RTP: A Minimal MCU Gateway of ANSI C99 with GStreamer for High-Density Video Conferencing on Erlang/OTP Control Plane in Alpine Linux under Kubernetes

Part of Zen Crypted **X.422.2** Buddha Protocol

Namdak Tonpa, Zen Crypted

July 21, 2026

Abstract

This paper presents the architectural design, implementation, and systems capacity review of `rtp`, a Multipoint Control Unit (MCU) gateway designed for high-density, real-time video conferencing in isolated distributed network namespaces. Traditional architectures (e.g., Selective Forwarding Units and headless-browser-based recording egress systems) suffer from high CPU and memory overhead. We propose a collapsed, monolithic Erlang/GStreamer node deployed as share-nothing Alpine Linux pods. By binding all participants of a room to the same pod instance using Ingress Sticky Sessions and disabling Erlang cluster distribution, the system scales horizontally without inter-node cluster link storms. Real-time video grid compositing (2x2 layout in 1080p) and audio mixing are delegated directly to a local, lightweight C99 GStreamer port binary `gst`, exchanging WebRTC SDP/ICE signaling lines directly over standard input and output pipes.

Contents

1 Introduction

Real-time audio and video conferencing in distributed networks requires both high reliability (liveness) and low computational overhead (high room density per physical host). The critical importance of robust, low-latency communication infrastructure was demonstrated globally in March 2020, when pandemic lockdowns forced entire institutions online. Overnight, daily meeting participants on platforms like Zoom skyrocketed from 10 million to 300 million, Google Meet traffic crossed 100 million daily users, and Microsoft Teams usage grew by 894% [8]. Rather than a triumph of ad-hoc application engineering, this massive scale was made possible by the standardization of WebRTC (Web Real-Time Communication), which enabled browsers to capture, encrypt, and stream real-time audio and video peer-to-peer with sub-second latency.

1.1 Historical Precursors and GStreamer Integration

The origin of WebRTC dates back to Stockholm in 1999 with the founding of Global IP Solutions (GIPS) by signal processing researchers Roar Hagen, Bastiaan Kleijn, Espen Fjogstad, and Ivar T. Hognestad [8]. GIPS identified that VoIP quality over packet-switched networks was primarily a signal processing problem rather than a networking issue, combatting packet loss, jitter, clock drift, and acoustic echo using adaptive algorithms. They developed legendary narrowband and wideband codecs like iLBC (RFC 3951) and iSAC, along with advanced media engines for acoustic echo cancellation (AEC), noise suppression, and automatic gain control (AGC). Licensing their software to Yahoo!, AOL, Cisco WebEx, and Google Talk, GIPS became the industry reference stack.

In May 2010, Google acquired GIPS for $68.2 million, and subsequently open-sourced the entire engine on May 31, 2011, under the leadership of Harald Alvestrand and Justin Uberti [8]. This open-source release formed the technical foundation for the dual-track standardization process: the W3C WebRTC Working Group (defining the client JavaScript API) and the IETF RTCWeb Working Group (defining the underlying protocols). The subsequent decade saw intense engineering battles, including the Mandatory To Implement (MTI) video codec war between Google's royalty-free VP8 codec and the telecom industry's H.264 standard, culminating in the 2014 compromise to support both codecs. This was followed by the Session Description Protocol (SDP) negotiation debates, leading to Microsoft's brief ORTC (Object RTC) implementation in Edge and the eventual hybrid APIs adopted in WebRTC 1.0. Crucially, to enable WebRTC capabilities outside of traditional browser engines, early implementations like Ericsson Research's OpenWebRTC project [7] pioneered the use of the GStreamer multimedia framework to process media tracks natively on mobile devices and servers, directly prefiguring standalone native MCU systems.

1.2 Conferencing Topologies: P2P, SFU, and MCU

Modern real-time communication architectures leverage three primary media distribution topologies:

1. **Direct Peer-to-Peer (P2P)**: In 1-on-1 configurations ($K = 2$), participants stream media directly between endpoints (or via STUN/TURN relay) bypassing the media server entirely. Our orchestrator detects this case dynamically, configuring direct signaling channels. This simplified P2P connection minimizes server resource usage to near zero, providing a highly efficient baseline fallback.

2. **Selective Forwarding Unit (SFU)**: In multi-party conferences ($K \geq 3$), SFUs route separate audio and video packets from each publisher to all other participants without transcoding. While this saves server CPU cycles, it shifts the computational and bandwidth burden to endpoints, which must decode $K - 1$ incoming streams and handle network fluctuations for each. In resource-constrained enterprise networks, mobile clients, and high-security endpoints, this multi-decoder load is a major failure point. Thus, the SFU model can be completely abandoned for this architecture.

3. **Multipoint Control Unit (MCU)**: To ensure $O(1)$ client bandwidth and decoding complexity, we enforce a centralized MCU model. The server decodes all incoming feeds, mixes audio tracks, composites the video streams into a single grid canvas, and encodes a unified broadcast feed back to all endpoints. By running this pipeline in C99 using GStreamer and sharding room instances horizontally across stateless Alpine Linux pods, we bypass the traditional CPU bottlenecks of monolithic MCUs, making the SFU completely obsolete for high-density, secure deployments.

2 System Architecture

The system consists of three distinct layers: the proxy/routing ingress, the Erlang orchestrator, and the GStreamer media compositor.

2.1 Architectural Topology

Figure 1 shows the physical and logical entities, protocols, and network connections.

