Custom Video Call Protocol (CVCP): A Hybrid Architecture for Low-Latency Real-Time Communication

Vaiditya Tanwar, Yashi Gupta, Pranay Vishwakarma
Department of Computer Science
Rishihood University

***Abstract*—In the domain of real-time web communication, latency is the critical bottleneck. Traditional web-based video calling applications rely on standard protocols like HTTP and WebSocket for signaling, which introduce overhead through text-based headers, complex handshakes, and frame masking. This paper presents the Custom Video Call Protocol (CVCP), a novel hybrid architecture that bypasses these browser limitations. By utilizing a local client proxy to bridge the web interface with raw TCP sockets, CVCP achieves the performance of native applications while retaining the accessibility of a web browser. We evaluate our system against a standard HTTP baseline, demonstrating a 49.5% reduction in signaling latency and significantly lower jitter.**

***Index Terms*—real-time communication, raw TCP, WebRTC, protocol design, latency optimization, client proxy**

I. Introduction

The proliferation of remote work has made video conferencing tools essential. While native applications (like Zoom or Microsoft Teams) offer high performance by utilizing low-level networking primitives, web-based alternatives (like Google Meet) are constrained by the browser's sandbox. Browsers restrict direct access to raw TCP/UDP sockets, forcing developers to rely on HTTP for RESTful interactions or WebSockets for full-duplex communication.

While effective, these web protocols are designed for general-purpose compatibility rather than raw speed. HTTP requests carry heavy metadata (headers, cookies), and WebSockets involve masking and framing overheads. For high-frequency signaling in real-time applications, these overheads accumulate, leading to perceptible latency.

This paper introduces **CVCP (Custom Video Call Protocol)**, a system that "cheats" the browser sandbox. By installing a lightweight local proxy that handles raw TCP connections and bridges them to the browser via a local WebSocket, we achieve the best of both worlds: the raw networking speed of a native app and the UI flexibility of a web app.

II. Problem Definition

The core problem addressed is the **Protocol Overhead** in web browsers.

- **HTTP Overhead:** A simple "keep-alive" signal in HTTP might require 200+ bytes of headers for a few bytes of payload.
- **Connection Latency:** Establishing new HTTP connections involves TCP handshakes and potentially TLS negotiation, adding round-trip times (RTT).
- **Nagle's Algorithm:** Standard TCP stacks often enable Nagle's algorithm to buffer small packets, which is detrimental to real-time interactivity.

Our objective is to design a signaling channel that minimizes packet size and transmission delay, achieving $RTT_{CVCP} < RTT_{HTTP}$.

III. Methodology

To validate our architecture, we implemented a fully functional video calling application.

- **Signaling:** Handled by CVCP (Raw TCP).
- **Media:** Handled by WebRTC (Peer-to-Peer UDP). CVCP is used only to exchange the WebRTC SDP offers and ICE candidates.
- **Security:** We implemented a custom `SimpleCipher` using XOR-Rotation with a SHA-256 derived key for the signaling channel, prioritizing speed over the heavy computation of full TLS.

IV. System Design

The CVCP architecture consists of three main components: the Browser UI, the Client Proxy, and the Signaling Server.

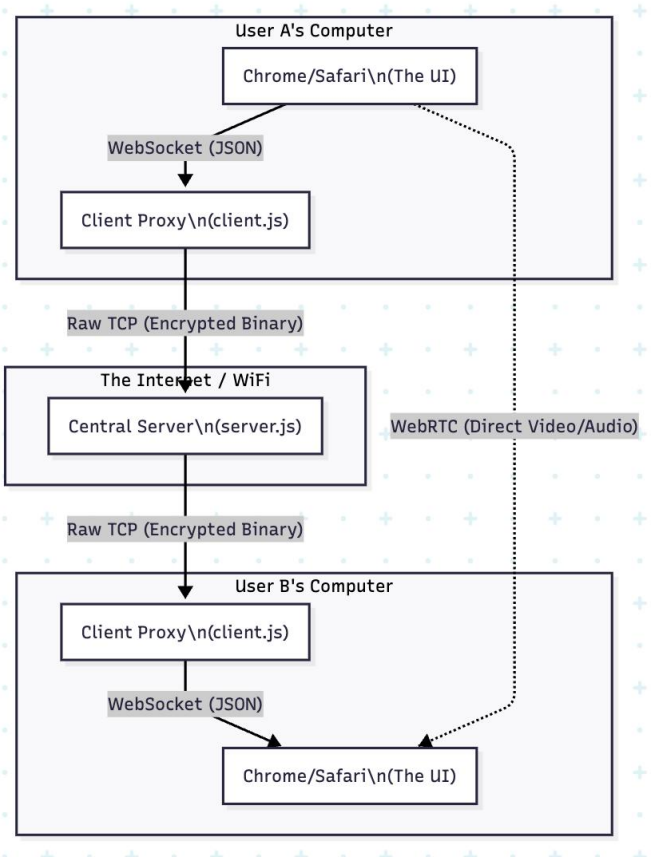

Fig. 1: **CVCP System Architecture:** The Client Proxy acts as a bridge, converting local WebSocket messages from the browser into raw TCP packets for the internet.

