Co-op Canvas: A Distributed Real-Time Collaborative Drawing System

Bianca Alexandru

Faculty of Computer Science, Alexandru Ioan Cuza University, Iași, Romania
bianca-ana-maria.alexandru@info.uaic.ro

1 Introduction

Vision: Co-op Canvas was born from my desire to create a digital whiteboard that feels less like a corporate tool and more like a communal art studio. I wanted to build a cozy shared space where creativity can flow freely, making digital art fun for both beginners and experienced artists.

Technical Objective: The primary technical challenge was real-time synchronization. My goal was to build a robust system that balances the strict data integrity required for application state (like layer management) with the high-speed, low-latency demands of freehand drawing. The system supports professional-grade features such as pressure sensitivity, anti-aliasing, and multi-layer compositing, all synchronized over a custom hybrid network protocol.

2 Applied Technologies

Hybrid Network Protocol: I planned from early on to harness the best parts from both TCP and UDP communication protocols as follows:

- **TCP (Reliability):** I utilized TCP for critical operations where data loss is unacceptable—specifically logins, layer management (add/delete/reorder), undo/redo synchronization, and saving. If a "Layer Added" command were lost, the client state would permanently desynchronize from the server. TCP's guarantee of ordered, error-free delivery was therefore non-negotiable for these tasks.
- **UDP (Speed):** For the actual drawing strokes, cursor movements, and line interpolation, I chose UDP. High-frequency mouse movements generate hundreds of packets per second. TCP's overhead from acknowledgments and retransmissions would cause lag, ruining the artistic feel. With UDP, I can "fire and forget" these packets. As detailed in the optimization logs, I treat a lost pixel as acceptable in favor of responsiveness, though I implemented line interpolation to mitigate visual gaps.

3 Application Structure

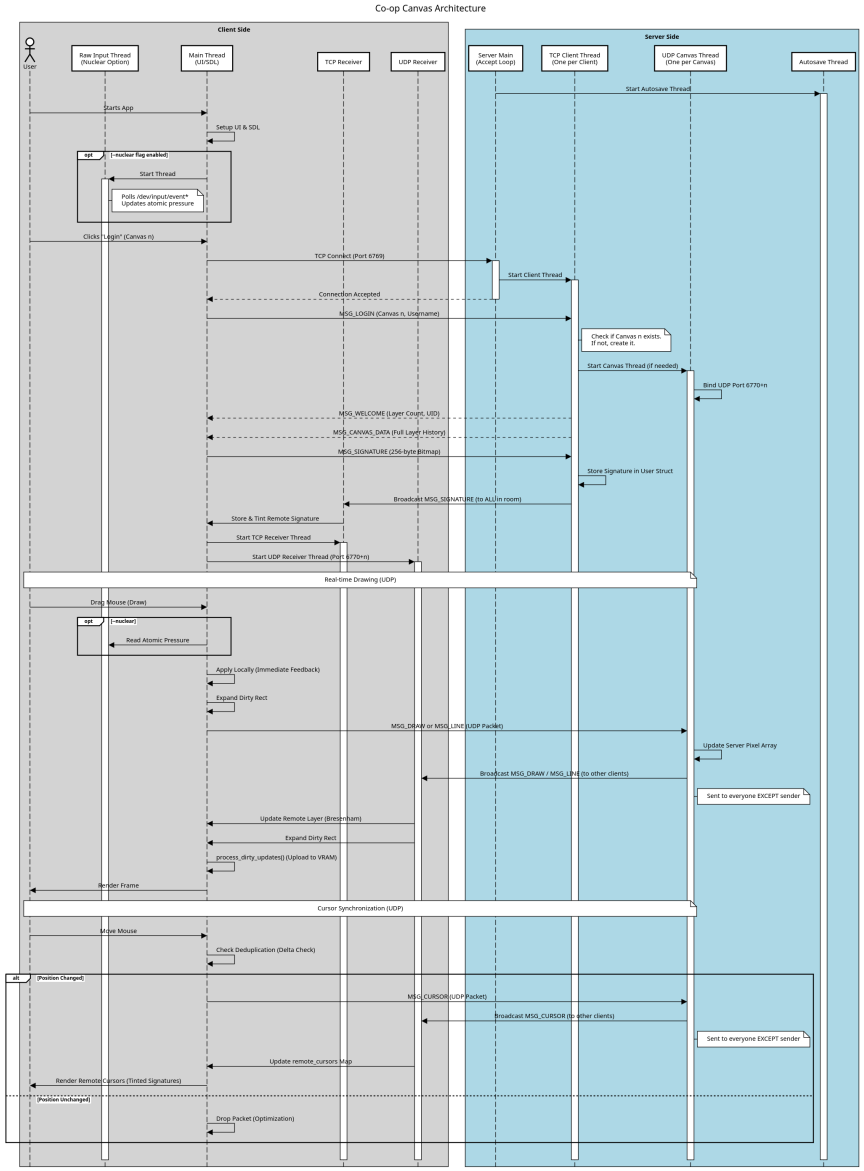

Fig. 1. Sequence Diagram of Co-op Canvas Architecture (Part 1)

