SYNRC: OpenWebRTC

Технічне завдання

для реалізації механізму синхронізації медіа потоків з низьколатентними вимогами реального часу і синхронізацією кадрів через головний тактовий генератор мікшера

На основі Технічних Вимог

Згідно з Постановою КМУ № 205 від 21.02.2025

Формалізованих за стандартом ISO/IEC/IEEE 29148:2018

Deterministic Real-Time Multimedia Synchronization: Architecting Low Latency Topologies in GStreamer and WebRTC

Namdak Tonpa, Zen Crypted

28 липня 2026 р.

Анотація

Achieving ultra-low latency and stable real-time multimedia transmission in heterogeneous environments requires rigorous architectural precision. The primary challenge lies in the deterministic reconciliation of conflicting asynchronous domains: non-blocking WebRTC transmission, unpredictable I/O operations (e.g., disk recording), and strict isochronous rendering (compositing). This article presents an engineering framework for resolving pipeline starvation, timestamp dilation, and latency accumulation by enforcing strict thread isolation, temporal queueing, and monotonic clock synchronization within GStreamer and WebRTC topologies.

Зміст

1. Introduction

The crux of real-time multimedia stability is the maintenance of a continuous, synchronized data flow across diverse subsystems. Naive topological designs often tightly couple sinks and processing elements within the same synchronous state space. Consequently, when unpredictable elements—such as a `filesink` or `hlssink2`—encounter blocking I/O, backpressure propagates upstream. If an isochronous compositor and a WebRTC sink share this state space, the compositor blocks, stalling the overarching temporal reference. Failure to isolate these domains dictates a departure from real-time execution, leading to catastrophic latency accumulation.

2. Media Sync Architecture

2.1. Asynchronous Domain Isolation

2.1.1. Thread Topology and I/O Decoupling

In GStreamer, queues define asynchronous thread boundaries. By default, data flow and state changes execute synchronously. If a downstream element stalls, it halts the thread of upstream elements. To guarantee localized I/O blocking or CPU spikes do not propagate backpressure, architectures must aggressively inject queues prior to encoders, sinks, and test sources.

This decoupling isolates components into independent OS threads, ensuring the master clock and compositor remain untethered. Furthermore, non-blocking state changes (`async=false`, `sync=false`) must be enforced on I/O-bound sinks.

2.1.2. Temporal Queuing and Absolute Drop Policies

Default queue implementations impose arbitrary limits on buffer counts and bytes, which are semantically poor metrics for real-time video where frame sizes fluctuate. To enforce an absolute zero-latency drop policy, queues must be strictly temporal.

By setting explicit temporal boundaries (`max-size-time=300000000`, `max-size-buffers=0`, `max-size-bytes=0`) coupled with a `leaky=downstream` policy, the queue is restricted to holding precisely 300 milliseconds of actual A/V time. If processing latency exceeds this threshold, older frames are instantaneously jettisoned. Trading recording fidelity for live stream continuity is a non-negotiable axiom of real-time design.

For base generators (`videotestsrc`, `audiotestsrc`), which produce deterministic, constant-rate data, micro-queues (`max-size-buffers=1` for video, `max-size-buffers=5` for audio) provide sufficient asynchronous decoupling without semantic delay, rendering the generators immune to downstream transients.

2.2. Synchronization and Temporal Monotonicity

2.2.1. Mitigating Timestamp Dilation

Timestamp dilation—perceived as a "slow motion"effect in WebRTC playback—is a classic desynchronization anomaly. It occurs when a processing pipeline (e.g., a compositor) is constrained by CPU saturation and fails to generate frames at its targeted framerate. If downstream filters enforce a strict high framerate (e.g., `30/1`), the compositor continues assigning timestamps based on the theoretical

rate. Consequently, one real-time second of processing may only yield 0.5 seconds of pipeline time, a discrepancy the WebRTC receiver faithfully obeys.

To mathematically guarantee A/V synchronization, the requested framerate must be explicitly lowered (e.g., `framerate=15/1`) to align with the hardware's guaranteed processing capacity. This ensures that generated frames are stamped accurately (0ms, 66ms, 133ms...), spanning exactly 1.0 real-time second, immediately eradicating the slow-motion defect.

2.2.2. Isochronous Interpolation for Variable Ingress

WebRTC clients dynamically throttle their egress framerates based on network congestion. Conversely, a compositor demands a constant, unwavering stream of frames. If a client's framerate drops, the compositor starves, blocking the pipeline.

This impedance mismatch is resolved by injecting a `videorate` element coupled with a strict `capsfilter` immediately post-decode. `videorate` acts as an interpolation buffer, deterministically duplicating frames to satisfy the compositor's isochronous requirements without inducing latency.

2.3. Hardware and Resource Constraints Optimization

Scaling software-based encoding and high-resolution compositing requires specific architectural adaptations on resource-constrained hardware (e.g., legacy x86, ARM platforms):

1. **Hardware Acceleration**: Offloading DSP tasks to dedicated ASICs (e.g., `v4l2h264enc`, QuickSync, NVENC is critical for predictable latency.
2. **Spatial Subsampling**: Reducing the compositor canvas resolution exponentially decreases memory bandwidth and CPU cycles.
3. **Thread Topology Management**: Enforcing strict CPU affinity and thread pinning mitigates context-switching overhead in the OS scheduler.
4. **Jitterbuffer Tuning**: Dynamically adjusting the `webrtcbin` internal RTP jitterbuffer latency based on measured network variance minimizes baseline delay in LAN environments while preserving stability on degraded WAN links.

2.4. Telemetry and Diagnostics in the RTP/RTCP Layer

Deep visibility into the RTP/RTCP session layer is paramount for maintaining stability. Utilizing the modern Rust RTP plugin (`rsrtp`), session state, network jitter, and clock skew can be profiled.

1. **Jitter and Latency**: Tracing incoming packet buffering evaluates if network jitter causes late arrivals to drop or monitors how out-of-order packets are reordered. Sequence number tracking explicitly identifies lost packets, correlating network health with pipeline starvation.
2. **Clock Skew & Synchronization**: Monitoring the skew calculation between the remote sender's clock and the local system clock (`timestamping-mode=skew`) maps RTP timestamps to local presentation time, isolating A/V sync drift.
3. **RTCP Quality Feedback**: Generating Sender Reports (SR) in `rtpsend` broadcasts packet counts and NTP timestamps. `rtprecv` handles Receiver Reports (RR), tracing NACKs (retransmission requests) or PLIs (keyframe requests) triggered by packet loss when operating under the `avpf` profile.

3. Conclusion

Architecting stable, ultra-low latency WebRTC and GStreamer pipelines mandates a shift from synchronous, tightly-coupled topologies to asynchronous, time-bounded, and heavily isolated designs. By enforcing strict thread boundaries, temporal queue policies, and monotonic timestamping, systems can achieve deterministic performance even under volatile network and I/O conditions.