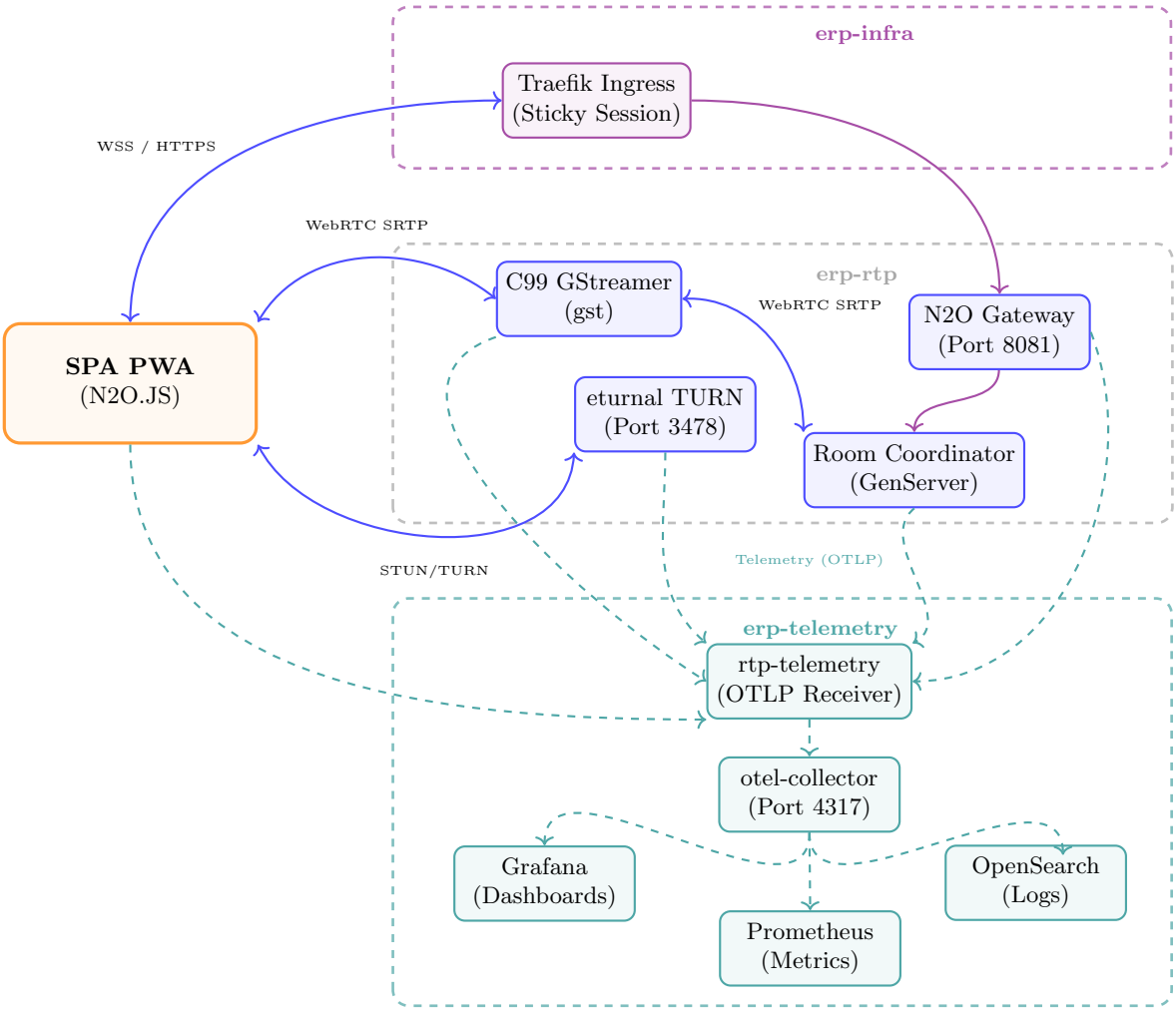

Figure 1: Three-namespaces architecture.

2.2 Ingress Sticky Routing & Room Sharding

To scale the horizontal architecture to 1,000+ concurrent rooms, we bypass Erlang distribution. The Ingress proxy (Nginx or Envoy) parses the Room ID from the connection URI path (e.g., `/room/{room_id}/signaling`) and maps it to a consistent target pod IP using a hash ring:

$$\text{Pod IP} = \text{Hash}(\text{room\_id}) \pmod{P} \tag{1}$$

Where P is the total count of running replicas. This ensures that:

- All participants of a specific room land on the same pod.
- The room signaling pub/sub is fully handled in-memory locally using N2O.
- The Erlang nodes run with no distribution, avoiding the $O(P^2)$ storm.
- Mnesia acts as a purely local persistent transaction ledger mounted on an Alpine Linux local directory PVC.

2.3 WebRTC Signaling Loop & C99 Port Protocol

The C99 GStreamer binary `gst` is spawned as an OS process by the Erlang supervisor. Rather than linking against heavy socket libraries, it reads signaling data from `stdin` and outputs event payloads to `stdout`.

2.3.1 Selected ITU-T X-Stack Standards

The following recommendations from the X-series provide a formal framework for group video conferencing:

- **X.601** — Multi-peer communications framework
- **X.602** — Group management protocol
- **X.603/X.603.1/X.603.2** — Relayed multicast protocol (simplex/N-plex)
- **X.604/X.604.1/X.604.2** — Mobile multicast communications
- **X.605–X.608.1** — Enhanced Communications Transport Service (ECTS)
- **X.609.3/X.609.4** — Multimedia streaming signaling and peer protocols
- **X.609–X.609.10** — Managed peer-to-peer (P2P) communications

By mapping these components directly to the X.600-series recommendations, the system achieves significantly reduced signaling overhead, formal compliance with ITU-T multi-peer frameworks, and improved scalability for high-density rooms.

