Collaborative Whiteboard with Live Presence

C++ Backend · React (TypeScript) Frontend

---

1. Project Overview

Project Name
CollabBoard – Real-time collaborative whiteboard with live cursors and presence.

Problem Statement
Modern collaborative tools require low-latency, multi-user presence (cursor movement, drawing sync) that feels natural and scalable. Most systems hide this complexity behind managed services. This project builds the core real-time infrastructure from scratch, focusing on correctness, performance, and clean architecture.

Goal
Build a production-style collaborative whiteboard system that:
- Supports up to 15 concurrent users per room
- Displays live mouse cursors with smooth interpolation
- Supports shared drawing and text objects
- Uses server-authoritative real-time messaging
- Handles disconnects, errors, and network issues gracefully
- Is designed to scale to WebRTC and multi-instance backends later

---

2. Tech Stack

Frontend
- **React** (UI framework)
- **TypeScript** (type-safe real-time protocol)
- **HTML Canvas** (whiteboard rendering)
- **WebSocket API** (real-time communication)
- **State management**: Zustand (simpler than Redux for this use case)
- **Build tooling**: Vite

Backend
- **C++20**
- **Boost.Asio + Boost.Beast** (WebSocket server)
- **nlohmann/json** (message encoding)
- **CMake** (build system)
- Single-process, event-driven architecture

Optional / Future
- **MessagePack / Protobuf** (binary encoding - only if profiling shows need)
- **Redis Pub/Sub** (multi-instance scaling)

- **WebRTC Data Channels** (ultra-low-latency cursors)
- **SQLite** (board persistence)

---

3. System Architecture (High Level)

Backend Responsibilities
- Manage rooms and participants
- Assign user identity and color
- Route real-time messages with sequence numbers
- Broadcast presence updates
- Maintain authoritative room state
- Handle disconnects, timeouts, and errors
- Rate limiting per user

Frontend Responsibilities
- Capture pointer and drawing events
- Throttle and batch updates (20 Hz for cursors)
- Render board state and cursors
- Smooth cursor motion (linear interpolation)
- Manage reconnects and error states
- Display connection quality indicators

---

4. Core Features

v1 (MVP - Week 4 Demo)
- Join / leave rooms with basic room passwords
- Live mouse cursors with names + colors
- Shared drawing strokes (pen tool)
- Heartbeat + disconnect cleanup
- Error handling and reconnection
- Ghost cursor cleanup (auto-hide after 3s inactive)

v2 (Post-MVP Enhancements)
- Typing indicators
- Text objects with shared editing
- Eraser tool
- Board state snapshot for late joiners (with size limits)
- Undo/redo (local + synchronized)
- Persistence (optional)

v3 (Scale-Out)
- Binary protocol (if needed after profiling)
- WebRTC cursor transport
- Multi-instance backend with Redis

- Presence metrics & observability
- Network quality indicators

---

5. Backend Architecture (C++)

Key Classes

Connection Layer
WsSession
- WebSocket read/write loop
- Write queue with backpressure handling
- Heartbeat tracking
- User identity
- Error state management

Services Layer (Simplified)
RoomService
- Room creation and management
- User join/leave logic
- Message routing (type → handler)
- Participant list management
- Room password validation

PresenceService
- Cursor updates (with rate limiting)
- Last-seen timestamps
- Ghost cursor detection

BoardService
- Drawing stroke events
- Snapshot generation (with size limits)
- Stroke history management

Models
Room
- Room ID
- Participants (userId → UserInfo)
- Broadcast logic
- Stroke history (limited buffer)

UserInfo
- User ID, name, color
- Connection state
- Last activity timestamp

PresenceState

- Cursor position
- Active status

Infrastructure
- In-memory state store
- Logging framework (spdlog recommended)
- Metrics hooks (counters for messages, rooms, users)

---

6. Frontend Architecture (React)

Major Components
BoardPage – Room container, handles routing
BoardCanvas – Pointer capture, drawing logic, local rendering
CursorLayer – Renders remote cursors with interpolation
PresencePanel – User list with online indicators
Toolbar – Tool selection (pen, eraser in v2, text in v2)
ConnectionStatus – Shows connection health

Realtime Layer
wsClient.ts – WebSocket lifecycle management
- Connection/reconnection logic
- Exponential backoff on failures
- Heartbeat (ping/pong)

protocol.ts – TypeScript message types matching backend

outbox.ts – Message throttling and batching
- Cursor updates: 20 Hz (50ms intervals)
- Drawing strokes: batched by stroke
- Control messages: immediate

inbox.ts – Message deduplication and ordering
- Sequence number validation
- Out-of-order buffer

State Management (Zustand)
Room Store
- Users (Map<userId, UserInfo>)
- Cursors (Map<userId, CursorState>)
- Board state (strokes array)
- Connection status
- Error state

