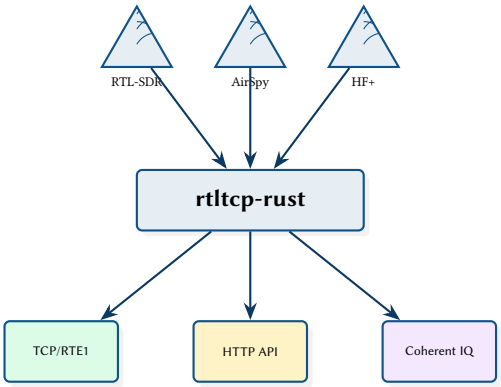
rtltcp-rust: A Multi-SDR Streaming Server with Extended Protocol and Coherent Calibration

Architecture, Protocol Design, and DSP Pipeline

Simon Knight

March 2026 • Version 0.1.0

Last updated: March 12, 2026



Abstract. Standard `rtl_tcp` servers stream unsigned 8-bit IQ samples from a single RTL-SDR device, discarding the higher-resolution native formats of modern SDR hardware. This report presents `rtltcp-rust`, a Rust reimplementation that supports three SDR families (RTL-SDR, AirSpy, AirSpy HF+), preserves native sample depth through a backward-compatible extended protocol (RTE1), and adds adaptive compression, UDP/FEC transport, an HTTP REST API, and a terminal UI. A coherent multi-device mode implements the HeIMDALL calibration FSM for KrakenSDR-class passive radar, producing synchronised multi-channel IQ frames over TCP. The server targets Raspberry Pi deployment via Docker-based cross-compilation.

Contents

SDR	Software-Defined Radio	2
IQ	In-phase / Quadrature	2
CPI	Coherent Processing Interval	
FSM	Finite State Machine	2
TUI	Terminal User Interface	2
API	Application Programming Interface	2
LE	Little-Endian	4
BE	Big-Endian	4
PPM	Parts Per Million	
UDP	User Datagram Protocol	5
TCP	Transmission Control Protocol	2

1 Introduction and Motivation

The `rtl_tcp` protocol [1] has become the de facto standard for streaming In-phase / Quadrature (IQ) samples from USB Software-Defined Radio (SDR) receivers to remote clients over TCP. Its simplicity—a 12-byte handshake followed by a raw byte stream—has led to wide adoption in applications such as SDR#, GQRX, and CubicSDR.

However, the original protocol has three fundamental limitations:

1. **Single device, single format.** Only RTL-SDR hardware is supported, locked to unsigned 8-bit IQ samples. Modern devices like the AirSpy (signed 16-bit) and AirSpy HF+ (32-bit float) have their samples crushed to 8-bit when served over `rtl_tcp`.
2. **No metadata channel.** Clients cannot discover the server's sample format, sample rate, or frequency without out-of-band coordination.
3. **No multi-device coherence.** Passive radar and direction finding require phase-synchronised multi-channel IQ streams, which the protocol cannot express.

`rtltcp-rust` addresses all three limitations through an extended protocol (RTE1) that negotiates native sample depth at connection time, a broadcast-based fan-out architecture that supports multiple concurrent clients per device, and a coherent multi-device mode that implements the HeIMDALL calibration Finite State Machine (FSM) [2] for KrakenSDR [3] compatibility.

Insight:
The 100 ms negotiation timeout is the key enabler for backward compatibility: standard clients never speak first, so they naturally fall through to legacy mode with zero code changes.

2 System Architecture

The server is a single Rust binary built on the Tokio async runtime [4]. It simultaneously supports a Transmission Control Protocol (TCP) streaming interface per device, an HTTP REST Application Programming Interface (API), an optional Terminal User Interface (TUI), and coherent multi-channel output.

2.1 Data Pipeline

The core data path is a two-stage channel architecture that bridges synchronous C driver callbacks into the async Tokio world, then fans out to an arbitrary number of consumers.