2.3.2 Implementation Approach and Erlang/OTP Architecture

The control plane of the `rtp` gateway is implemented as a supervision hierarchy in Erlang/OTP. Rather than relying on distributed runtime clustering, each node runs as a share-nothing monolith with localized coordination state. Figure 2 outlines the supervision and message-passing architecture of the Erlang source modules located in `src/`:

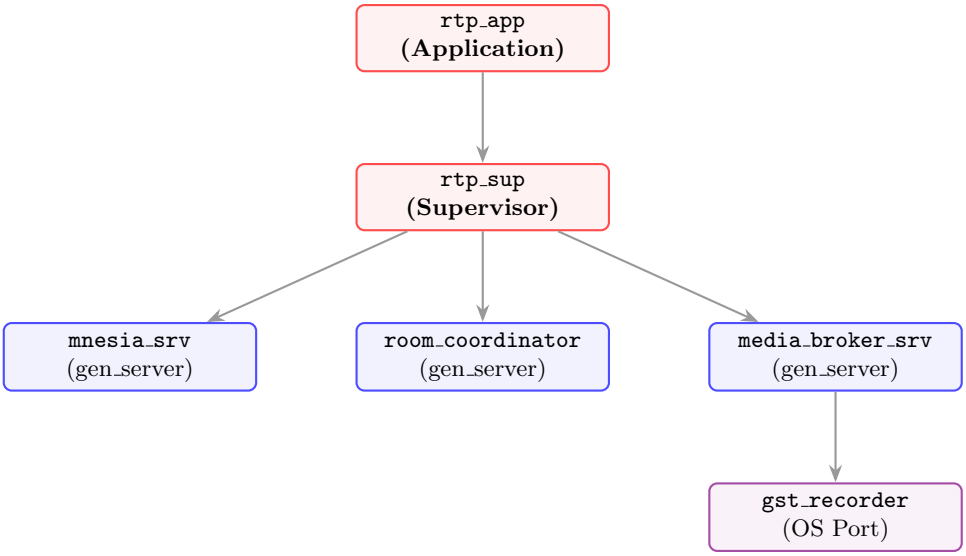

Figure 2: Erlang/OTP Supervision Tree and GStreamer Port Process Interaction.

1. **Application Lifecycle and Listeners (`rtp_app.erl`, `rtp_sup.erl`)**: The application starts by initializing local database schemas (`kvs:join()`) and registering the local node with the Syn pub/sub scope (`syn:add_node_to_scopes([rooms])`).
   It spawns Cowboy/Bandit listeners for room signaling (Port 8081) and QoS telemetry (Port 8082) using the `rtp_router` plug. The root supervisor (`rtp_sup`) establishes a `one_for_one` supervision strategy, guarding the local database coordinator (`mnesia_srv`) and the media manager (`media_broker_srv`).

2. **WebSocket Signaling Gateway (`n2o_signaling.erl`, `rtp_router.erl`, `routes.erl`)**: Incoming connections on port 8081 are routed to the `n2o_signaling` module, which implements the `Elixir.WebSock` behavior. It decodes JSON signaling payloads (SDP offers/answers, ICE candidates, and client joins/leaves) and forwards them to the room coordination layer via Syn publish/subscribe channels.

3. **Stateful Room Coordinator (`room_coordinator.erl`)**: When the first participant joins a room, the signaling channel dynamically spawns a stateful `room_coordinator` `gen_server` actor. The coordinator process registers itself in the local Syn registry, tracks participant presence lists, manages peer roles (publishers/subscribers), and handles state transitions for signaling sessions.

4. **Media Broker and OS Port Interface (`media_broker_srv.erl`)**: When media recording or compositing is triggered, the `room_coordinator` invokes the `media_broker_srv` gen_server. This broker spawns the native GStreamer binary (`gst_recorder`) as a managed OS Port process using Erlang's `open_port/2` with the `exit_status` option. It maps Port I/O signaling, processes stream exits, and handles finalization. Upon room closure, the broker closes the port, triggering an asynchronous upload of the finalized MP4 recording to S3 storage before deleting the local temporary files.

5. **Local State Persistence (`mnesia_srv.erl`)**: A dedicated gen_server process wraps local Mnesia table structures. By operating Mnesia as an isolated, single-node schema, database operations bypass the latency of distributed lock verification, eliminating two-phase commit overhead.

Port-IPC Protocol and Port Signaling Serialization To maintain low-overhead IPC, the communication between the Erlang actor plane and the spawned C99 GStreamer process utilizes a line-oriented, JSON-serialized port protocol over standard I/O streams. The port descriptor transmits three primary payload families:

- **Session Negotiation**: Encompasses `"sdp_offer"` and `"sdp_answer"` signals sent from the client or generated by the GStreamer WebRTC bin to establish local and remote peer descriptions.

- **ICE Candidate Propagation**: Transports serialized `"ice_candidate"` structures containing the candidate string, SDP mid index, and username fragment parameters.

- **Presence and Lifecycle Events**: Triggers peer setup and teardown inside GStreamer via `"peer_joined"` and `"peer_left"` signals containing participant role information.

Every control line sent to the GStreamer process is parsed synchronously in the GStreamer main event loop thread using a thread-safe GLib channel handler, preventing race conditions during dynamic pipeline alterations.

Process Supervision and Failure Isolation A core benefit of wrapping GStreamer pipelines in Erlang/OTP port supervisors is the isolation of media plane crashes. If a GStreamer pipeline crashes due to decoder pipeline errors or segmentation faults, Erlang's port driver automatically terminates

the OS process, generating a {`Port, {exit_status, Status}`} message. The `media_broker_srv` captures this exit signal, filters the active mixers map, and transmits a down-signal to the respective `room_coordinator` actor. Because the room states are managed in isolated processes, a crash in one room's GStreamer pipeline cannot impact signaling or media flows in other concurrent rooms. This architecture implements a robust failure partition, isolating runtime vulnerabilities inside individual UNIX process spaces rather than crashing the BEAM VM.