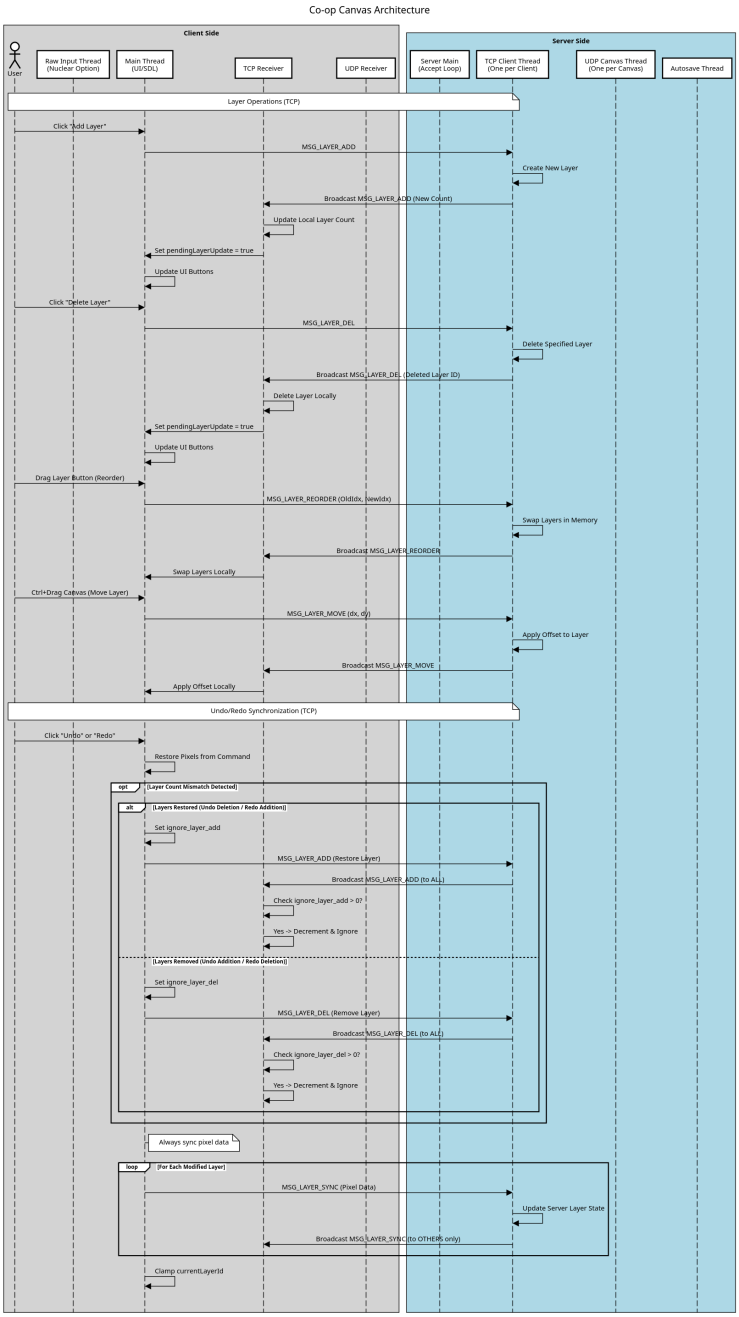

Fig. 2. Sequence Diagram of Co-op Canvas Architecture (Part 2)

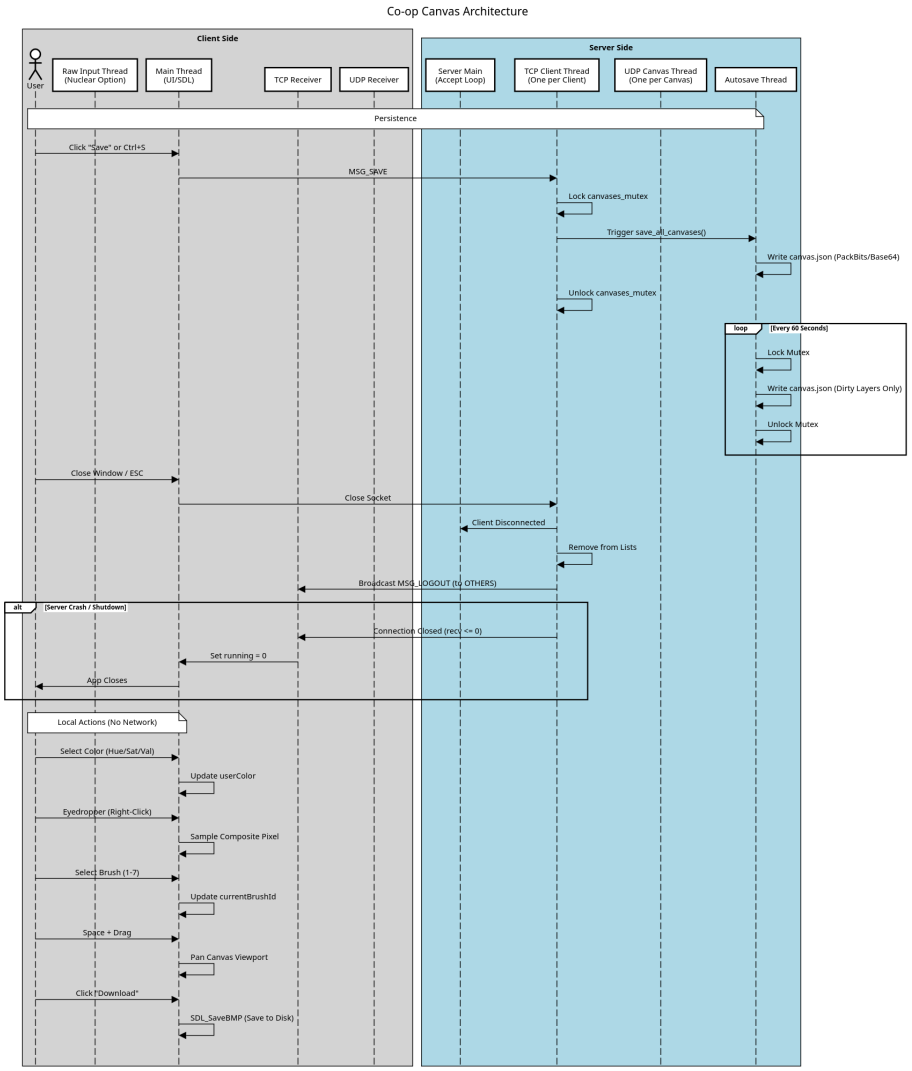

Fig. 3. Sequence Diagram of Co-op Canvas Architecture (Part 3)

3.1 Server Architecture

The server is designed as a multi-threaded, event-driven system capable of handling multiple concurrent drawing sessions ("Canvases").

Startup and On-Demand Creation The server does not pre-allocate resources for all possible canvases.

- **Initialization:** Upon startup, the server loads `canvas.json`, parses existing canvases, creates their `CanvasRoom` structures, and decodes their Base64 layer data into raw pixel memory.
- **On-Demand Logic:** When a client requests a login to a canvas ID that is not currently active in memory, the server calls `get_or_create_canvas(id)`. This function initializes the room, sets up the default "Paper" layer (white opaque) and "Layer 1" (transparent drawable), and spawns the necessary UDP threads. This ensures efficient memory usage.
- **Lazy Loading:** The server uses a `std::map` to store active canvases. Memory for the heavy pixel buffers (15 layers $\times$ 3.6MB) is only allocated when the first user joins a specific room. This allows the server to theoretically support thousands of empty rooms with zero memory overhead.

Threading Model

- **Main Accept Loop:** Blocks on `accept()` for port 6769. Spawns a new TCP session thread for every connecting client.
- **TCP Session Threads:** Handle the lifecycle of a single client connection. They process login handshakes, save commands, and layer operations. They also handle the "Welcome" sequence, sending the current layer structure and pixel data to new users.
- **UDP Canvas Threads:** Each active canvas has one dedicated thread listening on port $6770 + CanvasID$. This thread receives drawing packets and broadcasts them to all *other* clients in the same room. It protects the canvas pixel data using a mutex.
- **Autosave Thread:** A background thread wakes up every 60 seconds to serialize the state of all modified ("dirty") canvases to disk.

Server-Side Data Structures The server manages state using two primary structures:

- **Layer:** Contains a `uint8_t*` pixel buffer, an `opacity` value, and a `is_dirty` flag for the autosave system.
- `CanvasRoom`**:** A container for a specific drawing session. It holds a `std::vector<Layer*>`, a `std::vector<int>` of connected client socket IDs, and a `pthread_mutex_t` to ensure thread-safe access to the pixel data during UDP broadcasts.

Concurrency Control: To ensure high performance under heavy load, the server utilizes a **per-canvas mutex** strategy. Instead of a single global lock that would stall all rooms when one user draws, each `CanvasRoom` operates its own critical section. This allows users in Room A to draw at full speed even if Room B is currently performing a heavy layer reordering or saving operation.

