**Overview**

You will implement a 3-node distributed chat-room application. One node acts as **file server** (stores the shared chat file). The other two nodes are **user nodes** that run a distributed application and a **distributed mutual exclusion (DME)** middleware based on **Lamport’s mutual exclusion algorithm**. Users can view (read) the shared file at any time. To post (write), a user must obtain exclusive write access via the DME algorithm. Each post also records the client-side timestamp and user id.

We keep DME and application code separate (as required). The DME code is in dme.py. The user app that uses it is client.py. The server is server.py.

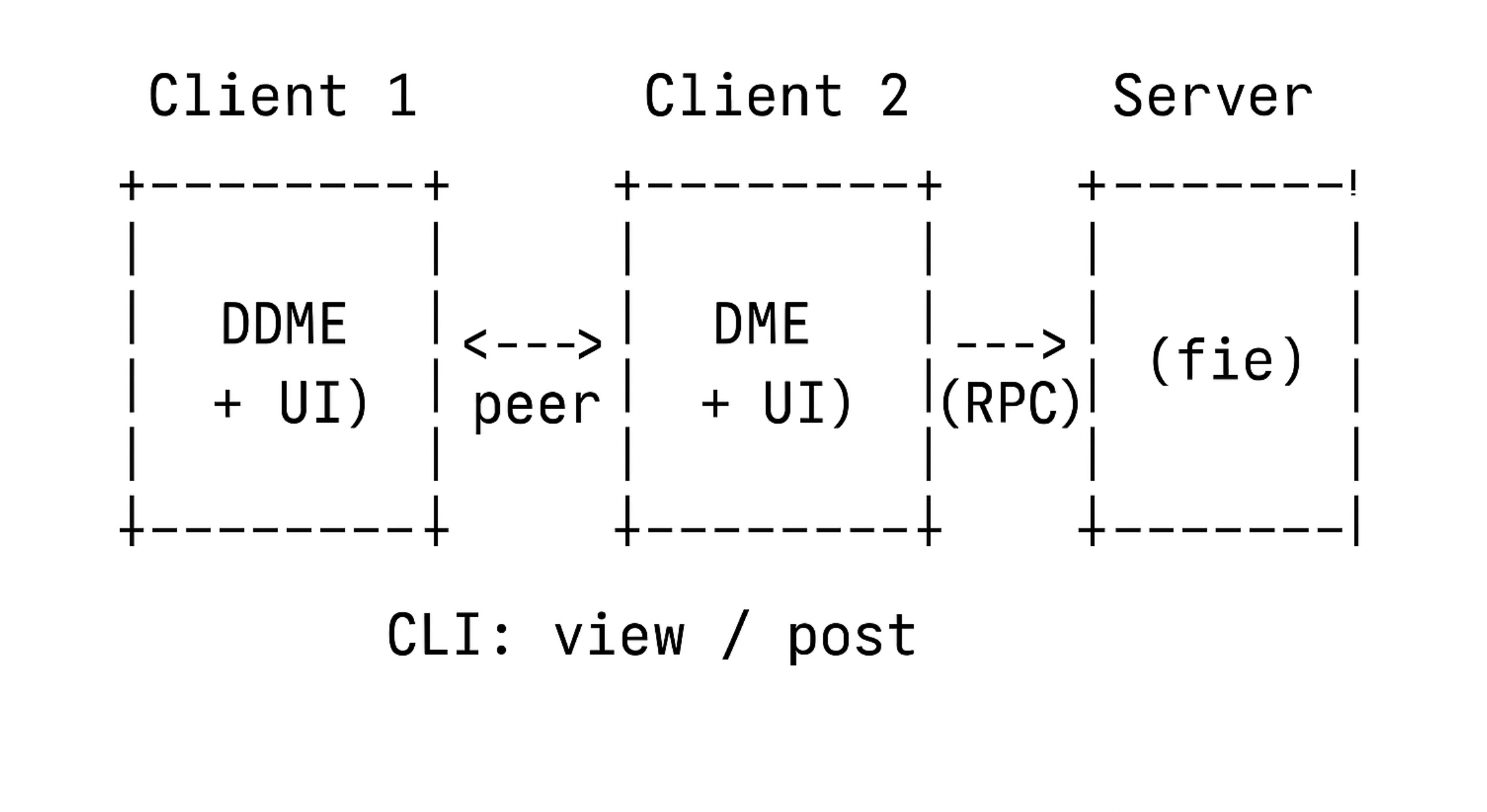
**Architecture**

Three processes/nodes:

* **Server (file server)**: listens for VIEW and POST RPCs from users. Stores chat.txt.
* **Client A (user node 1)**: runs dme.py middleware and client.py user app.
* **Client B (user node 2)**: same as Client A.

