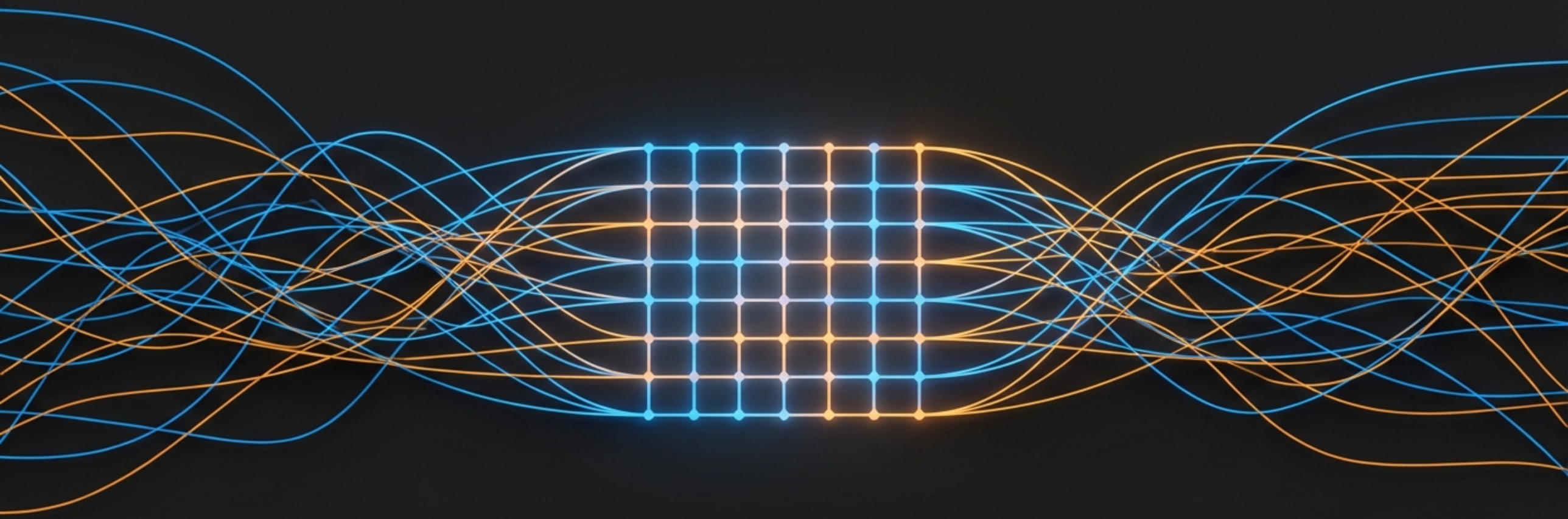


THE QUEST FOR CONSISTENCY IN A DISTRIBUTED CANVAS

A Collaborative Painting Application for CMPE 436

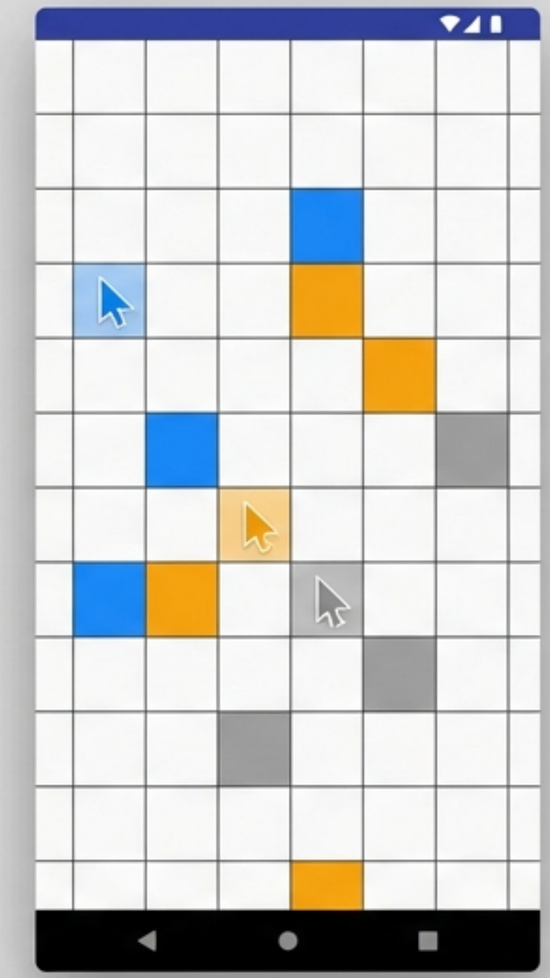
WE BUILT A REAL-TIME COLLABORATIVE WHITEBOARD

Simultaneous Painting: Multiple users, one shared canvas.