Layer 0: The Immutable Paper A critical design choice was the implementation of "Layer 0". Unlike other layers, Layer 0 is initialized as a solid white

opaque buffer $(255, 255, 255, 255)$. It is marked as "locked" on the server, meaning no client can draw on it or delete it. This serves as the "Paper" of the canvas, ensuring that the composite image always has a background and preventing "see-through" artifacts when all drawable layers are transparent.

Server-Side Persistence (JSON) While the network protocol relies on optimized binary structures for real-time performance, the server utilizes a structured JSON format for persistent storage on disk. This ensures that the application state is human-readable and can be easily debugged or backed up. The `canvas.json` file follows this schema:

```
{
  "canvases": [
    {
      "id": 1,
      "layers": [
        { "id": 0, "opacity": 255, "data": "Base64..." },
        { "id": 1, "opacity": 200, "data": "Base64..." }
      ]
    }
  ]
}
```

3.2 Client Architecture

The client is engineered as a high-performance, multi-threaded application designed to decouple the strict timing requirements of the 60 FPS rendering loop from the unpredictable latency of network I/O. The architecture relies on four concurrent threads of execution.

- **Main Thread (UI & Rendering):** This thread owns the SDL2 window and OpenGL context. It executes the game loop, which performs three critical tasks every 16ms:
  1. Polls SDL events (keyboard/mouse) and maps them to brush actions.
  2. Checks the `pendingLayerUpdate` atomic flag to see if the UI needs rebuilding due to network events.
  3. Uploads "dirty" pixel regions from RAM to the GPU and presents the frame.
- **TCP Receiver Thread:** A blocking thread dedicated to the control plane. It listens for state-critical messages (`MSG_LAYER_ADD`, `MSG_WELCOME`). Because SDL2 is not thread-safe, this thread does not modify the UI directly; instead, it updates the internal data structures (protected by `layerMutex`) and signals the Main Thread via atomic flags.
- **UDP Receiver Thread:** A high-priority thread dedicated to the data plane. It runs a tight loop processing `MSG_DRAW` and `MSG_CURSOR` packets.

It writes pixel data directly into the system RAM buffers (`layers[]`) and marks regions as "dirty" for the renderer. It utilizes the "Socket Draining" optimization (using `MSG_DONTWAIT`) to prevent packet backlogs.
- **Raw Input "Sidecar" Thread (Optional):** When the application is launched with the `-nuclear` flag, this thread starts via `RawInput_Start()`. It bypasses the window manager to read directly from the Linux kernel input nodes (`/dev/input/event*`). It parses binary event structs to extract high-resolution pressure data and updates a global `std::atomic<int>` variable, which the Main Thread reads during drawing.

The Hybrid Memory Model (Rendering Architecture) To enable high-speed network synchronization without stalling the GPU, the client maintains two distinct representations of the canvas. This "Hybrid" model moves from a CPU-bound compositing model to a GPU-accelerated one to achieve 60 FPS performance.

- **System RAM (Source of Truth):** An array of `uint8_t* layers[]` buffers represents the raw pixel data. All network threads (TCP/UDP) and brush logic operate exclusively on this memory. This allows for fast, lock-free reads during color picking and saving.
- **VRAM (Display State):** The client mirrors these arrays into `SDL_Texture*` objects residing on the GPU.
- **Batch Updates & Dirty Rectangles:** Instead of re-uploading the entire texture every frame, the client tracks "Dirty Rectangles" (regions modified by brushes). In the main render loop, `process_dirty_updates()` uploads only the modified pixels to the GPU exactly once per frame.
- **Compositing:** The final image is created by the GPU using `SDL_RenderCopy` for each layer texture, utilizing hardware alpha blending.

UI Framework and Event Dispatch The UI is built on a custom lightweight object-oriented framework defined in `ui.h`:

- **Component Architecture:** The abstract `Button` class provides the interface for `Draw()` and `Click()`. Specialized derivatives like `BrushButton`, `LayerButton`, and `ColorPicker` encapsulate their own rendering logic and network triggers.
- **Coordinate Transformation:** To support the "Infinite Canvas" feel, the client implements a Viewport system. Global variables `viewOffsetX` and `viewOffsetY` track the camera position. The Event Dispatcher intercepts raw mouse coordinates and transforms them ($Mouse + Offset$) before passing them to the canvas logic, while leaving UI clicks in screen space.
- **CLI Onboarding:** Upon connection, the client outputs a text-based tutorial to the standard output (terminal), instructing the user on advanced shortcuts (e.g., `Space+Drag` to pan, `[/]` for layers) that are not immediately visible in the GUI.

Startup and Login Flow

- **Animated Lobby:** The client loads a UI definition from `ui.json`. It renders an animated background by compositing static layers (1-12) with alternating animation frames (Layers 13/14) every 1000ms.
- **Handshake:** Upon clicking "Login", the client connects via TCP. It receives a `MSG_WELCOME` containing the canvas configuration and then enters a loop to download the full pixel data for all layers.
- **UDP Setup:** After TCP synchronization, the client initializes a UDP socket connected to the server's calculated canvas port ($6770 + CanvasID$) and spawns the receiver thread.

Data Integrity (Mutex Protection) The client's multi-threaded architecture introduces significant concurrency challenges. The TCP thread (handling state changes) and the Main thread (handling rendering) frequently access shared data structures simultaneously. To prevent race conditions, I implemented a `pthread_mutex_t` named `layerMutex`. This lock guards all critical access to the `layers` vector and `layerCount`, ensuring that the renderer never attempts to read a pixel buffer while the network thread is in the middle of resizing or deleting it.

UI Thread Safety (The Flag Pattern) A separate stability challenge was handling UI updates triggered by network events. SDL2 textures and UI rendering functions are *not* thread-safe and must only be accessed by the main thread. Attempting to create buttons from the TCP thread results in immediate segmentation faults. To solve this, I implemented a deferred update mechanism. Instead of the TCP thread modifying the UI directly, it simply sets an atomic flag `pendingLayerUpdate = true`. The main rendering loop checks this flag at the start of every frame; if set, it safely rebuilds the layer buttons and UI elements before the next draw call.

Client-Side Prediction (Zero-Latency) To ensure a responsive drawing experience, the client decouples the visual feedback from the network round-trip time (RTT).

- **Immediate Feedback:** When the user draws, the client calls `brush->paint()` on the local buffer immediately, *before* sending the UDP packet.
- **Fire-and-Forget:** The UDP packet is sent asynchronously. This means the user sees their own stroke instantly (0ms latency), even if the network has a 200ms delay.

4 Communication Protocol

4.1 Message Types

The protocol uses a 1-byte header to identify message types.