All communication between clients (for DME) is peer-to-peer. Communication to server for reading/writing is direct client → server.

**ASCII architecture:**



Sequence for post (Lamport DME summary):

1. Client generates a Lamport timestamp and sends REQUEST(ts, id) to all other clients.
2. Each client replies with REPLY when it is safe (based on lamport ordering).
3. Requester waits for REPLY from all (including itself) — then it enters critical section.
4. Client sends POST to the server (append with client timestamp and id).
5. Client sends RELEASE to all peers; other nodes update their request queues.

**Files & code**

You can run these programs in three terminal windows (or three separate VMs/containers), changing ports as needed. The implementation keeps things simple and robust for demonstration.

Save the three code files below: server.py, dme.py, client.py.

**server.py**

*# server.py*

*# Simple file server for the shared chat file.*

*# Usage: python server.py <port>*

*# Example: python server.py 9000*

*import socket*

*import threading*

*import json*

*import sys*

*import os*

*from datetime import datetime*

*HOST = '0.0.0.0'*

*CHAT\_FILE = "chat.txt"*

*LOCK = threading.Lock()*

*def ensure\_file():*

*if not os.path.exists(CHAT\_FILE):*

*with open(CHAT\_FILE, "w") as f:*

*f.write("")*

*def handle\_connection(conn, addr):*

*try:*

*data = conn.recv(65536).decode()*

*if not data:*

*return*

*req = json.loads(data)*

*action = req.get("action")*

*if action == "VIEW":*

*with LOCK:*

*with open(CHAT\_FILE, "r") as f:*

*content = f.read()*

*resp = {"status": "OK", "content": content}*

*conn.sendall(json.dumps(resp).encode())*

*elif action == "POST":*

*# expected fields: user\_id, text, client\_ts (string)*

*user = req.get("user\_id")*

*text = req.get("text")*

*client\_ts = req.get("client\_ts")*

*# record server receipt time too (for logs)*

*server\_ts = datetime.now().strftime("%d %b %Y %I:%M:%S%p")*

*line = f"{client\_ts} {user}: {text}\n"*

*with LOCK:*

*with open(CHAT\_FILE, "a") as f:*

*f.write(line)*

*resp = {"status": "OK", "server\_ts": server\_ts}*

*conn.sendall(json.dumps(resp).encode())*

*else:*

*conn.sendall(json.dumps({"status": "ERR", "message": "Unknown action"}).encode())*

*except Exception as e:*

*try:*

*conn.sendall(json.dumps({"status":"ERR","message":str(e)}).encode())*

*except:*

*pass*

*finally:*

*conn.close()*

*def start\_server(port):*

*ensure\_file()*

*s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*s.bind((HOST, port))*

*s.listen(5)*

*print(f"[SERVER] Listening on {HOST}:{port}, chat file: {CHAT\_FILE}")*

*try:*

*while True:*

*conn, addr = s.accept()*

*t = threading.Thread(target=handle\_connection, args=(conn, addr), daemon=True)*