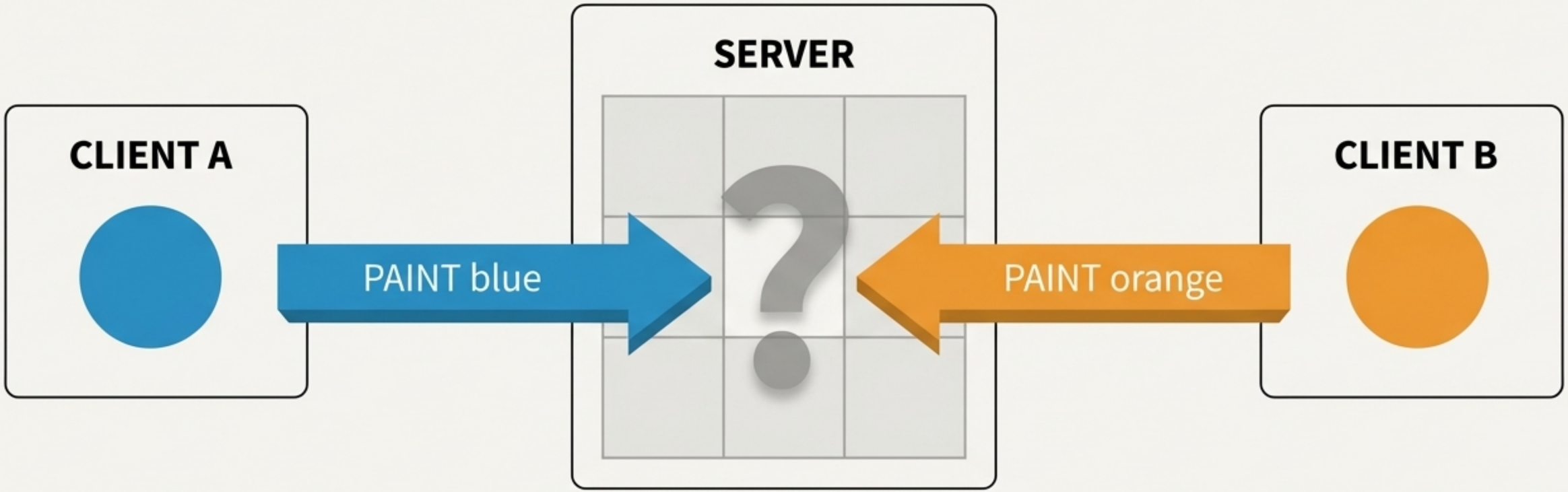
Consistent State: Every user sees the exact same image, always.

Persistent Canvas: The server saves and restores the state, surviving restarts.

Remote Management: Full control via admin commands.