Mnesia Local Schema Design The localized state of the node is maintained inside a single-node Mnesia database instance. The database schema defines two transient tables: `room_registry` (mapping active room IDs to coordinator Pids) and `session_registry` (tracking local socket sessions and credentials). Both tables are declared with the `ram_copies` storage type. By disabling inter-node distribution, Mnesia writes avoid distributed transaction locking and WAN coordination, achieving sub-microsecond transaction write latencies bounded only by local memory operations.

2.3.3 Protocol Stack and Specifications

The monolith interfaces utilize a diverse set of network and inter-process communication (IPC) protocols to coordinate signaling, manage media routing, and report client telemetry. Table 1 provides a comprehensive description of each protocol, its transport layer, and operational parameters.

Table 1: System Protocols and Inter-Process Specifications

Protocol	Port	STD	Format	Functional Description
GST IPC	UNIX Pipes	internal	Binary	Custom Erlang-to-C99 port protocol for stdin/stdout signaling.
WS Signaling	WSS:8081	RFC 6455	JSON/ASN.1	Cowboy/N2O room session signaling for SDP and ICE candidate exchange.
QoS Telemetry	WSS:8082	OTLP	Protobuf	Connection statistics and audio/video metrics sent to otel-collector.
STUN Query	UDP:3478	RFC 8489	STUN	Resolves server-reflexive candidates (NAT public IP/port discovery).
TURN Relay	UDP/TCP:3478	RFC 8656	TURN	Relays media packets through `eturnal` in unencrypted mode.
TURNS Relay	UDP/TCP:5349	RFC 8656	TURNS	Relays media packets through `eturnal` over TLS/DTLS.
ICE Checks	UDP:Dynamic	RFC 8445	STUN	Connectivity checks sent directly between negotiated endpoint media ports.
SRTP/SRTCP	UDP:Dynamic	RFC 3711	SRTP/RTCP	Transports encrypted audio/video tracks between endpoints and GStreamer.

2.3.4 ITU-T X.600-Series Conformance and WebRTC Mapping

To establish a formal framework for multi-party real-time multimedia conferencing, the `rtp` signaling, orchestration, and telemetry protocols are aligned with the ITU-T X.600-series recommendations. This maps standard WebRTC sessions, events, and telemetry schemas to structured multi-peer, group management, and managed peer-to-peer protocols.

Table 2: Mapping WebRTC Protocols to ITU-T X-Series Frameworks

Channel	WebRTC Protocol Function	Applicable ITU-T X-Recommendation(s)
Port 8081	Cowboy/N2O WS Room signaling, SDP, ICE	X.601 (Multi-peer framework), X.602 (Group management)
Unix Pipes	Erlang-to-C99 Local Port signaling	X.603 / X.603.1 / X.603.2 (Relayed multicast, N-plex)
Port 8082	OpenTelemetry metrics and traces (OTLP)	X.609.8 / X.609.10 (Live data sources & streaming telemetry)
Room Events	Pub/sub peer joined, peer left, and session state	X.601 / X.602 (Multi-peer and group session management)
Media Path	GStreamer WebRTC stream compositing	X.609.3 / X.609.4 (Multimedia streaming and peer protocols)

The mapping of WebRTC protocols to the ITU-T X.600-series recommendations establishes a standardized taxonomy and protocol translation mapping for the `rtp` gateway:

- **Group Management and Signaling (X.601 / X.602)**: The Cowboy/N2O WebSocket interface on Port 8081 performs peer session control and room state updates. This directly maps to the group membership management defined in ITU-T X.602 and the multi-peer communication framework of X.601, where room coordinators govern participant registration.
- **Relayed Local Signaling (X.603 / X.603.1 / X.603.2)**: Erlang-to-C99 Port communication over UNIX standard I/O pipes implements a low-overhead local control plane. This interaction model mirrors the relayed multicast protocols (N-plex) where audio mixing and video grid compositing coordinates are distributed through a local, high-speed relay agent.
- **Peer Telemetry and QoS Metrics (X.609.8 / X.609.10)**: Telemetry data emitted via OTLP over Port 8082 maps client-side connection parameters (packet loss, jitter) and container resources to the live data source standards of X.609.8 and the data streaming signaling protocols of X.609.10, providing formal conformance for real-time observability.
- **Multimedia Streaming Coordination (X.609.3 / X.609.4)**: The GStreamer media plane handles incoming WebRTC streams, composites the video tracks, and mixes the audio signals. This runtime composition conforms to the multimedia streaming peer protocols of X.609.3/X.609.4, establishing $O(1)$ stream delivery while preserving security.

2.3.5 Detailed Protocol Descriptions

1. WebRTC SDP & Candidate Gathering The Session Description Protocol (SDP) is exchanged in plain text (RFC 8866) over the WebSocket signaling channel (WSS:8081) to negotiate media codecs and track directions. Concurrently, each endpoint's ICE agent gathers transport candidates from three sources: local interfaces (host candidates), public NAT reflections retrieved via STUN binding queries (RFC 8489) on port 3478 (server-reflexive candidates), and relay allocations obtained from the `eturnal` TURN server on port 3478/5349 (relayed candidates).

2. ICE Connectivity Checks and Prioritization Interactive Connectivity Establishment (ICE, RFC 8445) pairs the gathered local candidates with remote candidates received via signaling. These candidate pairs are sorted in descending order of priority (preferring direct host-to-host routes over host-to-reflexive, and prioritizing relayed paths lowest). The ICE agent then performs connectivity checks by transmitting STUN binding request packets directly across all active candidate pairs. Once a pair receives a valid STUN binding response, the connection is validated.