*t.start()*

*except KeyboardInterrupt:*

*print("[SERVER] Shutting down")*

*finally:*

*s.close()*

*if \_\_name\_\_ == "\_\_main\_\_":*

*if len(sys.argv) != 2:*

*print("Usage: python server.py <port>")*

*sys.exit(1)*

*port = int(sys.argv[1])*

*start\_server(port)*

**dme.py (Lamport mutual exclusion middleware)**

*# dme.py*

*# Lamport Distributed Mutual Exclusion middleware*

*# Each client using this middleware must supply:*

*# - my\_id: unique identifier string*

*# - my\_port: port where this node listens for DME messages*

*# - peers: list of ("host", port, id) for other clients*

*#*

*# Provides:*

*# - request\_critical\_section() -> blocks until allowed*

*# - release\_critical\_section()*

*import socket*

*import threading*

*import json*

*import time*

*from queue import PriorityQueue*

*class LamportDME:*

*def \_\_init\_\_(self, my\_id, my\_port, peers):*

*self.my\_id = my\_id*

*self.my\_port = my\_port*

*self.peers = peers # list of dicts: {"host":..., "port":..., "id":...}*

*self.clock = 0*

*self.lock = threading.Lock()*

*# Replies management*

*self.replies\_needed = set()*

*self.replies\_event = threading.Event()*

*# Request queue: items are (timestamp, id)*

*self.request\_queue = PriorityQueue()*

*# Start listener for incoming DME messages*

*self.running = True*

*t = threading.Thread(target=self.\_listener, daemon=True)*

*t.start()*

*def \_increment\_clock(self, other\_ts=None):*

*with self.lock:*

*if other\_ts is None:*

*self.clock += 1*

*else:*

*self.clock = max(self.clock, other\_ts) + 1*

*return self.clock*

*def \_listener(self):*

*s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*s.bind(('0.0.0.0', self.my\_port))*

*s.listen(5)*

*while self.running:*

*try:*

*conn, addr = s.accept()*

*data = conn.recv(65536).decode()*

*if not data:*

*conn.close()*

*continue*

*msg = json.loads(data)*

*typ = msg.get("type")*

*if typ == "REQUEST":*

*ts = msg["timestamp"]*

*nid = msg["node\_id"]*

*# update clock*

*self.\_increment\_clock(ts)*

*# enqueue request*

*self.request\_queue.put((ts, nid))*

*# send REPLY*

*reply = {"type":"REPLY", "timestamp": self.\_increment\_clock(), "node\_id": self.my\_id}*

*conn.sendall(json.dumps(reply).encode())*

*elif typ == "RELEASE":*

*ts = msg["timestamp"]*

*nid = msg["node\_id"]*

*# update clock*

*self.\_increment\_clock(ts)*

*# remove from queue: reconstruct queue without the released request*

*self.\_remove\_request(nid, ts)*

*conn.sendall(json.dumps({"type":"ACK"}).encode())*

*elif typ == "REPLY":*

*ts = msg["timestamp"]*

*nid = msg["node\_id"]*

*self.\_increment\_clock(ts)*

*# mark reply received*

*with self.lock:*

*if nid in self.replies\_needed:*

*self.replies\_needed.remove(nid)*

*if not self.replies\_needed:*

*self.replies\_event.set()*

*conn.sendall(json.dumps({"type":"ACK"}).encode())*

*else:*

*conn.sendall(json.dumps({"status":"ERR", "message":"unknown type"}).encode())*

*conn.close()*

*except Exception as e:*

*# continue listening*

*continue*

*s.close()*

*def \_remove\_request(self, nid, ts):*

*# PriorityQueue doesn't allow direct removal — rebuild*

*items = []*

*while not self.request\_queue.empty():*

*items.append(self.request\_queue.get())*

*# remove matching*

*items = [it for it in items if not (it[1] == nid and it[0] == ts)]*

*# requeue*

*for it in items:*

*self.request\_queue.put(it)*

*def \_send\_to\_peer(self, peer, msg, expect\_response=False, timeout=5):*

*try:*

*s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*s.settimeout(timeout)*

*s.connect((peer["host"], peer["port"]))*

*s.sendall(json.dumps(msg).encode())*

*if expect\_response:*

*data = s.recv(65536).decode()*

*if data:*

*return json.loads(data)*

*s.close()*

*except Exception as e:*

*# network error or peer down; if peer is down then we treat as non-responsive*

*return None*

*return None*

*def request\_critical\_section(self):*

*# increment local clock and create request timestamp*

*ts = self.\_increment\_clock()*

*# enqueue own request*

*self.request\_queue.put((ts, self.my\_id))*

*# prepare set of peers required replies*

*self.replies\_needed = set(p["id"] for p in self.peers)*

*# send REQUEST to all peers*

*req\_msg = {"type":"REQUEST", "timestamp": ts, "node\_id": self.my\_id}*

*for p in self.peers:*

*# best-effort send; REPLY may come later*

*self.\_send\_to\_peer(p, req\_msg, expect\_response=True)*

*# Wait until all replies received and own request is at the head of queue*

*# Wait for replies\_event*

*# But also check queue head*

*# Use a loop with timeout to avoid deadlock if a peer is down:*

*start = time.time()*

*while True:*

*# first: wait for replies (but don't block forever)*

*if not self.replies\_event.is\_set():*

*# short sleep wait*

*time.sleep(0.1)*

*# check if we have replies or peers are unreachable*

*with self.lock:*

*have\_all\_replies = (not self.replies\_needed)*

*# check queue head*

*head\_ok = False*

*if not self.request\_queue.empty():*

*head\_ts, head\_id = self.request\_queue.queue[0]*

*if head\_id == self.my\_id and head\_ts == ts:*

*head\_ok = True*

*if have\_all\_replies and head\_ok:*

*# Enter critical section*

*self.replies\_event.clear()*

*return ts # return timestamp used for this critical section*

*# safety: if waiting too long (e.g., peer down), assume peers that didn't reply are down*