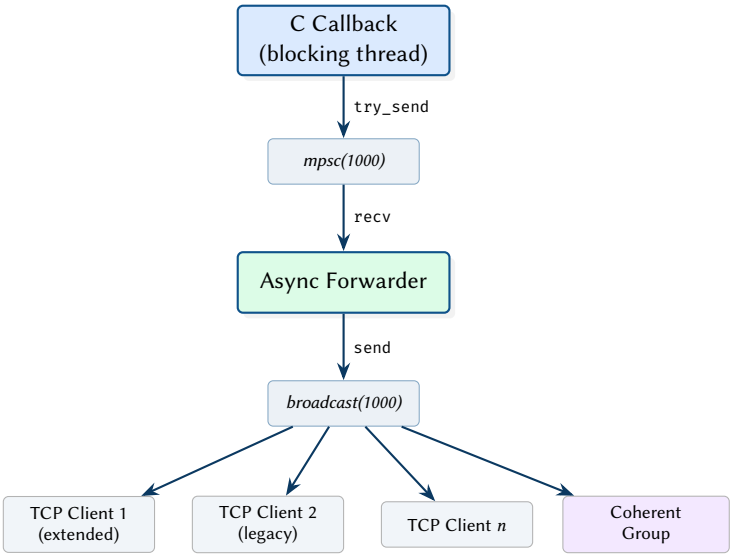
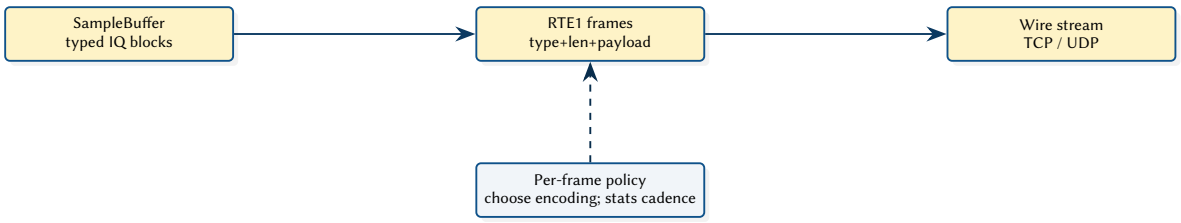


Figure 1. Two-stage data pipeline: device callback to broadcast fan-out.



Conceptual point. The system separates dataflow (typed buffers → frames → wire) from control policy (encoding selection, telemetry cadence). This keeps backwards compatibility while allowing per-frame adaptation.

Figure 2. Dataflow vs control-plane separation. Solid arrows carry IQ data; dashed arrows carry policy decisions that affect framing/encoding.

> **Design Decision 1: Two-Stage Channel Architecture**
>
> The C driver libraries (`librtlsdr`, `libairspyhf`, `libairspy`) deliver samples via blocking callbacks on threads they control. An `mpsc` channel bridges these into the Tokio async world (`try_send` is non-blocking, safe from C callbacks). The `broadcast` channel then provides $O(1)$ fan-out with per-subscriber lag detection—lagged clients receive `RecvError::Lagged(n)` rather than causing backpressure that would block faster consumers.

2.2 Device Abstraction

All SDR hardware is accessed through the `SDRDevice` trait:

```rust
#[async_trait]
pub trait SDRDevice: Send + Sync {
    fn set_frequency(&self, freq: u32) -> Result<()>;
    fn set_sample_rate(&self, rate: u32) -> Result<()>;
    fn set_gain(&self, gain: i32) -> Result<()>;
    fn run(&self, tx: mpsc::Sender<SampleBuffer>) -> Result<()>;
    fn stop(&self) -> Result<()>;
    fn sample_format(&self) -> SampleFormat;
    fn capabilities(&self) -> Result<DeviceCapabilities>;
    // ... default no-ops for bias_tee, freq_correction
}
```

Drivers are stored as `Arc<dyn SDRDevice>` trait objects. The `run()` method blocks on the C library's async read loop (e.g. `rtlsdr_read_async`), so it is always spawned via `tokio::task::spawn_blocking()`.

Table 1. Supported SDR hardware and native sample formats.

Device	Driver	Bits	Format	FFI
RTL-SDR	`rtlsdr.rs`	8	`U8`	`rtlsdr_sys`
AirSpy	`airspy.rs`	16	`I16`	Manual `#[link]`
AirSpy HF+	`airspyhf.rs`	32	`F32`	Manual `#[link]`

2.3 Server State Model

Insight: *The AirSpy and AirSpy HF+ drivers use fully manual FFI declarations rather than a `-sys` crate because no maintained Rust bindings exist for these libraries. The `build.rs` script uses `pkg-config` for linker path discovery, falling back to sysroot-relative paths for cross-compilation.*

Each device server maintains per-device state wrapped in `Arc` for shared access across Tokio tasks:

```rust
struct DeviceServerState {
    device: Arc<dyn SDRDevice>,
    data_tx: broadcast::Sender<SampleBuffer>,
    current_frequency: Arc<Mutex<u32>>,
    current_sample_rate: Arc<Mutex<u32>>,
    current_gain: Arc<Mutex<i32>>,
    bytes_sent: Arc<AtomicU64>,
    // ...
}
```

A `TaskManager` (`HashMap<String, (JoinHandle, CancellationToken)>`) controls device lifecycle, allowing the TUI to start and stop individual devices at runtime.