Selectors
- Computed view models for UI rendering
- Memoized cursor positions

Rendering Optimizations
- Cursor interpolation (lerp between updates)
- Ghost cursor cleanup (3 second timeout)
- requestAnimationFrame loop for smooth 60fps
- Canvas dirty region tracking
- Offscreen canvas for stroke caching

---

7. Message Protocol Design

Message Structure
All messages include:
```json
{
  "type": "message_type",
  "seq": 12345,        // Sequence number for ordering
  "timestamp": 167..., // Unix timestamp ms
  "data": { ... }
}
```

Presence Messages (loss-tolerant, high frequency)
cursor_move
```json
{
  "type": "cursor_move",
  "seq": 100,
  "data": { "x": 150, "y": 200 }
}
```

Drawing Messages (reliable)
stroke_start, **stroke_add**, **stroke_end**
```json
{
  "type": "stroke_add",
  "seq": 101,
  "data": {
    "strokeId": "uuid",
    "points": [[x1,y1], [x2,y2], ...],
    "color": "#000000",
    "width": 2
  }
}
```

Control Messages (reliable, critical)
join_room
```json
{
  "type": "join_room",
  "data": {
    "roomId": "room123",
    "userName": "Alice",
    "password": "optional"
  }
}
```

welcome (server → client on join)
```json
{
  "type": "welcome",
  "data": {
    "userId": "uuid",
    "color": "#ff5733",
    "users": [{ "userId": "...", "name": "Bob", "color": "..." }]
  }
}
```

room_state (snapshot for late joiners)
```json
{
  "type": "room_state",
  "data": {
    "strokes": [...], // Last N strokes only
    "snapshotSeq": 500
  }
}
```

user_joined / **user_left**
ping / **pong** (heartbeat)
error (generic error message)

Design Principles
- **Presence is ephemeral**: Lost cursor updates are acceptable
- **Control is authoritative**: Join/leave must be reliable
- **Drawing is reliable**: Strokes must arrive in order
- **Sequence numbers**: Enable ordering and gap detection
- **Rate limiting**: Server enforces max message rates per user

---

8. Error Handling Strategy

Client-Side Errors
- **WebSocket connection failure**: Exponential backoff retry (1s, 2s, 4s, max 30s)
- **Invalid message format**: Log and discard, show warning toast
- **Sequence gap detected**: Request room_state resync
- **Rate limit exceeded**: Throttle locally, show warning

Server-Side Errors
- **Malformed JSON**: Close connection with error code
- **Invalid room ID**: Send error message, disconnect
- **User limit exceeded**: Send error, reject join
- **Message too large**: Drop message, log warning
- **Rate limit exceeded**: Temporary mute user (10s)

Error Boundaries
- React error boundary wraps BoardPage
- Fallback UI with "Reload" button
- Errors logged to console + optional monitoring service

Recovery Patterns
- **Disconnect**: Client auto-reconnects, requests room_state
- **State corruption**: Server can force client refresh
- **Memory leak**: Server tracks per-room memory, evicts old strokes

---

9. Development Process

Phase 1 — Foundation (Week 1)

Backend Tasks
- WebSocket acceptor with Boost.Beast
- WsSession read/write loop
- Basic message codec (JSON)
- Room creation and join logic
- User assignment (ID, color, name)

Frontend Tasks
- React app scaffold with Vite
- WebSocket connection hook
- Room join UI (room ID + username + optional password)
- Basic connection status display

Deliverable: Two browsers can join the same room and see each other's names.

Phase 2 — Live Presence (Week 2)

Backend Tasks
- Cursor routing via PresenceService
- Rate limiting (20 Hz per user)
- Heartbeat + timeout cleanup (30s timeout)
- user_joined / user_left broadcasts

Frontend Tasks
- Pointer tracking on canvas
- Throttled cursor sends (50ms / 20 Hz)
- Cursor rendering layer with interpolation
- Ghost cursor cleanup (hide after 3s)
- User list UI

Deliverable: Live cursors visible across users with smooth motion.

Phase 3 — Whiteboard Core (Week 3)

Backend Tasks
- Stroke message handling (start/add/end)
- Stroke history per room (limit to last 1000 strokes)
- Board snapshot on join (send last N strokes)
- Sequence number enforcement

Frontend Tasks
- Drawing tools (pen only in v1)
- Stroke rendering on canvas
- Local prediction + server confirmation
- Canvas optimization (dirty regions)

Deliverable: Users can draw together in real time with synchronized strokes.

Phase 4 — Hardening (Week 4)

Backend Tasks
- Error message types
- Memory limits per room
- Metrics logging (messages/sec, rooms, users)

Frontend Tasks
- Reconnect handling with exponential backoff
- Error boundary and fallback UI

- Connection quality indicator
- Sequence gap detection and resync

Deliverable: Stable, demo-ready system that handles errors gracefully.