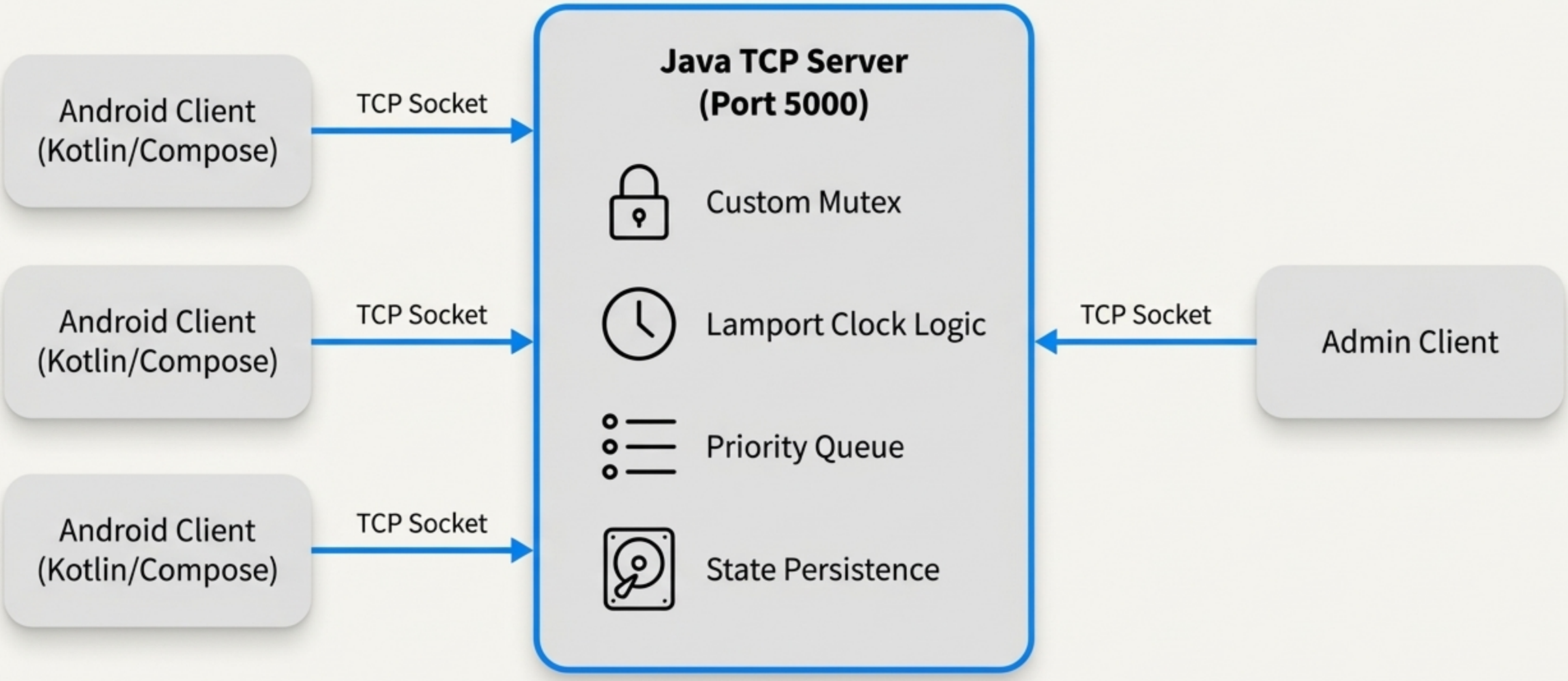


THE FUNDAMENTAL CHALLENGE: A RACE TO THE SAME PIXEL

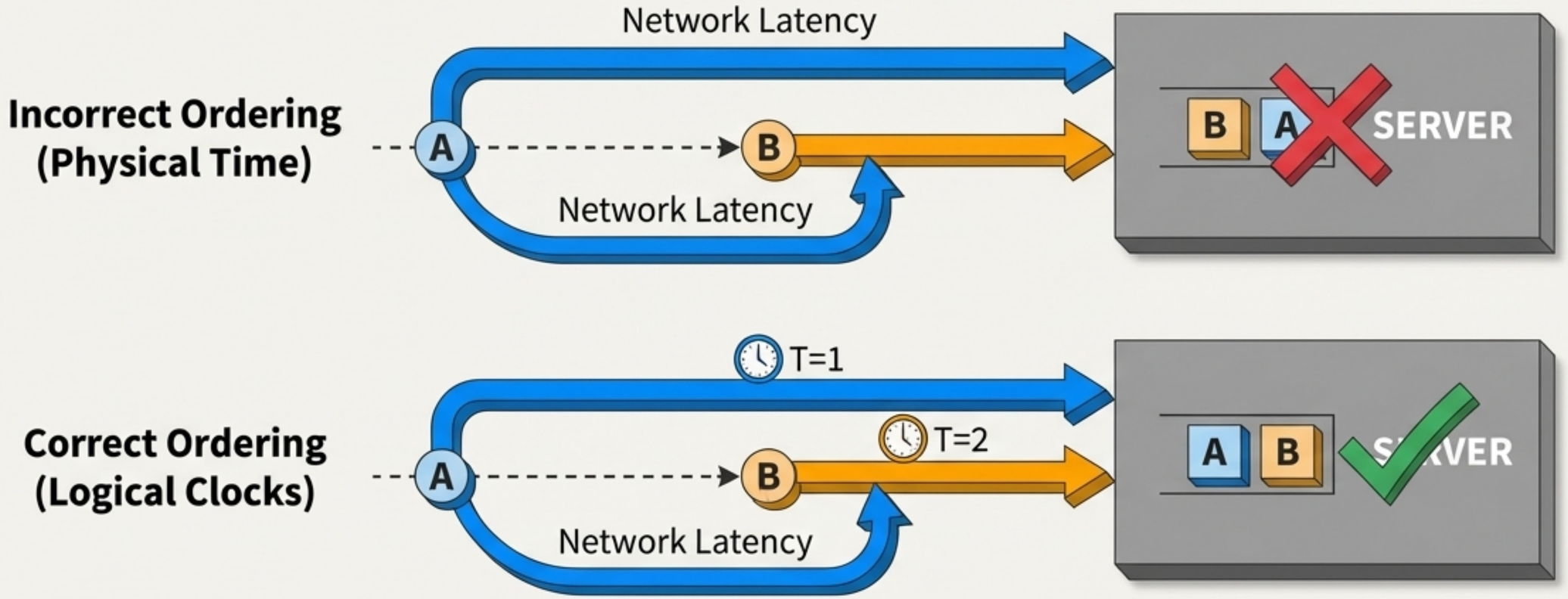


How do we guarantee that every user sees the same final color?
How do we definitively decide which action happened “first”
when network latency makes physical time unreliable?