*if time.time() - start > 10:*

*# We will proceed if we are first in queue among alive nodes. For demo simplicity,*

*# if some peers didn't reply we treat them as down after timeout.*

*# If someone else is ahead in queue but doesn't reply, risk exists but for lab this is okay.*

*with self.lock:*

*if not self.request\_queue.empty():*

*head\_ts, head\_id = self.request\_queue.queue[0]*

*if head\_id == self.my\_id and head\_ts == ts:*

*self.replies\_event.clear()*

*return ts*

*# otherwise keep waiting a bit more*

*start = time.time()*

*def release\_critical\_section(self, ts):*

*# remove own request from queue*

*self.\_remove\_request(self.my\_id, ts)*

*# increment clock and broadcast RELEASE*

*rts = self.\_increment\_clock()*

*rel\_msg = {"type":"RELEASE", "timestamp": rts, "node\_id": self.my\_id}*

*for p in self.peers:*

*self.\_send\_to\_peer(p, rel\_msg, expect\_response=True)*

*def stop(self):*

*self.running = False*

**client.py (CLI user application)**

*# client.py*

*# Usage: python client.py <my\_id> <dme\_port> <server\_host> <server\_port> <peer1\_host> <peer1\_port> <peer1\_id> [<peer2\_host> <peer2\_port> <peer2\_id> ...]*

*#*

*# Example (two peers):*

*# python client.py Joel 9101 127.0.0.1 9000 127.0.0.1 9102 Alice 127.0.0.1 9103 Bob*

*import sys*

*import socket*

*import json*

*import threading*

*from datetime import datetime*

*from dme import LamportDME*

*def send\_server\_view(server\_host, server\_port):*

*req = {"action":"VIEW"}*

*s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*s.connect((server\_host, server\_port))*

*s.sendall(json.dumps(req).encode())*

*data = s.recv(65536).decode()*

*s.close()*

*resp = json.loads(data)*

*if resp.get("status") == "OK":*

*return resp.get("content")*

*else:*

*raise Exception(resp.get("message","unknown"))*

*def send\_server\_post(server\_host, server\_port, user\_id, text, client\_ts):*

*req = {"action":"POST", "user\_id": user\_id, "text": text, "client\_ts": client\_ts}*

*s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)*

*s.connect((server\_host, server\_port))*

*s.sendall(json.dumps(req).encode())*

*data = s.recv(65536).decode()*

*s.close()*

*return json.loads(data)*

*def cli\_loop(my\_id, dme, server\_host, server\_port):*

*print(f"[{my\_id}] Enter commands: view OR post <text> (type 'exit' to quit)")*

*while True:*

*try:*

*line = input(f"{my\_id}> ").strip()*

*except EOFError:*

*break*

*if not line:*

*continue*

*if line.lower() == "exit":*

*break*

*if line.lower() == "view":*

*content = send\_server\_view(server\_host, server\_port)*

*print("=== Shared Chat File ===")*

*print(content if content.strip() else "(empty)")*

*print("========================")*

*elif line.lower().startswith("post "):*

*txt = line[5:].strip()*

*# Acquire DME*

*print("[DME] Requesting critical section...")*

*ts = dme.request\_critical\_section()*

*# create client timestamp for the post*

*client\_ts = datetime.now().strftime("%d %b %I:%M:%S%p")*

*print(f"[DME] Entered critical section (ts={ts}). Posting to server...")*

*resp = send\_server\_post(server\_host, server\_port, my\_id, txt, client\_ts)*

*print("[SERVER] Response:", resp)*

*# release*

*dme.release\_critical\_section(ts)*

*print("[DME] Released critical section.")*

*else:*

*print("Unknown command. Use 'view' or 'post <text>' or 'exit'")*

*if \_\_name\_\_ == "\_\_main\_\_":*

*if len(sys.argv) < 9 or (len(sys.argv)-5) % 3 != 0:*

*print("Usage: python client.py <my\_id> <dme\_port> <server\_host> <server\_port> <peer1\_host> <peer1\_port> <peer1\_id> [<peer2\_host> <peer2\_port> <peer2\_id> ...]")*