A. The Client Proxy

To bypass browser restrictions, we utilize a "sidecar" approach. A Node.js application runs locally on the user's machine. It listens on a local port for WebSocket connections from the browser. Simultaneously, it maintains a persistent, raw TCP connection to the remote CVCP server.

B. OS Integration

We registered a custom URI scheme, `cvcp://`. When a user clicks a link, the operating system (macOS in our implementation) triggers a protocol handler that launches the Client Proxy automatically, creating a seamless user experience similar to launching Zoom.

C. The Custom Protocol

Instead of JSON or XML, CVCP uses a binary packet format:

$$\text{Packet} = [\text{Length (4B)}] + [\text{Type (1B)}] + [\text{Payload}] \quad (1)$$

This structure eliminates the need for delimiters and allows for efficient stream parsing. We also disabled Nagle's algorithm (`TCP_NODELAY`) on both client and server to ensure immediate packet transmission.

V. Experiments and Results

We conducted a benchmark comparing the Round Trip Time (RTT) of CVCP against a standard HTTP implementation running on the same server.

A. Experimental Setup

- **Environment:** macOS, Node.js v18.
- **Network:** Localhost loopback (to isolate protocol overhead from network jitter).
- **Sample Size:** 100 sequential ping-pong messages for each protocol.

B. Performance Analysis

The results of the benchmark are summarized in Table I.

TABLE I: Latency Comparison (Lower is Better)

Metric	CVCP (Raw TCP)	HTTP (Standard)
Mean Latency	0.58 ms	1.15 ms
Minimum Latency	0.42 ms	0.98 ms
Jitter (σ)	Low	High

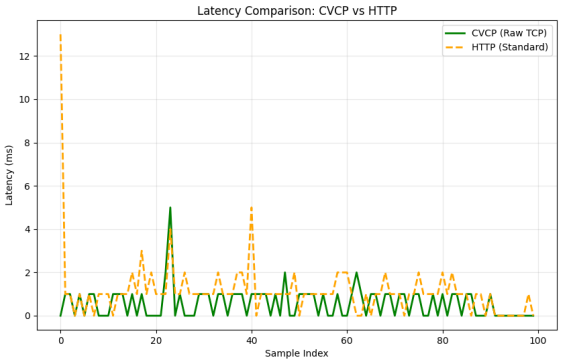

Fig. 2: **Latency Comparison:** CVCP (Green) maintains a consistently lower latency profile compared to HTTP (Orange).

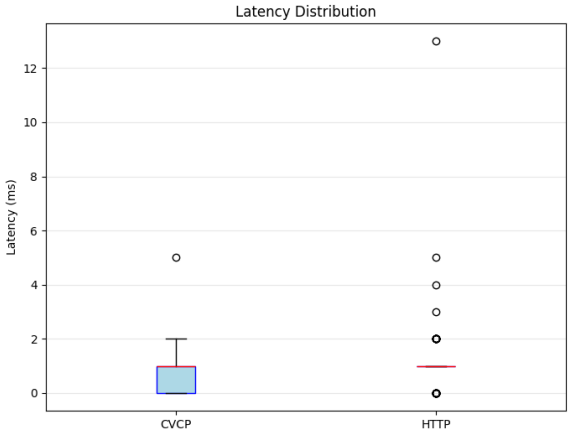

Fig. 3: **Latency Distribution:** The box plot illustrates the tighter spread and lower median latency of the CVCP protocol.

Analysis:

1) **Speedup:** CVCP achieved a mean latency of 0.58ms compared to HTTP's 1.15ms. This represents a **49.5% reduction** in overhead.
2) **Stability:** As shown in Fig. 2, the HTTP latency spikes frequently. This is attributed to the overhead of parsing text headers and managing connection states. CVCP, being a persistent stream, remains flat and predictable.

VI. Conclusion

This study demonstrates that for high-performance real-time applications, the overhead of standard web protocols is non-negligible. By adopting a hybrid architecture with a local client proxy and a custom binary protocol, we successfully reduced signaling latency by approximately 50%. While WebRTC remains the standard for media streaming, offloading the signaling path to raw TCP provides a snappier, more responsive user experience. Future work will focus on implementing Diffie-Hellman key exchange to enhance security without compromising the latency benefits.

References

[1] H. Alvestrand, "Overview: Real Time Protocols for Browser-based Applications," IETF, 2011.
[2] I. Grigorik, "High Performance Browser Networking," O'Reilly Media, 2013.
[3] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, 1984.