- **0x01** `MSG_LOGIN`**:** Client joins a canvas.
- **0x02** `MSG_LOGOUT`**:** Graceful disconnect.
- **0x03** `MSG_WELCOME`**:** Server acknowledges login, sends layer count and User ID.
- **0x04** `MSG_CANVAS_DATA`**:** Bulk pixel transfer during sync.
- **0x05** `MSG_SAVE`**:** Triggers server-side JSON save.
- **0x06** `MSG_DRAW`**:** UDP packet for a brush stroke at (x, y).
- **0x07** `MSG_CURSOR`**:** UDP packet for remote cursor tracking.
- **0x08** `MSG_LINE`**:** UDP packet containing start (x, y) and end (ex, ey) coordinates. This is a bandwidth optimization: instead of sending a burst of points for a fast stroke, the client sends a single line segment, and the receiver interpolates it.
- **0x09** `MSG_ERROR`**:** Error reporting.
- **0x0A** `MSG_LAYER_ADD`**:** Request to add a layer.
- **0x0B** `MSG_LAYER_DEL`**:** Request to delete a layer.
- **0x0C** `MSG_LAYER_SELECT`**:** (Internal) Switch active layer.
- **0x0D** `MSG_LAYER_SYNC`**:** Full layer data upload. Used by the Undo system to push the restored state to the server.
- **0x0E** `MSG_LAYER_REORDER`**:** Swaps the Z-index of two layers.
- **0x0F** `MSG_SIGNATURE`**:** Transmits a 256-byte compressed signature bitmap.
- **0x11** `MSG_LAYER_MOVE`**:** Transmits (dx, dy) offsets when a user drags a layer with Ctrl+Click.

4.2 Binary Message Structures

To ensure cross-platform compatibility and prevent compiler-inserted padding from corrupting the network stream, all protocol structures are marked with `__attribute__((packed))`. This forces the compiler to align members on 1-byte boundaries, making the memory layout identical on both the sender and receiver regardless of the architecture.

TCP Control Structure The `TCPMessage` structure is used for all reliable control traffic (logins, layer management, undo/redo). It uses a fixed-size header followed by a small payload for metadata.

```
struct TCPMessage {
    uint8_t  type;         // Message ID (0x01 - 0x11)
    uint8_t  canvas_id;    // Target Canvas Room
    uint16_t data_len;     // Length of optional payload
    uint8_t  layer_count; // Used in MSG_WELCOME
    uint8_t  layer_id;     // Target layer for operations
    uint8_t  user_id;      // Unique ID assigned by server
    char     data[256];    // Generic payload (filenames, etc.)
} __attribute__((packed));
```

- **type:** The 1-byte header identifying the operation (e.g., `MSG_LAYER_ADD`).

- **canvas_id:** Identifies which drawing session the message belongs to.
- **data_len:** Specifies how many bytes of the `data` array are valid, allowing for variable-length string transmission within the fixed buffer.
- **layer_count / layer_id:** Context-specific fields. `layer_count` is sent by the server during the welcome handshake, while `layer_id` targets a specific layer for deletion or selection.
- **user_id:** A unique identifier assigned by the server to each client, used to associate UDP cursor packets with the correct user signature.
- **data:** A 256-byte buffer used for auxiliary information like layer names, error messages, or small data blobs.

UDP Data Structure The `UDPMessage` is optimized for high-frequency transmission. It contains all the necessary parameters for a single brush stroke or line segment in a compact 17-byte payload.

```
struct UDPMessage {
    uint8_t  type;       // MSG_DRAW, MSG_CURSOR, or MSG_LINE
    uint8_t  brush_id;   // Selected brush (Round, Square, etc.)
    uint8_t  layer_id;   // Active layer being drawn on
    int16_t  x, y;       // Start coordinates
    int16_t  ex, ey;     // End coordinates (for MSG_LINE)
    uint8_t  r, g, b, a;// RGBA color components
    uint8_t  size;       // Brush diameter
    uint8_t  pressure;   // 0-255 pressure value
} __attribute__((packed));
```

- **type:** Distinguishes between a single point (`MSG_DRAW`), a cursor update (`MSG_CURSOR`), or an interpolated segment (`MSG_LINE`).
- **brush_id:** Tells the receiver which `paint()` method to invoke from the polymorphic brush system.
- **x, y / ex, ey:** 16-bit signed integers representing canvas coordinates. This allows for a coordinate range of $\pm 32,767$, supporting large canvases while minimizing packet size.
- **r, g, b, a:** The RGBA color state, allowing each stroke to have its own color and transparency.
- **size:** The diameter of the brush in pixels.
- **pressure:** A 0-255 value captured from the tablet driver, used to modulate brush dynamics in real-time.

4.3 Port Scheme

The system uses a deterministic port allocation strategy:

- **Base TCP Port:** 6769 (All control traffic).
- **UDP Port Formula:** $UDP\_PORT = 6769 + 1 + CanvasID$.
- This ensures traffic from Canvas 0 (Port 6770) never interferes with Canvas 1 (Port 6771).

4.4 Packet Loss Mitigation (Bresenham Interpolation)

Since UDP is unreliable, packets representing parts of a stroke may be lost.

- **The Issue:** If packet N is lost, the receiver sees a gap between the coordinates of packet $N-1$ and $N+1$.
- **The Fix:** The receiver does not just draw points. It uses **Bresenham's Line Algorithm** to draw a continuous line between the last received point and the current point. This ensures that even with 10-20% packet loss, the visual stroke remains continuous and smooth.

5 User Interface and Interaction

5.1 Identity System: The Death of the Username

In a standard collaborative app, users are identified by text strings (e.g., "User123"). For Co-op Canvas, I wanted to enforce the artistic theme even in the login process. I completely removed the concept of a text-based username. Instead, a user's identity is defined solely by their **hand-drawn signature**.

Note on Security: It is important to clarify that this is a **visual recognition system** designed for artistic expression and social presence. It does not utilize cryptographic digital signatures (such as Public/Private keys or Ed25519 signing). Users recognize each other by their unique hand-drawn "tags" rather than through a secure authentication protocol.

The Signature Mechanic When a user launches the application, they are presented with a blank "Signature Box" instead of a text field. They must draw a unique symbol or name using the mouse or tablet. This drawing becomes their persistent identity for the session.

- **Visual Feedback:** When User A is drawing on the canvas, User B sees User A's signature floating next to their cursor.
- **Self-Filtering:** To prevent visual clutter, the client explicitly hides the local user's own floating signature, showing only the signatures of other collaborators.
- **Dynamic Coloring:** The signature is not a static image. It is transmitted as an alpha mask (grayscale). On the receiver's side, the signature is tinted in real-time to match the user's currently selected brush color. If User A switches from Red to Blue, their floating signature immediately turns Blue on User B's screen. This provides intuitive feedback about who is drawing what.

Communication Flow

1. **Login:** Client sends `MSG_SIGNATURE` (256 bytes) immediately after connection.

2. **Server Storage:** The server stores this blob in the `User` struct associated with the socket.
3. **Broadcast:** When a new user joins, the server broadcasts their signature to all existing users. Conversely, the server sends the signatures of all existing users to the new joiner.
4. **Client Caching:** The client decompresses the blob once into an `SDL_Texture` and caches it in a `std::map<int, RemoteClient>`. This ensures the expensive decompression step happens only once per user, not every frame.

5.2 Main Canvas Interface

Fig. 4. Login Screen Interface

The login screen serves as the entry point for the application:

1. **Signature Box:** A dedicated area where the user draws their unique signature. This drawing is captured, compressed, and used as their identity marker in the session.
2. **Previous Canvas (<):** Decrements the target Canvas ID.
3. **Next Canvas (>):** Increments the target Canvas ID.
4. **Login Button (GO!):** Initiates the TCP connection to the selected room.