THE SYSTEM ARCHITECTURE: A CENTRALIZED SERVER MODEL



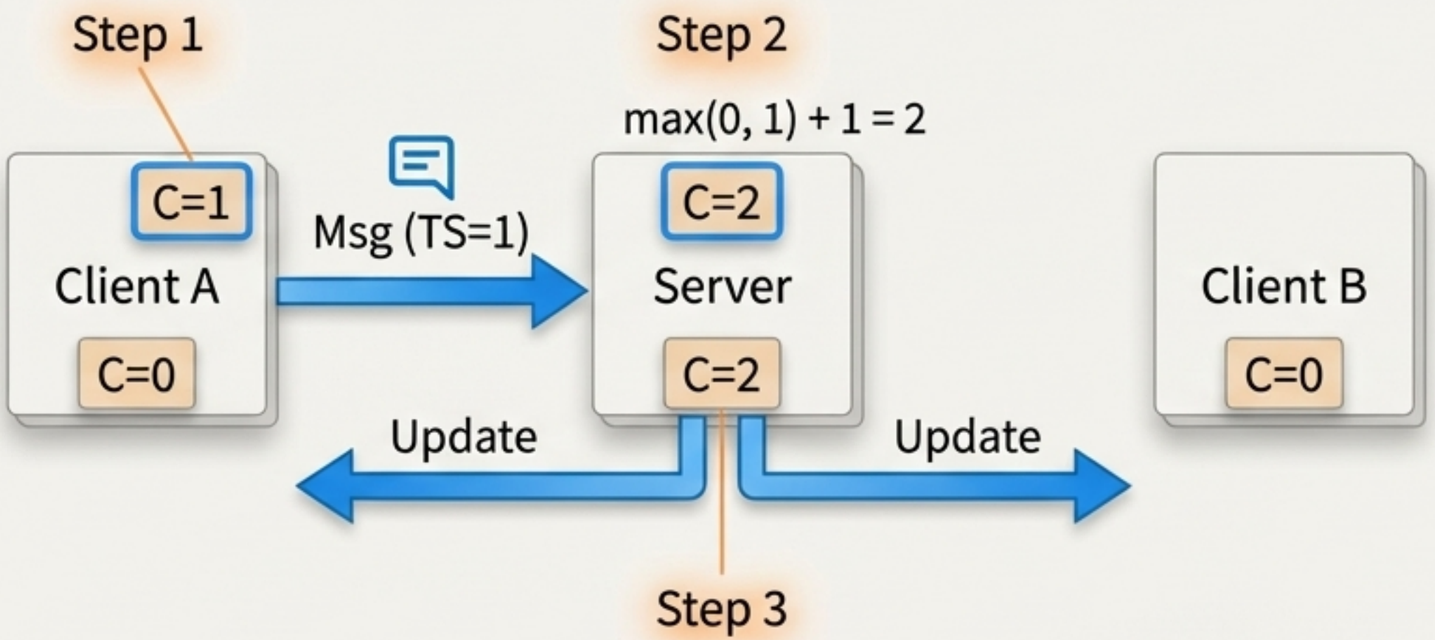
PILLAR 1: ESTABLISHING A CAUSAL ORDER FOR EVENTS



Problem: In a distributed system, we can't rely on wall-clock time. We need a logical way to order events.

Solution: We implemented Lamport's Logical Clock algorithm to create a consistent, causal timeline for all paint operations.

HOW OUR LAMPORT CLOCK WORKS

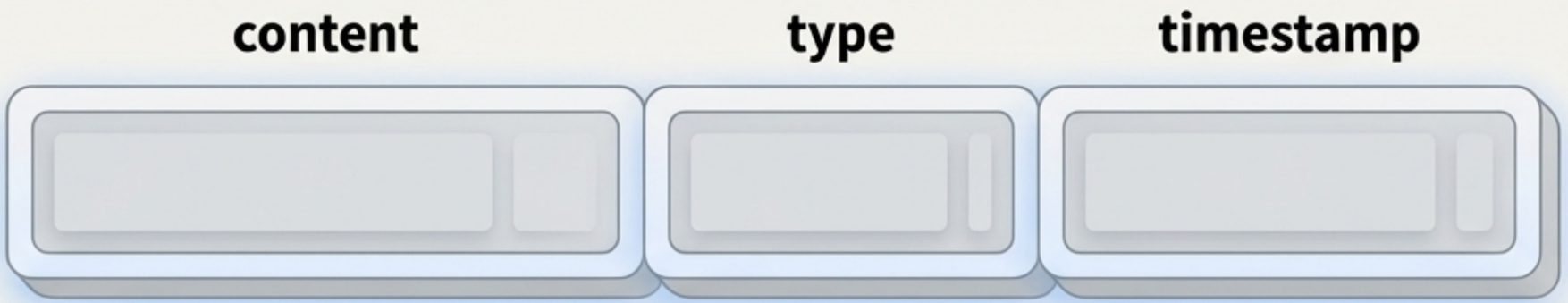


Core Rules:

- A process increments its clock before sending a message.
- On receiving a message, a process updates its clock: local_clock = max(local_clock, received_timestamp) + 1

FROM THEORY TO PRACTICE: THE MESSAGE PROTOCOL

All communication is a simple, structured message.




```
JOIN message: Aley|JOIN|1
PAINT message: 3,5,#FF0000|PAINT|42
CLEAR message: clear|CLEAR|101
```

The Lamport timestamp is a fundamental part of every single message, ensuring every action can be correctly ordered.

