer into `clients[]` — but `remove_client()` compacts the array by copying the last element over the removed slot, so a concurrently-held pointer can silently start referring to a *different user*. the group table's `client_t *members[]` pointers have the same problem permanently.

- **unserialised counters and logs.** `active_clients` is double-incremented (seen in the real userlog above), `message_count` is unguarded, and two threads appending to the same log file interleave at the mercy of stdio buffering.
- **uninitialised heap as string.** `clients_string()` does `sprintf(buf + strlen(buf), ...)` on a fresh `malloc` — correct only when the allocator hands back zeroed pages, which macos happened to do in the session above. `members_string()` and `groups_string()` also iterate `client_count` instead of their own counts.
- **what it gets right:** the mutexes exist and mostly bracket the mutations; the per-thread blocking-io design means *message routing* itself has no torn reads (a `send()` on a fd is atomic enough at these sizes); and `thread_pack` is malloc'd per connection and freed by the handler — no stack-lifetime bug, which is the classic thread-spawn mistake.

the general lesson: thread-per-client makes the *io* trivially correct and pushes all the danger into shared state. the fashionable alternatives — a single-threaded event loop, or message-passing to one owner thread — are popular precisely because they make the audit above unnecessary.

7 artefacts

the repo also carries the assignment `report.pdf`, the original `makefile` (`LDFLAGS = -pthread`), the star-wars `credentials.txt`, and `userlog.txt`~/~/`messagelog.txt` from the december 2023 marking runs — the transcript above appends to the same format. the only change made for this page was compiling with `-Wall` (three unused-variable warnings, zero errors) and driving it with scripted clients; the bugs were left exactly as submitted, because they are the most instructive part.

8 see also

- [messenger web app](#) — the same product idea rebuilt on websockets instead of raw sockets
- [ultimate tic tac toe](#) — another line-protocol-over-tcp client, with the framing done properly by the course's referee
- [bytelocker in c](#) — more systems c
- [computer science projects](#) — parent section

references

Kernighan, Brian W. and Ritchie, Dennis M. (1988). *The C Programming Language*, Prentice Hall.

Kurose, James F. and Ross, Keith W. (2020). *Computer Networking: A Top-Down Approach*, Pearson.

Tanenbaum, Andrew S. (2008). *Modern Operating Systems*, Pearson.