Fig. 5. Main Drawing Interface

5. **Color Picker:** Select a color of the current hue.
6. **Hue Slider:** A rainbow gradient bar for selecting the base hue.
7. **Size Controls:** Buttons (+/-) to adjust the size of the current brush.
8. **Download:** Downloads the current drawing as a .bmp file.
9. **Eraser Tool:** Shortcut to select the Hard Eraser brush.
10. **Current Color Preview:** Displays the currently selected RGBA color.
11. **Brush Palette:** Vertical toolbar for selecting brush types (Round, Square, Pressure, Airbrush, etc.).
12. **Save Button:** Triggers a server-side save of the current canvas state to JSON.
13. **Layer Stack:** Visual representation of the layer hierarchy. The active layer is highlighted.
14. **Layer Controls:** Buttons to Add (+) or Delete (-) layers.
15. **Undo/Redo:** Redo only appears when its stack is not empty.

Keyboard Shortcuts:

- **General:**
  - **ESC:** Logout / Exit Application.
  - **TAB:** Toggle UI Visibility (Hide/Show Interface).
- **Lobby:**
  - **0-9:** Type a two-digit number (00-99) to jump to a specific Canvas ID.
- **Tools:**
  - **1-7:** Select Brush (1:Round, 2:Square, 3:Eraser, 4:Soft Eraser, 5:Pressure, 6:Airbrush, 7:Textured).
  - **Q** / **W:** Decrease / Increase Brush Size.
  - **A** / **S:** Decrease / Increase Brush Opacity.
- **Layers:**
  - **[** / **]:** Select Previous / Next Layer.
  - **Ctrl** + **Drag:** Move the content of the current layer.
- **Actions:**
  - **Ctrl** + **Z** / **Ctrl** + **Y:** Undo / Redo.
  - **Ctrl** + **S:** Save Canvas to Server.
  - **Space** + **Drag:** Pan Canvas View.

Panning and "The Void" To support large-scale collaborative art, I implemented a Panning system.

- **The Void:** The area outside the canvas is rendered in a deep "Space Gray" (#28283C). This provides visual contrast and a sense of "infinite" workspace.
- **View Offsets:** The client maintains `viewOffsetX` and `viewOffsetY` variables. All mouse coordinates are transformed from screen-space to canvas-space by subtracting these offsets before being sent to the server.
- **Navigation:** Users can pan the view by holding the `Spacebar` and dragging with the left mouse button, or by using the middle mouse button.

6 Brush System Implementation

To achieve a natural drawing feel, I implemented a polymorphic `Brush` system in `brushes.h`. Each brush inherits from a base class and implements a custom `paint()` method. This allows for complex behaviors like pressure sensitivity and texture mapping to be encapsulated cleanly.

6.1 Round Brush (Standard)

The most basic brush. It iterates over a square area of size $2r \times 2r$ and fills pixels where $x^2 + y^2 \leq r^2$. This creates a standard hard-edged circle.

```
// Round Brush:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    (void)angle; (void)pressure; // Parameters must be present
    int r = size / 2;
    if (i*i + j*j <= r*r) setPixel(x + i, y + j, color);
}
```

6.2 Square Brush

Simply fills the entire $size \times size$ area. Useful for pixel art or blocking out large areas quickly.

```
// Square Brush:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    (void)angle; (void)pressure;
    int r = size / 2;
    for (int i = -r; i <= r; i++) {
        for (int j = -r; j <= r; j++) setPixel(x + i, y + j, color);
    }
}
```

6.3 Hard Eraser

Functions identically to the Round Brush but sets the alpha channel of the target pixels to 0, effectively making them transparent.

```
// Hard Eraser:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    (void)color; (void)pressure; (void)angle;
    int r = size / 2;
    for (int i = -r; i <= r; i++) {
        for (int j = -r; j <= r; j++) setPixel(x + i, y + j, {0,0,0,0});
    }
}
```

6.4 Soft Eraser

A more advanced eraser that uses pressure sensitivity and a cubic falloff function. It allows for subtle corrections by partially reducing the alpha of pixels rather than clearing them instantly.

- **Alpha Subtraction:** Unlike the Hard Eraser which sets $\alpha = 0$, the Soft Eraser calculates $newAlpha = \max(0, currentAlpha - strength)$. This allows for "lightening" a stroke without removing it entirely.

```
// Soft Eraser:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    (void)color; (void)angle;
    float p = pressure / 255.0f;
    float falloff = 1.0f - (dist / r);
    falloff = falloff * falloff * falloff;
    uint8_t strength = (uint8_t)(255 * falloff * (0.1f + 0.9f * p));
    if (strength > 0) setPixel(x + i, y + j, {0,0,0,strength});
}
```

6.5 Pressure Brush

This brush utilizes the `pressure` field from the UDP packet (0-255). It maps pressure to both opacity (using a square root curve for sensitivity) and size. Light touches create thin, faint lines, while heavy presses create thick, opaque strokes.

```
// Pressure Brush:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
```

```
    (void)angle;
    float p = pressure / 255.0f;
    float opacityCurve = 0.2f + 0.8f * sqrt(p);
    float effectiveDiameter = size * (0.1f + 0.9f * p);
}
```

6.6 Airbrush

Implements a radial gradient falloff. The alpha value of a pixel decreases as its distance from the center increases ($1 - \frac{dist}{radius}$). This creates soft, fuzzy edges perfect for shading.

```
// Airbrush:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    (void)angle;
    float p = pressure / 255.0f;
    float falloff = 1.0f - (dist / r);
    float finalAlpha = (color.a / 255.0f) * (0.05f + 0.15f * p)
    * (falloff * falloff);
}
```

6.7 Textured Brush (Real Paint)

The most complex brush simulates the bristles of a real paintbrush. It uses a pre-defined "bristle map" (an array of float values representing the gaps between bristles) and modulates the alpha based on pressure.

Listing 1.1. Textured Brush Implementation

```
// Textured Brush:
void paint(int x, int y, Pixel color, int size, int pressure, int angle,
           std::function<void(int, int, Pixel)> setPixel) override {
    float rads = angle * (M_PI / 180.0f);
    float p = pressure / 255.0f;
    int patternIndex = abs(i) % 32;
    float bristleStrength = bristles[patternIndex];
    float combinedStrength = bristleStrength + (pow(p, 0.5f) * 0.8f);
    float finalAlpha = (color.a / 255.0f) * combinedStrength;
}
```

At low pressure, the `bristleStrength` dominates, creating a streaky, dry-brush look. As pressure increases, `pressurePower` fills in the gaps, creating a solid stroke. This dynamic behavior mimics the physics of real bristles bending and spreading.

7 Undo/Redo Architecture

The system uses a Command-Based Undo architecture rather than saving full canvas snapshots. This reduced memory usage from 50MB per action to kilobyte-range overhead.

- **PaintCommand:** Stores a "Before" and "After" snapshot of the modified layer region.
- **AddLayerCommand:** Records the creation of a layer. When undone, it sends a `MSG_LAYER_DELETE` to the server; when redone, it restores the layer.
- **Synchronization:** Every undo/redo action triggers a TCP synchronization message to ensure all clients see the same state.