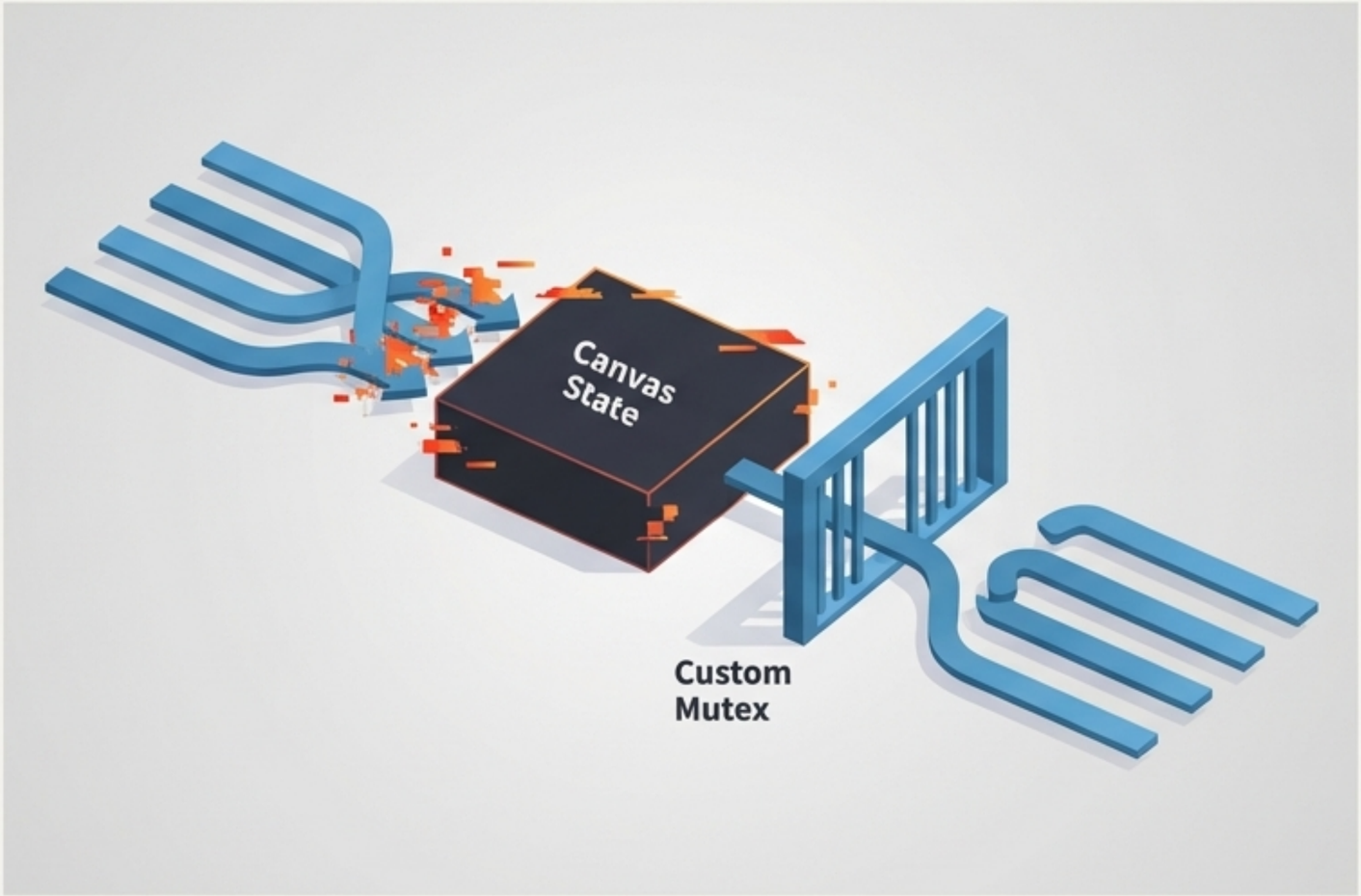
PILLAR 2: PROTECTING SHARED STATE ON THE SERVER

Problem Statement:

With a thread-per-client model, multiple threads might try to modify the canvas state simultaneously, leading to data corruption.

Our Approach:

To demonstrate a fundamental understanding of synchronization, we built a custom mutual exclusion lock from first principles, without using `synchronized` or `java.util.concurrent`.