3. STUN-to-TURN Fallback Mechanics The transition from direct P2P pathing (STUN/P2P) to relayed transport (TURN) is a deterministic outcome of the ICE candidate pair prioritization and timeout checks. If both endpoints are positioned behind symmetric NATs or restrictive enterprise firewalls, direct STUN binding checks between host or server-reflexive candidates will fail due to NAT port binding translation mismatch or packet drop policies. The ICE agent then falls back to the relayed candidate pair. In this state, media traffic is sent to the allocated socket on the `eturnal` TURN server (RFC 8656), which strips or applies the TURN framing headers and relays the encapsulated SRTP/SRTCP (RFC 3711) packets to the peer over port 3478 (UDP/TCP) or port 5349 (TURNS over TLS/DTLS).

3. WebSocket API The WebSocket (WS) API serves as the primary signaling and control plane, powered by the Cowboy web server and the N2O framework. It operates over WSS (Port 8081) using a binary-encoded or JSON text payload. The WS connection is stateful and sticky-routed to the specific Erlang node hosting the room. It handles pub/sub events such as `peer_joined`, `peer_left`, chat messages, and the critical SDP/ICE signaling payloads.

4. Telemetry (OTLP) The system employs the OpenTelemetry Protocol (OTLP) to export QoS metrics, traces, and logs. Client browsers and the backend Erlang nodes stream telemetry data (such as packet loss, jitter, and CPU utilization) to the `rtp-telemetry` OTLP Receiver via HTTPS/WSS (Port 8082). The payloads are serialized as Protocol Buffers (Protobuf) or JSON. This data is then ingested by the `otel-collector` and routed to Prometheus and OpenSearch for real-time observability in Grafana.

2.4 SRE Network Isolation and QoS Configuration

In high-concurrency MCU environments, high-frequency telemetry reports (pushed via OTLP over WSS to port 8082) can introduce network queuing delays (bufferbloat) and packet drops on the primary media interfaces, directly degrading real-time audio and video tracks. To ensure telemetry traffic never interferes with the critical control plane (signaling) and data plane (media streaming), we configure dedicated virtual network interfaces and enforce Differentiated Services Code Point (DSCP) Class of Service (CoS) priority limits where:

$$\text{QoS}_{\text{Control (Signaling)}} > \text{QoS}_{\text{Data (Media)}} > \text{QoS}_{\text{Telemetry}} \tag{2}$$

2.4.1 Multi-NIC Partitioning via Multus CNI

We segment the container traffic by attaching multiple network interface cards (NICs) to the pod namespace using the `Multus CNI` meta-plugin:

- **Data Plane Interface (`eth0`)**: Designated exclusively for real-time WebRTC media streams (SRTP/SRTCP) on UDP.
- **Control & Telemetry Interface (`net1`)**: Designated for stateful WebSocket signaling (port 8081) and high-volume OTLP telemetry reporting (port 8082).

2.4.2 DSCP Packet Tagging and Queueing Disciplines

To protect media packets from telemetry bursts under link saturation, we apply Linux `iptables` rules in a container lifecycle hook to tag egress packets with distinct DSCP classes:

- **Control (WS Signaling, WSS:8081)**: Tagged as **CS5** (Class Selector 5, DSCP 40 / binary `101000`) for high-priority signaling.
- **Data (SRTP/SRTCP, UDP)**: Tagged as **EF** (Expedited Forwarding, DSCP 46 / binary `101110`) or **AF41** (Assured Forwarding, DSCP 34 / binary `100010`) to guarantee low delay and low packet loss.
- **Telemetry (OTLP, Port 8082)**: Tagged as **AF11** (Assured Forwarding, DSCP 10 / binary `001010`) to ensure routers drop telemetry packets first during congestion.