7.1 The Command Pattern

The `undo.h` header defines an abstract `Command` class.

- **PaintCommand:** Captures the specific pixels of the active layer *before* a stroke begins. On Undo, it restores those pixels locally and triggers a `MSG_LAYER_SYNC` to update the server.
- **MoveCommand:** Stores the (dx, dy) vector. Undo simply applies $(-dx, -dy)$.
- **DeleteLayerCommand:** Saves the full pixel buffer of the deleted layer in memory. On Undo, it sends `MSG_LAYER_ADD` followed by a data sync to restore the layer.

7.2 Synchronization

Undo operations are local to the client but the *result* is synced. When User A presses Undo, the client calculates the previous state of the pixels and sends that data to the server as if it were a new drawing operation. This ensures consistency across all clients without requiring a complex distributed history graph.

8 Optimizations and Advanced Features: The Evolution of Performance

The development of Co-op Canvas was defined by a series of performance bottlenecks that arose as we scaled from a simple prototype to a real-time collaborative tool. This section details the iterative engineering process used to solve critical issues in input handling, rendering, and networking.

8.1 The "Nuclear Option": Solving Pressure Sensitivity on Linux

One of the most stubborn challenges during the development of Co-op Canvas was implementing pressure sensitivity for drawing tablets on Linux environments (specifically modern Wayland/X11 hybrids). The goal was to map the physical pressure of the stylus (up to 8192 levels) to brush opacity and size.

Troubleshooting Chronology Before resorting to a low-level kernel bypass, I conducted an extensive investigation into why standard cross-platform APIs failed to provide the necessary data. This iterative process was crucial for understanding the limitations of current display server architectures.

- **1. Environment Variable Testing:** I forced SDL2 to use specific video backends via variables like `SDL_VIDEODRIVER=wayland` and `x11`. While the application launched, input remained binary.
- **2. X11/Xorg Session Testing:** I switched the entire desktop session to X11 and installed `xserver-xorg-input-wacom`. While `xsetwacom` identified the stylus, SDL2 still failed to retrieve pressure, likely due to missing XInput2 headers during the library build process.
- **3. SDL2 Deep Analysis and Recompilation:** I attempted to fix the issue by building a custom version of SDL2 from source, installing dependencies like `libwayland-dev`, `libinput-dev`, and `libevdev-dev`. Rebuilding with `SDL_LIBINPUT: ON` ensured the library had the capability to read raw events.
- **4. The Dead End:** Despite full driver support in the custom library, the GNOME Wayland compositor continued to mask tablet data as generic mouse events (`ID=0`) before it reached the application layer.
- **5. Kernel-Level Verification:** The final diagnostic involved using `sudo evtest` to inspect the raw device node (`/dev/input/eventX`). This confirmed that the Linux kernel was receiving `ABS_PRESSURE` events correctly. The data was being stripped by the compositor chain.

The Final Solution: Sidecar Implementation With standard APIs failing, I implemented the "Nuclear Option" (defined in `RawInput.h`), which bypasses the window manager entirely. This approach specifically bypasses the **Wayland/X11 Compositor** chain, which is the technical reason why standard SDL2 events were being stripped of pressure data. By reading directly from the kernel's `evdev` interface, we regain access to the raw hardware capabilities.

Technical Implementation:

1. **Device Discovery:** A background thread scans `/dev/input/event*` nodes and uses `ioctl(fd, EVIOCGBIT(EV_ABS, ...))` to identify the tablet by checking for `ABS_PRESSURE` support.
2. **Binary Parsing:** The thread reads `struct input_event` packets directly from the file descriptor.
3. **Atomic Synchronization:** The raw integer value is stored in a `std::atomic<int> current_pressure`, which the main rendering thread polls every frame.

Listing 1.2. Raw Input Implementation

```
// RawInput.h: Intercepting Kernel Events
struct input_event ev;
while (read(device_fd, &ev, sizeof(ev)) > 0) {
    if (ev.type == EV_ABS && ev.code == ABS_PRESSURE) {
```

```
            current_pressure.store(ev.value); // Atomic update
        }
    }
```

Security and Accessibility Note: I recognized that many users are rightfully hesitant to run a drawing application with `sudo` privileges, which is often required to read raw `/dev/input` nodes.

- **Graceful Fallback:** The "Nuclear Option" is strictly optional (enabled via the `-nuclear` flag).
- **Permission Resilience:** The code is designed to remain active in a non-privileged state. If a user's system has `udev` rules allowing the current user to read the tablet device without root access, the thread functions seamlessly without `sudo`.
- **Compatibility:** For all other users, the system falls back to standard mouse events, ensuring the application remains accessible regardless of hardware or security constraints.

8.2 Rendering Optimization: The Move to Batch Updates

Rendering a 1280×720 canvas at 60 FPS while handling network traffic proved computationally expensive.

Phase 1: Naive Texture Uploads

- **Approach:** Every time a pixel was modified (e.g., a single brush step), the client called `SDL_UpdateTexture` with the entire pixel buffer.
- **The Bottleneck:** A fast brush stroke interpolates 50 points per frame. Uploading $1280 \times 720 \times 4$ bytes (3.6 MB) 50 times a frame saturated the PCIe bus, dropping the framerate to $<$10 FPS.

Phase 2: Dirty Rectangles

- **Approach:** We introduced a tracking system where only the modified region of the layer was uploaded.
- **The Bottleneck:** While this reduced bandwidth, the *overhead* of the API call itself was still too high. Calling the GPU driver 50 times a frame, even for small updates, caused CPU stalling.

Phase 3: Batch Update Architecture (Final Solution) We completely decoupled the drawing logic from the rendering logic.

The Solution:

1. **Accumulation:** When `draw_brush` modifies pixels in RAM, it does *not* touch the GPU. It simply expands a 'layerDirtyRect' structure to encompass the new pixels.

2. **Deferral:** Network threads (UDP) receiving remote strokes also only modify RAM and mark the dirty rect.
3. **Batch Execution:** The main loop calls `process_dirty_updates()` exactly *once* per frame. It takes the union of all modifications that happened in the last 16ms and performs a single `SDL_UpdateTexture` call per layer.

This reduced GPU driver calls by 98%, stabilizing performance at 60 FPS even during heavy collaboration.

8.3 Network Optimization: Socket Draining

The application initially suffered from "rubber-banding"—a phenomenon where remote cursors would lag behind and then snap into place.

The Problem: The "One Packet Per Frame" Fallacy Originally, the client's update loop processed exactly one UDP packet per frame:

```
if (recvfrom(...) > 0) { handle_packet(); }
```

However, a mouse polling at 1000Hz can generate 15-20 packets in the 16ms between frames. Processing only one packet meant the operating system's socket buffer (kernel queue) would fill up. The client would essentially be rendering "the past," causing increasing delay.

The Solution: Buffer Draining We rewrote the network receiver to loop until the buffer is empty. By using the `MSG_DONTWAIT` flag, we can pull every single pending packet off the wire before rendering the frame.