ENGINEERING A CUSTOM MUTEX

Core Primitives Used:

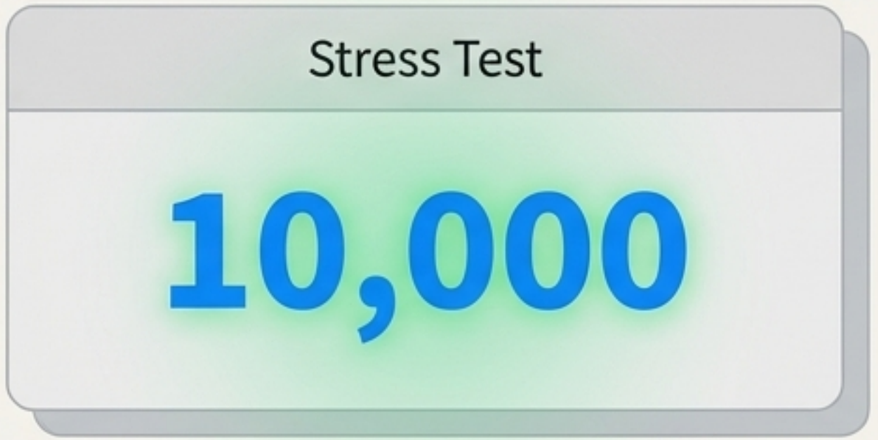
- `wait()`
- `notifyAll()`

Provided API:

- `lock()`
- `unlock()`
- `tryLock()`
- `await()` & `signalAll()` (for condition variables)

Proof of Robustness:

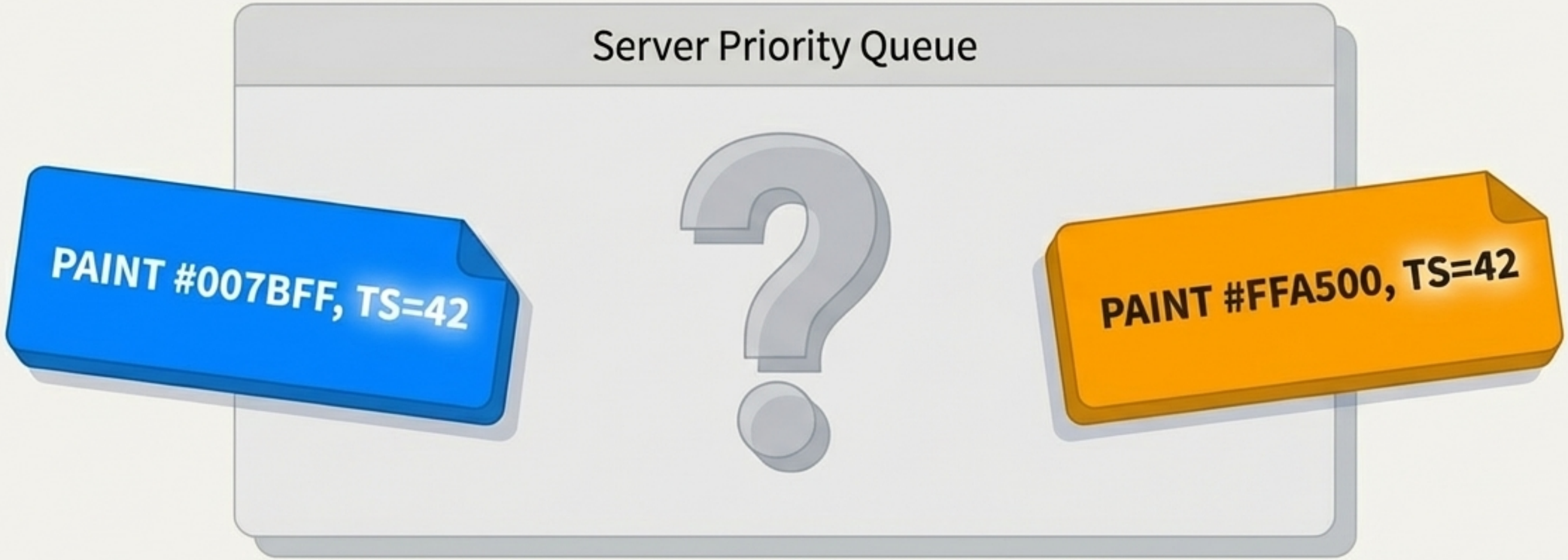
We stress-tested the implementation to ensure it was free of deadlocks and race conditions.