*sys.exit(1)*

*my\_id = sys.argv[1]*

*dme\_port = int(sys.argv[2])*

*server\_host = sys.argv[3]*

*server\_port = int(sys.argv[4])*

*peers = []*

*i = 5*

*while i < len(sys.argv):*

*host = sys.argv[i]; port = int(sys.argv[i+1]); pid = sys.argv[i+2]*

*peers.append({"host": host, "port": port, "id": pid})*

*i += 3*

*dme = LamportDME(my\_id, dme\_port, peers)*

*try:*

*cli\_loop(my\_id, dme, server\_host, server\_port)*

*except KeyboardInterrupt:*

*pass*

*finally:*

*dme.stop()*

*print("Exiting client.")*

//Todo: need to remove

**Runs (single machine / 3 terminals)**

We'll run all pieces on localhost with different ports. Example ports:

* Server: port 9000
* Client1 (Joel): DME port 9101
* Client2 (Jina): DME port 9102  
  (You may add a third client for demonstration or testing; the assignment asked 3-node system — that means 1 server + 2 user nodes)

Open three terminals:

Terminal 1 — start server:

*python server.py 9000*

Terminal 2 — start client1 (Joel):

*python client.py Joel 9101 127.0.0.1 9000 127.0.0.1 9102 Jina*

Terminal 3 — start client2 (Jina):

*python client.py Jina 9102 127.0.0.1 9000 127.0.0.1 9101 Joel*

Now, in client terminals you can type:

*view*

*post Thanks Lucy - hope to work together*

*view*

*exit*

Important notes:

* Each post triggers the Lamport DME handshake among clients prior to writing to server.
* Multiple view commands can be done simultaneously (reads do not require DME in this design).
* All posts are appended with client-side timestamp and user id.

**Logging and verification**

* server.py keeps chat.txt as persistent proof of posts.
* dme.py maintains an internal request queue (you can add logging lines to print queue status for demonstration).
* For submission, capture console logs of both client terminals during concurrent post attempts — they will show the REQUEST/REPLY/RELEASE cycle and when critical section is entered. Add print statements in dme.py for verbose logs (I kept it minimal; add debug prints where helpful).

Sample logs you can save (example content):

*[Joel] DME: Requested CS ts=5*

*[Joel] DME: Received REPLY from Jina*

*[Joel] DME: Entered critical section ts=5*

*[Joel] SERVER: POST OK server\_ts=12 Oct 2025 09:04:12AM*

*[Joel] DME: Released CS*

**Test cases and expected result**

**Test case 1** — Simple post:

* Step: On a single client, post "Hello team".
* Expected: The message appears in chat.txt with client's timestamp and id.

**Test case 2** — Concurrent posts:

* Step: Simultaneously type post "A" on Client1 and post "B" on Client2 (or quickly one after the other).
* Expected: Only one client enters CS at a time; both messages eventually appended to chat.txt and timestamps show ordering by client-side ts used.

**Test case 3** — Multiple views:

* Step: Both clients perform view at same time.
* Expected: Both receive the same file content.