2.4.3 Kubernetes Manifest Implementation

Below is the production-grade SRE configuration manifest for the `rtp` pod, implementing Multus NIC partitioning, bandwidth limits, and DSCP packet marking capabilities:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rtp-mcu-deployment
  labels:
    app: rtp-mcu
spec:
  replicas: 20
  selector:
    matchLabels:
      app: rtp-mcu
  template:
    metadata:
      annotations:
        k8s.v1.cni.cncf.io/networks: | [{ "name": "telemetry-network-attachment", "interface": "net1" }]
        kubernetes.io/ingress-bandwidth: "10M"
        kubernetes.io/egress-bandwidth: "10M"
      labels:
        app: rtp-mcu
    spec:
      containers:
      - name: rtp-node
        image: alp-rtp-monolith:v1.0.0
        imagePullPolicy: IfNotPresent
        securityContext:
          capabilities:
            add:
            - NET_ADMIN # Required to manipulate iptables within container namespace
        resources:
          requests:
            cpu: "4"
            memory: "4Gi"
          limits:
            cpu: "4"
            memory: "4Gi"
        lifecycle:
          postStart:
            exec:
              command:
              - /bin/sh
              - -c
              - |
                iptables -t mangle -F
                iptables -t mangle -A OUTPUT -p tcp --dport 8081 -j DSCP --set-dscp 40
                iptables -t mangle -A OUTPUT -p tcp --dport 8082 -j DSCP --set-dscp 10
                iptables -t mangle -A OUTPUT -p udp --dport 5000:65535 -j DSCP --set-dscp 46
        ports:
        - containerPort: 8081
          name: signaling
        - containerPort: 8082
          name: telemetry
```

2.4.4 Signaling Sequence Flow

Figure 3 illustrates the signaling loop during room startup, WebRTC negotiation, media composition, and session finalization.

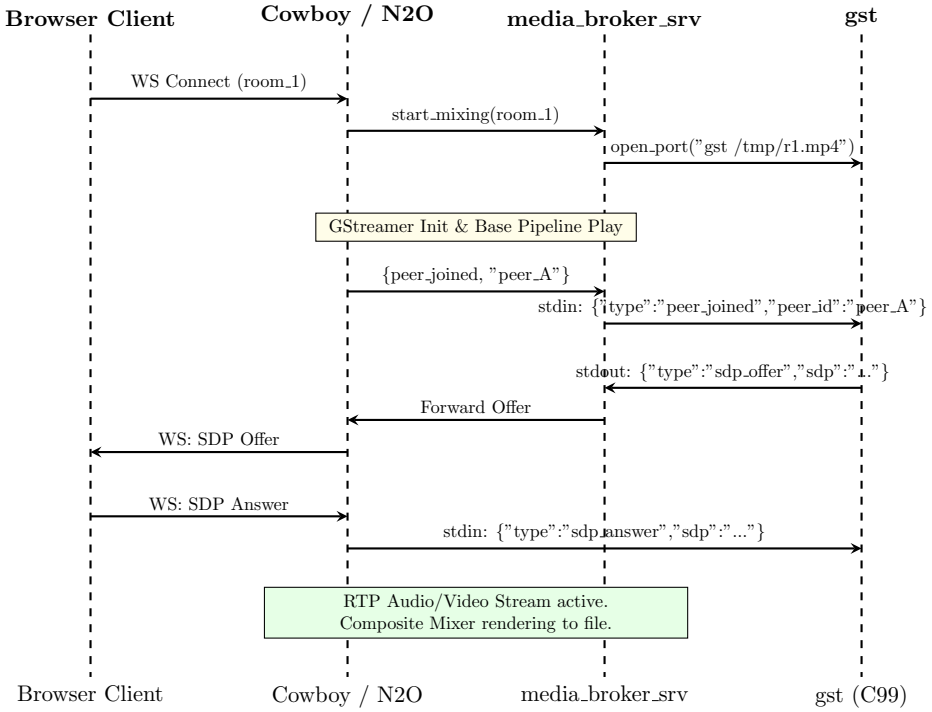

Figure 3: Signaling sequence showing local Port I/O communication between Erlang and the C99 GStreamer binary.

2.5 Multipoint Media Composition

The GStreamer media broker executes video grid compositing and audio conferencing concurrently. The media plane runs as a single GStreamer pipeline initialized in a C99 OS port process.

2.5.1 Video Matrix Grid Compositing

Incoming H.264/VP8 video packets are decoded into raw YUV frames. The compositor element (`compositor`) maps each input stream index $i \in \{1, 2, 3\}$ to a quadrant layout inside a unified 1080p canvas ($W_{\text{out}} = 1920$, $H_{\text{out}} = 1080$). The dimensions of each video frame are scaled to $w = 960$ and $h = 540$. The placement coordinates (x_i, y_i) are formulated as:

$$x_i = (i \pmod 2) \cdot w \tag{3}$$

$$y_i = \lfloor i/2 \rfloor \cdot h \tag{4}$$

The resulting video layout is composed as shown in Equation 5:

$$\text{Canvas}[x, y] = \begin{cases} \text{Stream}_0[x, y] & 0 \le x < 960,\ 0 \le y < 540 \\ \text{Stream}_1[x - 960, y] & 960 \le x < 1920,\ 0 \le y < 540 \\ \text{Stream}_2[x, y - 540] & 0 \le x < 960,\ 540 \le y < 1080 \\ \text{Stream}_3[x - 960, y - 540] & 960 \le x < 1920,\ 540 \le y < 1080 \end{cases} \tag{5}$$

2.5.2 Audio Mixing & Summation

Audio tracks are mixed using GStreamer's `audiomixer` element. Incoming Opus packets are decoded into linear pulse-code modulated (LPCM) audio buffers. The mixed audio signal $S_{\text{mixed}}(t)$ at timestamp t is the sum of the K active participant signals:

$$S_{\text{mixed}}(t) = \sum_{i=1}^{K} S_i(t) \tag{6}$$

To prevent digital clipping/overflow, the mixed LPCM buffer passes through a limiter/clipper stage before being re-encoded into Opus for WebRTC downstream broadcast:

$$S_{\text{output}}(t) = \text{Clamp}(S_{\text{mixed}}(t), -S_{\max}, S_{\max}) \tag{7}$$

2.5.3 C99 GStreamer Pipeline Implementation

The implementation in `c_src/gst.c` integrates WebRTC media reception, grid composition, and recording:

1. **Continuous Live Sources**: To prevent preroll deadlocks when no peers are connected, the pipeline initializes with a continuous static background black video (`videotestsrc pattern=black`) and audio silence (`audiotestsrc volume=0`) bound permanently to pad `sink_0` on the compositor and audiomixer.

2. **Explicit Layering (Z-Ordering)**: GStreamer's `compositor` layers input pads according to the `zorder` property. The background pad is configured with zorder = 1. When a dynamic peer video track is decoded in the `on_decoded_pad` callback, the calculated compositor pad is assigned an explicit, higher zorder = idx + 10 (e.g., $11, 12, 13$), layering it on top of the black canvas.

3. **Loop Decoupling and Leaky Queues**: Because each peer's `webrtcbin` acts as both a media source and a broadcast destination, the graph contains feedback loops. We introduce independent `tee` elements (`vtee`/`atee`) to broadcast the mixed grid back to all clients. Each broadcast branch is isolated via dynamic queues (`vqueue_<peer_id>`) configured with leaky downstream properties (`leaky=2`) and restricted capacity (`max-size-buffers=60`), preventing slow or disconnected peers from blocking the upstream compositor.