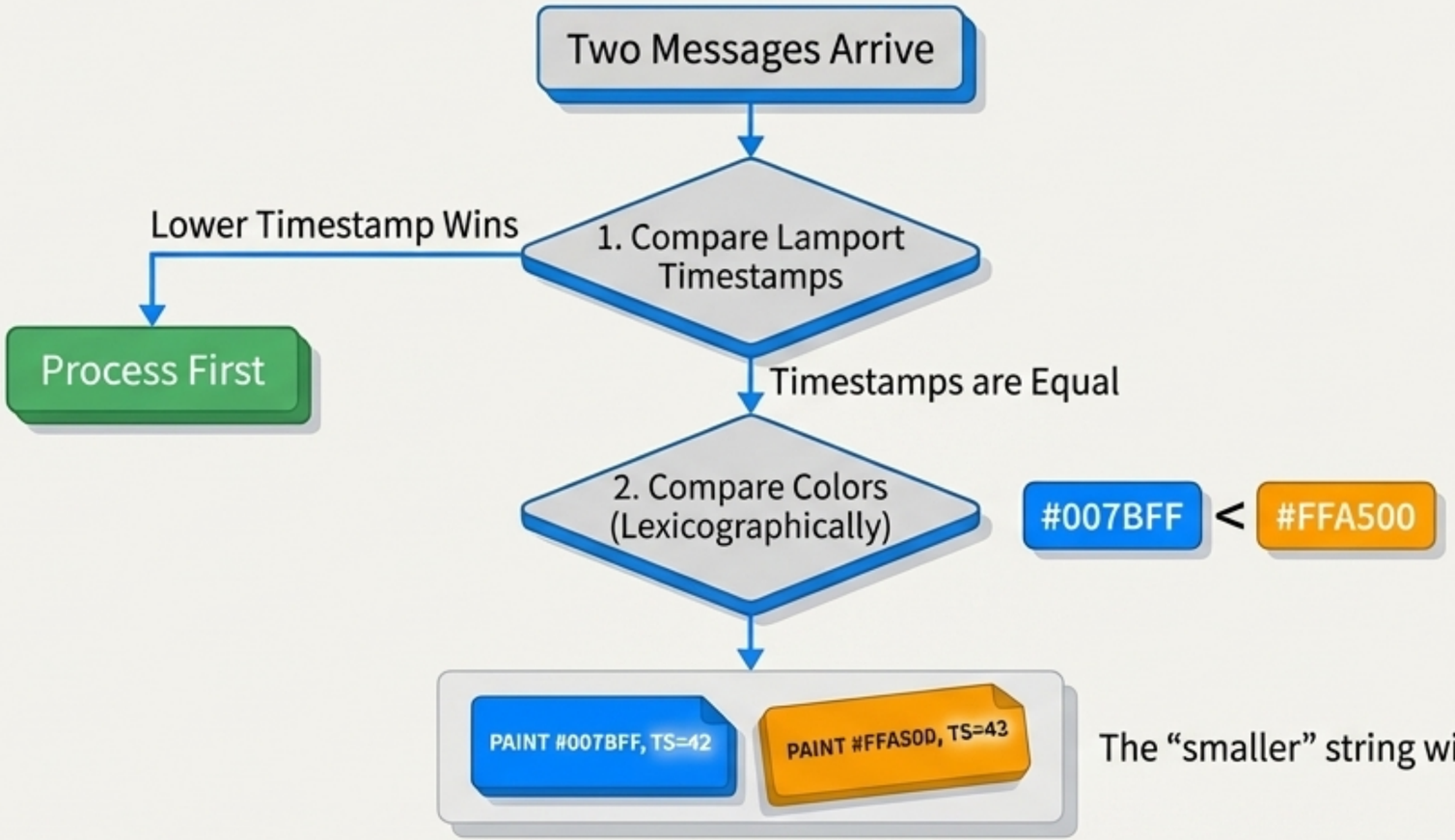
20 threads × 500 operations = 10,000 concurrent operations successfully executed without corruption.

PILLAR 3: A DETERMINISTIC CONFLICT RESOLUTION ALGORITHM

What happens if two paint operations arrive with the *exact same* Lamport timestamp? This is rare but possible. An ambiguity here would break consistency.