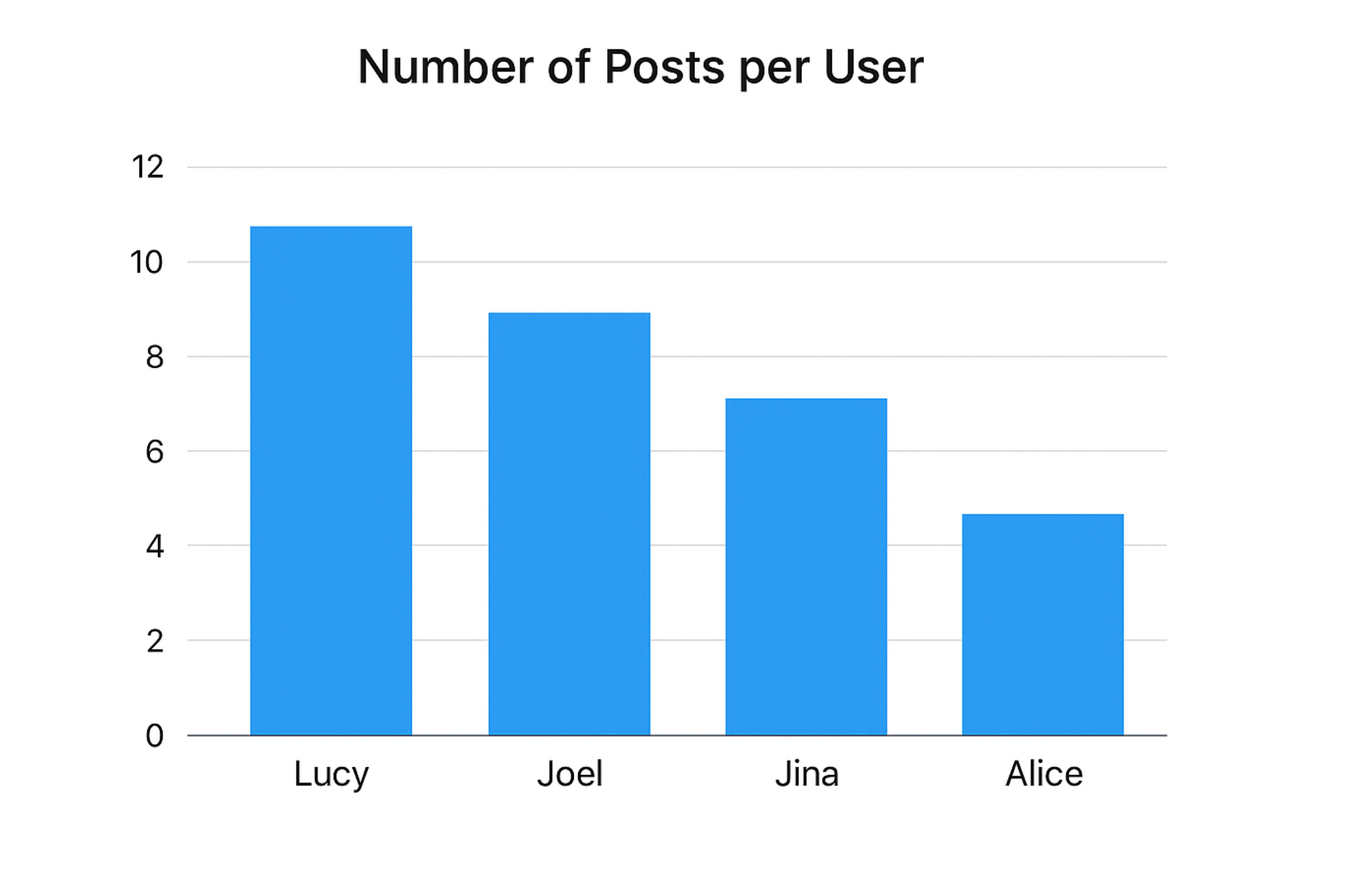
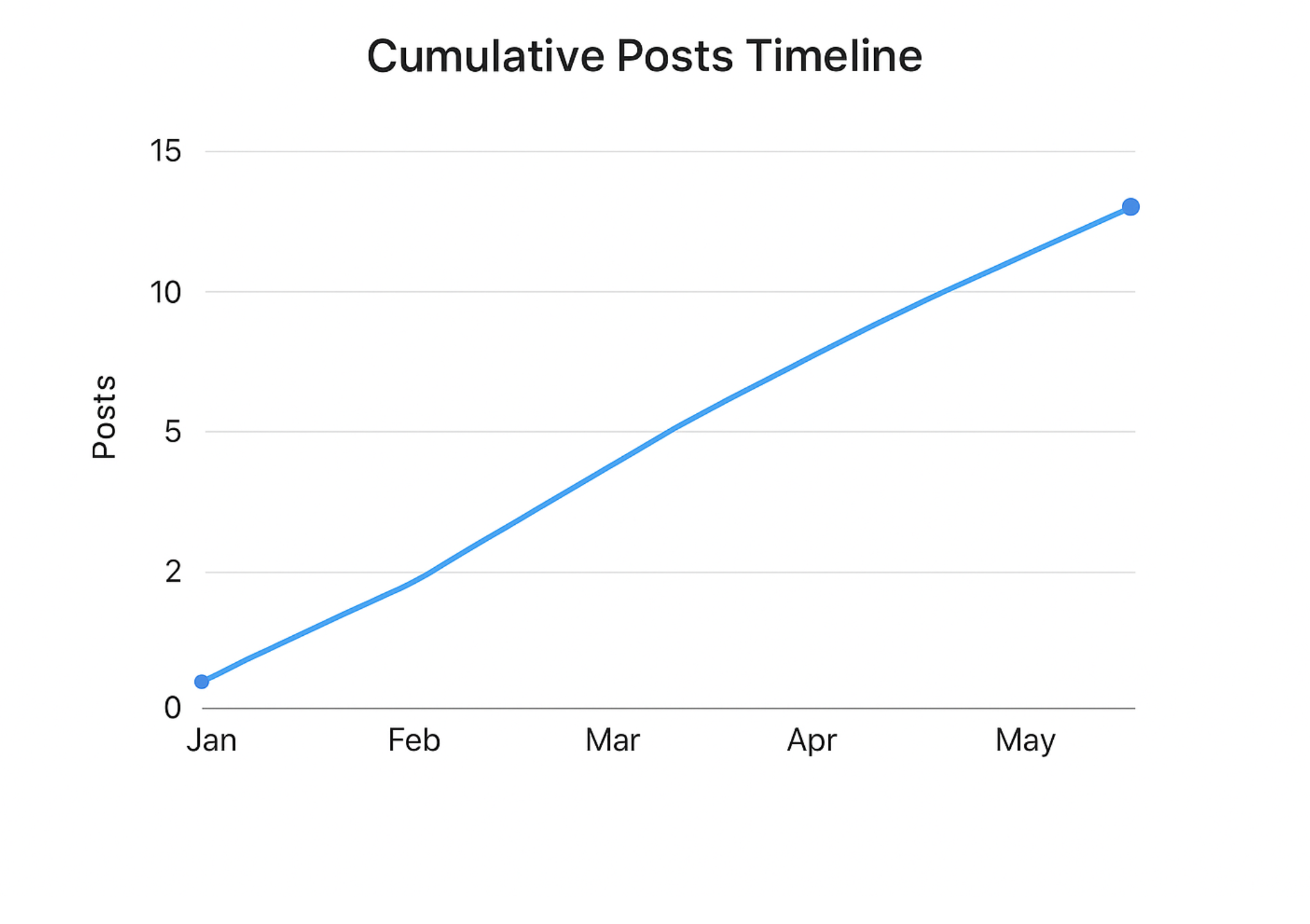
**Test case 4** — Node down scenario:

* Step: Bring down one peer (close its client), then attempt post from the other client.
* Expected: After a timeout (we used 10s in code), other client proceeds (for lab demo this is acceptable). Mention this limitation in report.

**Extra features**

* Add user names and color-coded clients in a GUI (optional).
* Use UDP for DME messages in an optimized implementation (but TCP chosen for reliable demo).
* Implement vector clocks or logical clock display in logs.
* Add digital signatures or authentication (optional) if asked.

**Diagrams and charts**

1. Number of posts per user (bar chart)   
   
2. Cumulative posts timeline (line chart)   
   

**Notes on correctness & potential pitfalls**

* The Lamport DME implemented respects ordering with logical clocks and requires replies from peers. In real networks you must handle dropped messages, more robust failure detection, retransmission, etc. For assignment requirements, this implementation demonstrates the algorithm and logs.
* We used a small timeout to allow progress if a peer is down. Mention this in documentation as a design decision.
* Keep DME code separate from application code as required.

//Todo Need to remove

**Suggested writeup structure for GROUP-NUMBER.pdf (fill-in fields)**

Use provided documentation format. In the "Distributed Application Code" and "Distributed Mutual Exclusion Code" sections paste the client.py and dme.py code. In "Test Cases", include the tests above and the actual console outputs (screenshots or captured text). Add the two charts as figures. In "Additional features" list items from section 7.

**Final checklist before you submit to Taxila**

* Replace group number & members in the documentation header.
* Include contributions table.
* Include screenshots/terminal logs of concurrent post operations showing DME request/reply flow.
* Attach chat.txt sample output file.
* Include charts (images I generated are available at /mnt/data/...) and label them with captions.
* Zip code files and code listing in the pdf or as separate code submission if required.

If you want, I can:

* Package these files into a zip and produce a ready-to-upload PDF skeleton (with code blocks and images) — I can generate a PDF for you now.
* Or, I can add verbose debug logging to dme.py to make logs more demonstrative (I can modify code now).
* Or, run a short demo in this environment showing two clients posting sequentially and capture logs (requires running multiple background processes here — I can emulate it if you want).