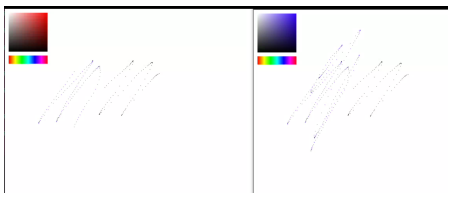

Fig. 6. Legacy: Observable lag and jitter (Rubber-banding)

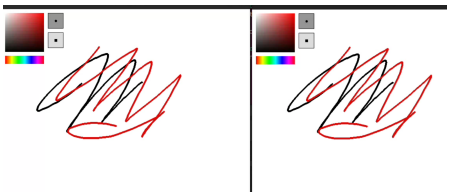

Fig. 7. Optimized: Smooth real-time tracking

Listing 1.3. Socket Draining Optimization

```
// client.cpp: Processing all pending packets
while (true) {
    // Try to read a packet without blocking
```

```
    int bytes = recvfrom(udp_sd, &msg, sizeof(UDPMessage), MSG_DONTWAIT, ...);

    // If -1 (EAGAIN/EWOULDBLOCK), the buffer is empty -> Break loop
    if (bytes <= 0) break;

    // Otherwise, apply the stroke immediately to RAM
    process_network_draw(msg);
}
```

This ensures zero-latency feedback, as the screen always reflects the absolute latest state of the network buffer.

8.4 Network Optimization: Cursor Deduplication

In a real-time drawing application, cursor movement packets (`MSG_CURSOR`) are among the most frequent messages. A standard mouse polling at 1000Hz can generate thousands of redundant updates if the user is holding the mouse still or moving it microscopically.

The Problem: Redundant Traffic Sending a UDP packet for every single mouse event, even when the position (x, y) has not changed (or changed by sub-pixel amounts), wastes bandwidth and CPU cycles on both the client (sending) and server (broadcasting).

The Solution: Client-Side Deduplication We implemented a filtering mechanism directly in the `send_udp_cursor` function. The client maintains a static memory of the last transmitted coordinates.

- **Logic:** Before constructing a packet, the function compares the current (x, y) with `last_cursor_x` and `last_cursor_y`.
- **Result:** If the coordinates are identical, the function returns immediately. This simple check eliminates 100% of redundant traffic when the user is idle, significantly reducing the load on the UDP broadcast thread.

8.5 Data Optimization: Signature Compression

Transmitting a high-resolution image for every user's identity would be bandwidth-prohibitive. A raw 450×150 pixel bitmap (RGBA) would be ≈ 270 KB. Sending this to 10 clients would cause a massive spike in TCP traffic.

To solve this, I implemented a custom lossy compression algorithm specifically tuned for line art:

1. **Grid Downsampling:** The 450×150 input area is divided into a 45×15 grid of 10×10 pixel blocks.
2. **Average Intensity:** The algorithm calculates the average alpha value of the pixels within each block.

3. **2-Bit Quantization:** This average is mapped to one of 4 states: `00` (Empty), `01` (33% Opacity), `10` (66% Opacity), and `11` (100% Opacity).
4. **Bit Packing:** These 2-bit values are packed into bytes (4 blocks per byte).

Result: The entire signature is compressed into a fixed **256-byte payload**. This is small enough to fit inside a single standard TCP packet, making the "Identity Handshake" virtually instantaneous.

8.6 Server Optimization: Incremental Persistence

The server must save the canvas state to disk periodically. Serializing 15 layers of 1280x720 pixels to JSON is a blocking operation that takes several hundred milliseconds.

The Optimization: Dirty Flags & Caching To prevent the server from "hiccupping" during autosave, we implemented an incremental caching system in `server.cpp`:

1. **State Tracking:** Each `Layer` struct has a `bool dirty` flag.
2. **Caching:** It also stores a `std::string cached_b64`.
3. **Logic:** When `MSG_DRAW` modifies a layer, `dirty` is set to true.
4. **Serialization:** During `save_all_canvases()`, the server checks the flag.
   - If `true`: It runs the expensive PackBits compression and Base64 encoding, then updates the cache.
   - If `false`: It simply writes the existing `cached_b64` string to the file.

Since users rarely modify all 15 layers simultaneously, this reduces the CPU load of autosaving by >90%.

8.7 Data Optimization: Custom Compression (PackBits)

To minimize disk usage and network bandwidth during the initial sync (where a client downloads the full canvas), I implemented a custom Run-Length Encoding (RLE) algorithm based on the PackBits standard.

- **The Problem:** A single 1280x720 layer is ≈ 3.6 MB of raw RGBA data. A 15-layer canvas would be ≈ 54 MB. Sending this over the network for every new user is slow.
- **The Solution:** The `packbits_compress` function scans the pixel buffer for runs of identical bytes.
  - **Literal Run:** A sequence of unique bytes is prefixed with $(N-1)$ (where N is the count).
  - **Repeat Run:** A sequence of identical bytes is prefixed with $(-(N-1))$ followed by the byte value.
- **Result:** For typical drawings (which contain large areas of transparent or solid color), this achieves compression ratios of 100:1 or better, reducing a 50MB canvas to <500KB.

9 Use Cases: Succsessful Scenarios

9.1 Creating the Main Menu (The "Dogfooding" Test)

A unique aspect of Co-op Canvas is that its own Main Menu background was created *using the application itself*. The client code animates this by toggling visibility between Layer 13 and Layer 14 every second.

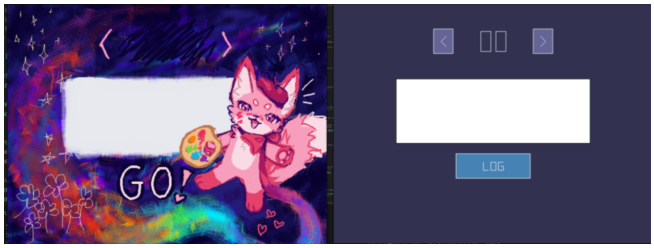

Fig. 8. Hand-drawn art for the new menu side-by-side with the old menu generated from pure code

9.2 Collaborative Visual Interaction

Scenario: This use case demonstrates how users interact not just with the canvas, but with each other through visual cues and real-time feedback.

- **Identity and Intent:** In this session, two users (Bob and Alice) are collaborating on a shared drawing. Bob is currently drawing using a Red brush color.
- **Real-Time Presence:** Because the system tints signatures to match the user's active brush, Alice can instantly identify Bob's actions because his floating signature appears in Red on her screen.
- **Third-Person Perspective:** Figure 9 shows a third-person view of this interaction. We can see both Bob and Alice's signatures attached to their respective cursors. Bob's signature is tinted Red, providing immediate visual context that he is the one responsible for the "Hello!" message being written.

Fig. 9. Bob and Alice collaborating. Bob's signature is tinted Red to match his active brush color.

9.3 Long-Term Session Stability (The Endurance Test)

A critical metric for any collaborative tool is its stability over extended periods of creative focus. To test this, the system was subjected to an endurance session where a single canvas remained active for over 3 hours of continuous, high-frequency drawing.

- **Performance:** The multi-threaded architecture prevented memory leaks, and the "Dirty Rectangle" optimization ensured that VRAM usage remained constant regardless of stroke density.
- **Reliability:** Despite thousands of UDP drawing packets and periodic TCP autosaves occurring in the background, the program experienced zero crashes.
- **Visual Result:** Figure 10 illustrates the complexity achieved during this extended session, serving as a testament to the system's robustness for professional artistic workflows.