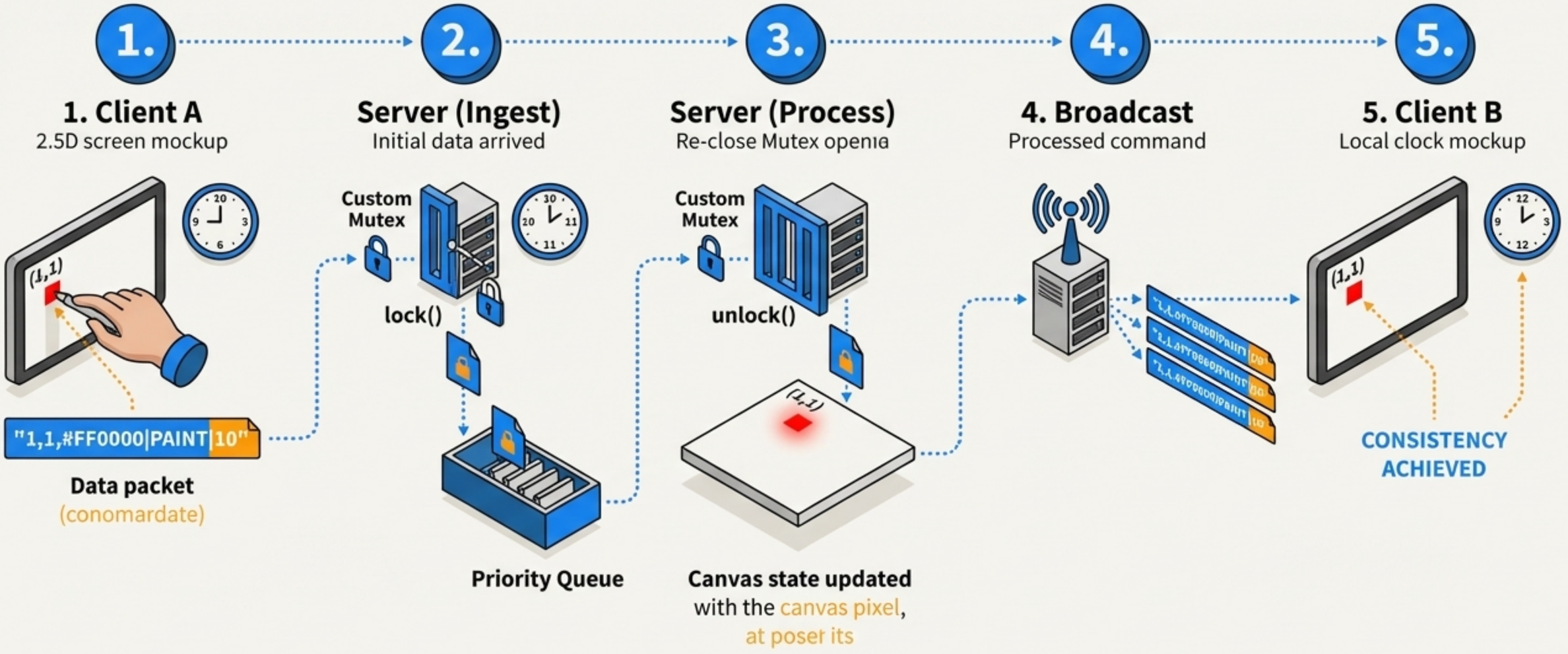


THE TWO-STEP TIE-BREAKER



The Result: A fully deterministic ordering. Given the same set of messages, every client and the server will always arrive at the identical canvas state. No ambiguity.

A PAINT STROKE'S JOURNEY THROUGH THE SYSTEM



A SYSTEM PROVEN THROUGH RIGOROUS TESTING

Lamport Clock Synchronization

Verified causal ordering across clients.

Custom Mutex Stress Test

Passed 10,000 concurrent operations.

Conflict Resolution

Tested all tie-breaking scenarios.

State Persistence

Confirmed successful save and recovery of canvas.

Concurrent Load

Simulated 5 clients painting simultaneously without errors.

Total Tests: **47** automated tests ensure system integrity.

PERFORMANCE UNDER PRESSURE

100+

Concurrent Clients Handled

<50ms

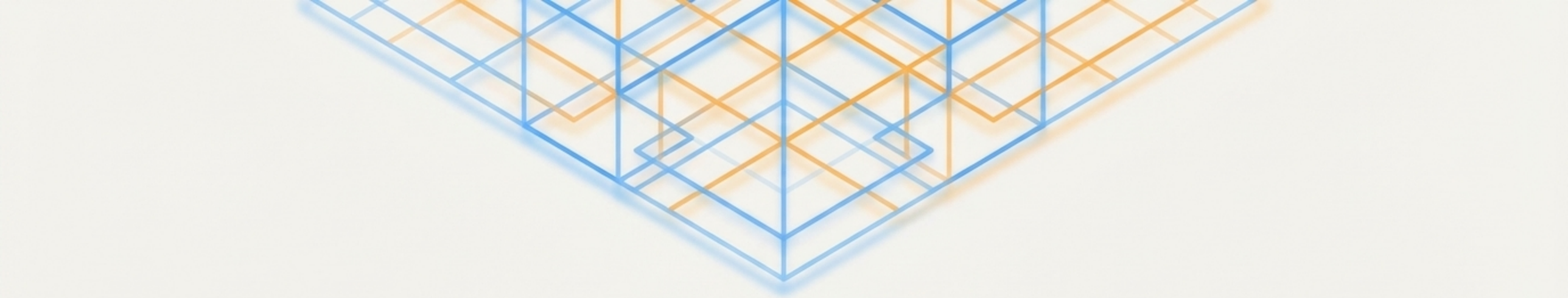
Latency for Paint Operations (Local Network)

1000+

Messages Processed per Second

Every 5 Ops

Auto-save triggers for State Persistence



MORE THAN A PAINTING APP: A STUDY IN APPLIED CONSISTENCY

By combining fundamental principles—Lamport’s logical clocks for causal ordering, careful mutual exclusion for state protection, and a deterministic algorithm for conflict resolution—we demonstrated that even in a chaotic distributed environment, perfect consistency is achievable.