---

10. Performance Targets

- **Cursor latency**: < 100ms perceived (local network)
- **Cursor update rate**: 20 Hz (client throttle)
- **Max users per room**: 15 concurrent
- **Max strokes per room**: 1000 (older strokes dropped)
- **Max message size**: 64 KB
- **Server CPU usage**: < 5% per active room (local testing)
- **Memory per room**: < 50 MB
- **Reconnection time**: < 2s on network recovery

---

11. Snapshot Strategy

Problem
Late joiners need current board state, but sending thousands of strokes is expensive.

Solution (v2)
1. **Limit history**: Keep last 1000 strokes in memory
2. **Chunked delivery**: Split snapshot into multiple messages if needed
3. **Compression**: Consider gzip for large snapshots
4. **Incremental sync**: Send snapshot seq number, then stream new updates

v1 Simplification
- Send last 500 strokes as single message
- If > 64KB, send only last 200 strokes
- Client displays "loading board..." during snapshot

---

12. Why This Project Is Industry-Level

This project demonstrates:
- **Event-driven backend design** with Boost.Asio
- **Real-time systems thinking** (latency, throughput, loss tolerance)
- **Modern C++ practices** (RAII, smart pointers, move semantics)
- **Client-side performance optimization** (throttling, interpolation, canvas optimization)
- **Clean protocol design** (JSON schema, versioning, error codes)
- **Separation of concerns** (layers, services, models)
- **Error handling and recovery** (reconnection, rate limiting, graceful degradation)

- **Scalable evolution path** (WebRTC, Redis, binary protocol)

This is **infrastructure engineering**, not a CRUD app.

---

13. Testing Strategy

Unit Tests
Backend (Google Test)
- Room join/leave logic
- Message routing
- Cursor rate limiting
- Snapshot generation

Frontend (Vitest + React Testing Library)
- WebSocket client reconnection logic
- Cursor interpolation math
- Message throttling
- State store reducers

Integration Tests
- Multi-client scenarios (2-5 clients)
- Disconnect/reconnect flows
- Large room state (many strokes)
- Rate limit enforcement

Load Tests (Phase 4)
- 15 concurrent users per room
- Cursor spam (100 msg/sec per user)
- Memory leak detection (long-running rooms)

---

14. Future Extensions

v3+ Features
- **WebRTC data channels**: Peer-to-peer cursor updates
- **Multi-room persistence**: SQLite or PostgreSQL
- **Playback / time-travel**: Record and replay sessions
- **Mobile touch support**: Multi-touch drawing
- **Access control**: Room ownership, moderation
- **Undo/redo**: Operational transform or CRDT-based
- **Text tool**: Rich text with collaborative editing
- **Image upload**: Paste images onto board
- **Export**: PNG, SVG, PDF export

Architecture Evolution

- **Multi-instance backend**: Redis Pub/Sub for room routing
- **Binary protocol**: Protobuf or MessagePack for bandwidth optimization
- **CDN integration**: Asset delivery
- **Monitoring**: Prometheus metrics, Grafana dashboards
- **Authentication**: OAuth, JWTs

---

15. Dependencies and Setup

Backend Dependencies
```bash
# Ubuntu/Debian
sudo apt install cmake g++ libboost-all-dev

# Clone nlohmann/json
git clone https://github.com/nlohmann/json.git
```

Frontend Dependencies
```bash
npm create vite@latest collabboard-frontend -- --template react-ts
cd collabboard-frontend
npm install zustand
```

Project Structure
```
collabboard/
├── backend/
│   ├── CMakeLists.txt
│   ├── src/
│   │   ├── main.cpp
│   │   ├── ws_session.hpp
│   │   ├── room_service.hpp
│   │   ├── presence_service.hpp
│   │   ├── board_service.hpp
│   │   └── models/
│   └── tests/
├── frontend/
│   ├── src/
│   │   ├── App.tsx
│   │   ├── components/
│   │   ├── lib/
│   │   │   ├── wsClient.ts
│   │   │   ├── protocol.ts
│   │   │   └── outbox.ts
│   │   └── store/
```

```
│   │       └── roomStore.ts
│   └── package.json
└── README.md
```

---

16. Success Metrics

Demo Day Goals
- ✅ 3-5 users drawing simultaneously
- ✅ Smooth cursor movement (no jitter)
- ✅ Strokes appear within 200ms
- ✅ Graceful handling of disconnect/reconnect
- ✅ Clean, professional UI
- ✅ No crashes during 10-minute session

Code Quality Goals
- ✅ No memory leaks (valgrind clean)
- ✅ RAII patterns throughout C++ code
- ✅ TypeScript strict mode, no `any` types
- ✅ 80%+ test coverage on core logic
- ✅ Clean separation of concerns

End of Specification

This is a living document. Update as architecture decisions are made during development.