4. **Dynamic Peer Lifecycle Management**: We track peer resources in a structured `PeerInfo` context. On receiving a `"peer_left"` signal via `stdin`, GStreamer invokes `cleanup_peer()`, which systematically unlinks pads, releases requested compositor/mixer pads, destroys the peer's converter and queue elements, and frees its `webrtcbin`. This prevents grid placement leakage and ensures memory and CPU utilization scale with the active stream count rather than historical connection counts.

5. **Safe Unix Signal finalization**: Direct Unix signals run outside the main loop thread and are unsafe to manipulate GStreamer bins directly. We use GLib's thread-safe `g_unix_signal_add` to schedule signal handling on the main event thread. When a termination signal is received, the handler sends an `EOS` event specifically to the recording muxer (`mp4mux`), which flushes standard MP4 index trailers safely before exit.

3 Capacity, Telemetry and Metrics

This section evaluates resource provisioning and cluster capacity for a target workload of **1,000 concurrent MCU rooms** (compositing multiple 720p streams @ 30 fps into unified outputs) with hardware acceleration. The target distribution comprises:

- **80% (800 rooms)**: 6-participant rooms.
- **20% (200 rooms)**: 20-participant rooms.

To satisfy these requirements while minimizing costs, we specify a heterogeneous GPU-accelerated cluster consisting of **40 physical nodes**: 20 Small Nodes and 20 Big Nodes.

3.1 Node Specifications

Table 3 outlines the hardware profiles for the Small and Big nodes configured with GPU acceleration and cost-effective DDR4 memory.

Table 3: Node Types and Specifications (GPU-Accelerated, Lower DDR4)

Component	**Small Node (6-person)**	**Big Node (20-person)**
Target Use	6-participant rooms	20-participant rooms
CPU	Single Xeon Silver/Gold (16–24 cores)	Dual Xeon Gold/Platinum (32–64+ cores)
RAM	64–96 GB DDR4	128 GB DDR4
GPU	1 × Mid-High NVIDIA (e.g., RTX 4090 / A10)	2 × High-End NVIDIA (e.g., A40 / L40S)
Network	25–40 Gbps	40–100 Gbps
Storage	256 GB NVMe SSD	512 GB NVMe SSD

3.2 Per-Node and Cluster Capacity

Under GPU acceleration, decoding is performed via NVDEC and encoding via NVENC, drastically reducing CPU core utilisation. Table 4 lists the real-time room capacity per node type, and Table 5 summarizes the total cluster capacity and resource footprint.

Table 4: Real-Time Capacity per Node

Metric	**Small Node**	**Big Node**
Rooms per Node	45	12
Input Streams per Node	≈ 270	≈ 240
Output Streams per Node	45	12

Table 5: Total Cluster Capacity and Resource Summary

Category	**Small Nodes (20)**	**Big Nodes (20)**	**Total / Headroom**
6-Person Rooms	900	–	900 (**+100 headroom**)
20-Person Rooms	–	240	240 (**+40 headroom**)
Max Mixed Rooms	–	–	**1,140**
Target Workload	800	200	**1,000 (covered with 14% headroom)**
Total DDR4 RAM			$\approx$ 3.8–4.2 TB
Total NVIDIA GPUs			60

3.3 System Model & Parameter Definitions

We model the memory and CPU consumption of the GStreamer MCU processes to validate these provisioning limits.

3.3.1 Memory Allocation Model

The total RAM consumption M_{total} of a node is formulated as:

$$M_{\text{total}} = M_{\text{base}} + N \cdot M_{\text{conn}} + R \cdot (M_{\text{room}} + M_{\text{gst}}(K)) \tag{8}$$

Where the GStreamer media compositor RAM footprint for K active peer connections is:

$$M_{\text{gst}}(K) = M_{\text{base\_gst}} + K \cdot M_{\text{peer\_gst}} \tag{9}$$

For a Small Node hosting $R = 45$ rooms with $K = 6$ participants each:

- $M_{\text{base\_gst}} \approx 120$ MB (base GStreamer and pipeline structures).
- $M_{\text{peer\_gst}} \approx 35$ MB (buffers and decode pipelines per 720p stream).
- $M_{\text{gst}}(6) = 120 + 6 \cdot 35 = 330$ MB per room.
- $M_{\text{total}} \approx 45 \cdot (330 + 0.005) \approx 14.85$ GB of heap/process memory, easily fitting within the 64–96 GB DDR4 Small Node configuration.

Similarly, for a Big Node hosting $R = 12$ rooms with $K = 20$ participants each:

- $M_{\text{gst}}(20) = 120 + 20 \cdot 35 = 820$ MB per room.
- $M_{\text{total}} \approx 12 \cdot (820 + 0.005) \approx 9.84$ GB process RAM, fitting well within the 128 GB DDR4 specification. The remaining host memory provides ample caching, OS buffers, and network ring buffers.

3.3.2 CPU and GPU Offloading Model

With GPU-offloaded media pipelines, the total CPU load C_{total} remains low because the heavy matrix composition, pixel format conversion, and H.264 encoding are executed on CUDA and NVENC. The node's processing capability is instead bound by the GPU's hardware session limits (NVENC concurrent streams) and network throughput.

The network throughput T_{net} per node is defined by incoming and outgoing media streams:

$$T_{\text{net}} = R \cdot (K \cdot B_{\text{in}} + B_{\text{out}}) \tag{10}$$

Where B_{in} is the incoming client bitrate (1.5 Mbps H.264 720p) and B_{out} is the mixed output canvas bitrate (4.0 Mbps 1080p).

- **Small Node Traffic**: $45 \cdot (6 \cdot 1.5 + 4.0) = 585$ Mbps. This is well within the 25–40 Gbps network interface capacity.
- **Big Node Traffic**: $12 \cdot (20 \cdot 1.5 + 4.0) = 408$ Mbps. This is comfortably supported by the 40–100 Gbps interfaces, ensuring no network queuing jitter.

3.4 Systems Soundness & Liveness Analysis

HPA Oscillation (Thundering Herd)