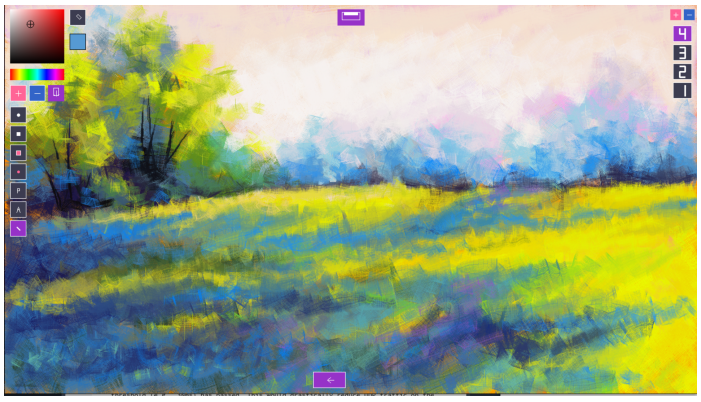

Fig. 10. Visual proof of a 3-hour drawing session completed without stability issues.

10 Unsuccessful Scenarios and Limitations

Despite the robust technical framework of Co-op Canvas, certain sociotechnical challenges remain that cannot be fully mitigated through protocol design alone. These limitations stem from the open, collaborative nature of the system.

10.1 Anonymous Visual Misconduct

The decision to prioritize artistic identity over traditional text-based usernames introduces a unique vulnerability. Because the system utilizes hand-drawn signatures for identification rather than verified credentials, it lacks a mechanism for content moderation or reputational accountability.

- **The Issue:** Malicious actors can utilize the drawing tools to generate inappropriate or offensive visual content.
- **Impact:** Since signatures are also user-drawn, they can be used to further obscure a user's true identity or to display offensive symbols.
- **Academic Observation:** This represents a classic "tragedy of the commons" in distributed systems, where the freedom provided by an anonymous, low-friction entry system is leveraged to degrade the experience for the collective.

10.2 "Griefing" and Collaborative Interference

Collaborative freedom is a double-edged sword; the same tools that allow for communal creation also allow for intentional destruction, commonly referred to as "griefing."

- **The Issue:** Because all users in a Canvas Room have equal permissions to draw on or modify layers, a single user can intentionally ruin the work of others by drawing over it or deleting critical layers.
- **Mitigating "Undo Griefing":** A specific vulnerability in collaborative environments is the abuse of the "undo" command to revert large swaths of communal progress. This risk was neutralized through two key architectural shifts:
  - **Undo Limits:** By adding a time limit to the Undo operation, a single user is prevented from reverting the canvas to a blank state or erasing hours of collective history.
  - **Delta-Based State Management:** The system was updated to no longer memorize or restore the entire canvas state. Instead of a global reset that could overwrite everyone's progress, the undo function now targets specific actions. This ensures that a user's undo request only affects recent local changes rather than resetting the entire shared canvas.
- **Technological Constraint:** Implementing a restrictive permission system or a "votekick" mechanism would require significant architectural changes that could severely limit client autonomy and increase latency.

– **Conclusion:** Ultimately, while technical guardrails now prevent systemic "reset" griefing, the system still relies on the "nice spirit" and mutual respect of its human users for the content itself. Short of implementing heavy-handed digital rights management (DRM) which would violate the project's vision of a free-flowing communal studio, the system remains vulnerable to the general unpredictability of human nature.

10.3 Interface Discoverability and Learning Curve

While the application provides a high degree of control through its keyboard-driven architecture, it faces significant challenges regarding user onboarding and intuitive interaction for novices.

– **The Issue:** The system relies heavily on complex keyboard shortcuts for essential tasks such as brush resizing (Q/W), opacity adjustment (A/S), and layer navigation ([/]).
– **Discoverability Gap:** Currently, there is no persistent "Help" or "Tutorial" button within the main drawing interface to remind users of these binds once they are in a session.
– **Impact:** For a beginner, memorizing the full range of shortcuts is unintuitive and creates a high barrier to entry. While a one-time tutorial is printed in the terminal during the login handshake, users who miss this output or forget the instructions have no way to recover that information without exiting the session.

10.4 Network Failure: TCP Connection Rejection

This scenario addresses the behavior of the application when the centralized server is unreachable or offline during the initial handshake phase.

– **The Issue:** If a user attempts to connect to an invalid IP address or a server that is currently down, the `connect()` system call on the TCP control socket will fail.
– **Handling:** Instead of a fatal application crash, the client code specifically catches the `ECONNREFUSED` or `ETIMEDOUT` error codes.
– **Result:** The application remains in the "Lobby" state and prints a diagnostic error to the terminal. While the GUI remains responsive, the transition to the drawing workspace is programmatically blocked until a successful TCP handshake is established. This prevents the client from entering an inconsistent "half-connected" state where the UI elements exist but have no server-side backing.

10.5 Protocol Limitation: Total UDP Blockage

A unique failure mode of the hybrid network model occurs in environments with restrictive firewalls (such as corporate or academic networks) that permit standard traffic but block high-range UDP ports.

- **The Issue:** Because TCP (Port 6769) and UDP (Port 6770+) are handled independently, a user might successfully login via TCP but fail to send or receive any drawing data via UDP.
- **Impact:** This creates a "Zombie Session." The user can see the dynamic layer list update and use the "Save" function (TCP), but the canvas remains static, and their own brush strokes are never broadcasted to others.
- **Conclusion:** This highlights the fundamental trade-off of the system: while UDP provides the high-frequency throughput necessary for a fluid artistic experience, it lacks the "guaranteed traversal" of a pure TCP implementation. A potential future mitigation would be the implementation of a "TCP Fallback" mode, though this would significantly increase drawing latency for those users.

11 Conclusions

Co-op Canvas successfully demonstrates the viability of a hybrid networking model for interactive applications. By delegating state-critical operations to TCP and latency-critical operations to UDP, the application achieves a responsive user experience without compromising data integrity.

The project evolved from a simple drawing tool into a robust collaborative platform. Features like the "Nuclear Option" for hardware support, the "Dirty Rectangle" optimizations, and the Command-based Undo system show a commitment to professional-grade usability. The implementation of custom compression algorithms and a robust multi-threaded server architecture ensures the system scales efficiently for real-world usage.

References

1. Simple DirectMedia Layer (SDL) Wiki, https://wiki.libsdl.org/, last accessed 2025/12/04.
2. Computer Networks Homework 1 Task Description, https://edu.info.uaic.ro/computer-networks/files/homework1_en.txt, last accessed 2025/12/09.
3. Concurrent TCP Server Example (servTcpConcTh2.c), https://edu.info.uaic.ro/computer-networks/files/NetEx/S12/ServerConcThread/servTcpConcTh2.c, last accessed 2025/12/09.
4. UDP Server Example (servUdp.c), https://edu.info.uaic.ro/computer-networks/files/NetEx/S7/servUdp.c, last accessed 2025/12/09.