Configuring Kubernetes Horizontal Pod Autoscaling (HPA) against standard CPU limits (e.g., 80% target) creates scaling instabilities when CPU is offloaded to GPUs. Because CPU utilization remains low under hardware acceleration, standard HPA fails to react to room saturation. The autoscaling algorithm must instead utilize custom metrics mapped directly to active GStreamer mixers (`active_gst_mixers`) and GPU memory utilization.

Local Database Isolation (No Database Clustering)

Under our sharded node model, Mnesia runs strictly as a local persistent storage database inside each pod's local volume. Because Erlang nodes do not form a distribution cluster, database transaction locking and schema updates are restricted to the local host. This completely avoids split-brain partitions, node discovery synchronization issues, and cluster-wide database deadlocks during rapid scale-up/down events.

Scale-Out Architecture Bottlenecks at 1000+ Rooms

When scaling the cluster horizontally to support $R = 1,000$ active rooms across the 40 GPU-accelerated nodes:

1. **Erlang Distribution Overhead Bypassed**: By routing all room participants to the same node via Ingress Sticky Session hashing, signaling is fully handled locally in-memory. Consequently, the Erlang distribution

mechanism is completely disabled (the nodes run with `-no_distribution`), totally avoiding quadratic $O(P^2)$ node connection storms.

2. **Mnesia Transaction Locking Isolated**: Since Mnesia runs as a purely local database on each isolated host, transaction lock contention is strictly confined to the local node's processes, eliminating distributed two-phase commit latency.

3. **Network Port & Bandwidth Saturation**: At an average stream bitrate of 1.5 Mbps, the aggregate network ingestion throughput reaches $4,000 \times 1.5$ Mbps $= 6.0$ Gbps. SREs must deploy network interfaces using `hostNetwork` configurations with SRIOV-enabled interfaces to bypass container network encapsulation.

4 Conclusion

The proposed monolithic architecture is highly sound for large-scale signaling and STUN/TURN routing, easily supporting over 10,000 concurrent user events. By transitioning to hardware-accelerated video compositing (using NVIDIA NVENC and CUDA offloading) and partitioning media processing dynamically between Small and Big nodes, the system achieves the target scale of 1,000 concurrent rooms within a cost-effective, low-memory 40-node cluster.

The `rtp` architecture demonstrates that video conferencing signaling and media compositing can be consolidated into a share-nothing Erlang/GStreamer node. By replacing microservice networks and headless browsers with local `gst` C99 binary opened as port interfaces and Ingress Room Sticky Sessions, the system avoids cluster synchronization locks and mesh overhead. This provides a highly resilient, cost-effective SRE model for secure networks.

I want to dedicate this work to Mariia Bielikova who support me during this creation.

References

[1] Rosenberg, J., and Schulzrinne, H. An Offer/Answer Model with the Session Description Protocol (SDP). Internet Engineering Task Force (IETF), RFC 3264, June 2002.

[2] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A Transport Protocol for Real-Time Applications. Internet Engineering Task Force (IETF), RFC 3550, STD 64, July 2003.

[3] Baugher, M., McGrew, D., Naslund, E., Carrara, E., and Norrman, K. The Secure Real-time Transport Protocol (SRTP). Internet Engineering Task Force (IETF), RFC 3711, March 2004.

[4] Keranen, A., Mutabozi, E., and Melnikov, A. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. Internet Engineering Task Force (IETF), RFC 8445, October 2018.

[5] Jennings, C., Bostrom, H., and Bruaroey, J.-I. JavaScript Session Establishment Protocol (JSEP). Internet Engineering Task Force (IETF), RFC 8829, January 2021.

[6] W3C Recommendation. WebRTC 1.0: Real-Time Communication Between Browsers. W3C, January 2021. Available online: `https://www.w3.org/TR/webrtc/`.

[7] Ålund, S., and Ericsson Research. OpenWebRTC: Transcend the Browser. Ericsson Research Blog, September 2014. Available online: `https://github.com/EricssonResearch/openwebrtc`.

[8] Sean Song. WebRTC: The Protocol That Made the Browser a Phone. Sean's Lab Blog, April 2026. Available online: `https://seanslab.org/posts/webrtc-the-protocol-that-made-the-browser-a-phone`.

[9] ITU-T Recommendation X.601. Information technology - Multi-peer communications framework. International Telecommunication Union, 2000.

[10] ITU-T Recommendation X.602. Information technology - Group management protocol. International Telecommunication Union, 2001.

[11] ITU-T Recommendation X.609. Managed peer-to-peer communications. International Telecommunication Union, 2010.

[12] Marier, F. Peer-to-peer video conferencing using GStreamer and WebRTC. Available online: `https://feeding.cloud.geek.nz/posts/peer-to-peer-video-conferencing-using/`, 2020.

[13] Toradex Developer Center. Video Processing with GStreamer on Linux. Available online: `https://developer.toradex.com/software/linux-resources/multimedia/video-processing-gstreamer/`, 2026.

[14] Synrc Research. SRE Handbook: Operational Practices & Topology. Available online: `https://cd.erp.uno/sre.pdf`, 2026.