3 Extended Protocol (RTE1)

3.1 Negotiation

The protocol exploits a behavioural asymmetry in standard `rtl_tcp` clients: they connect and *wait* for the server's handshake without sending any data first. Extended clients identify themselves by sending the 4-byte magic `"RTE1"` before the server's 100 ms timeout expires.

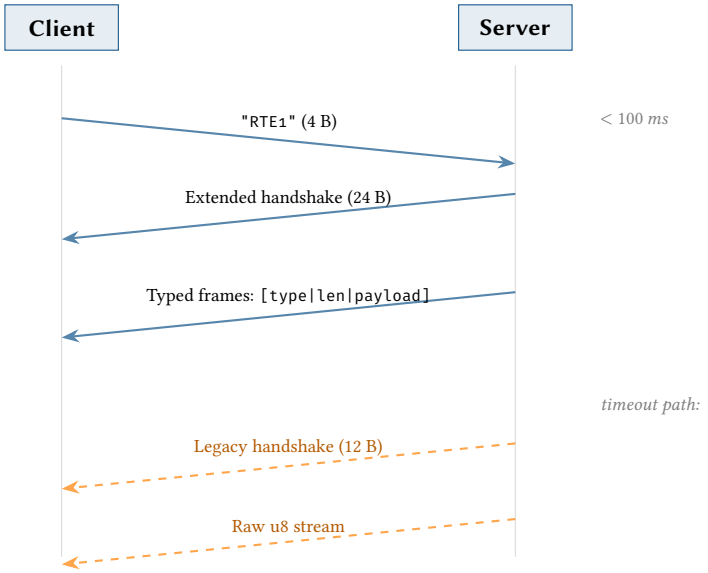


Figure 3. Protocol negotiation: extended vs. legacy path.

: Zero-Cost Backward Compatibility

The extended protocol must not require any changes to existing `rtl_tcp` clients. A timeout-based detection mechanism ensures that legacy clients receive the standard 12-byte RTL0 handshake and raw u8 stream with no modifications.

3.2 Extended Handshake

The 24-byte extended handshake advertises the device's native sample format, current tuning parameters, and server capabilities:

Table 2. Extended handshake wire format (24 bytes, server → client).

Offset	Size	Field	Encoding	Description
0	4	magic	ASCII	`"RTE1"` echo
4	1	sample_format	u8	0=u8, 1=i16le, 2=f32le
5	1	bits_per_sample	u8	8, 16, or 32
6	1	capability_flags	u8	Bit 0: stats frames
7	1	reserved	zero	—
8	4	sample_rate	u32 Big-Endian (BE)	Sample rate (Hz)
12	4	center_freq	u32 BE	Center frequency (Hz)
16	4	device_serial	u32 BE	Lower 32 bits of serial
20	4	reserved	zero	—

Design Decision 2: Big-Endian Control Fields, Little-Endian Sample Data

The handshake and command fields use big-endian (network byte order) for consistency with the original `rtl_tcp` protocol. Sample data uses little-endian because both ARM (Raspberry Pi) and x86 targets are natively Little-Endian (LE), avoiding byte-swap costs on the critical data path.

3.3 Typed Framing

In extended mode, each transmission unit is a typed frame:

Table 3. Frame header format (5 bytes).

Offset	Size	Field	Encoding
0	1	frame_type	u8
1	4	payload_length	u32 LE

Frame types:

- `0x00` (**Data**) — raw IQ in negotiated format
- `0x01` (**Stats**) — JSON telemetry (~1 Hz)
- `0x02` (**DataEncoded**) — compressed/encoded data with 8-byte sub-header

3.4 Adaptive Compression

For each sample buffer, the server evaluates all enabled encodings and selects the smallest representation that beats the baseline by at least `min_gain_bytes` (default 64):

1. **Raw / quantise-to-u8** — baseline size
2. **LZ4** [5] — block compression
3. **Delta + LZ4** — wrapping-subtraction delta coding followed by LZ4; first frame is a keyframe

Design Decision 3: Per-Frame Encoding Selection

Rather than committing to a single encoding for the session, the server picks the best encoding per frame. This adapts naturally to varying signal conditions: delta+LZ4 compresses well for narrowband signals (high sample-to-sample correlation) while raw may win for wideband noise where compression overhead exceeds savings.

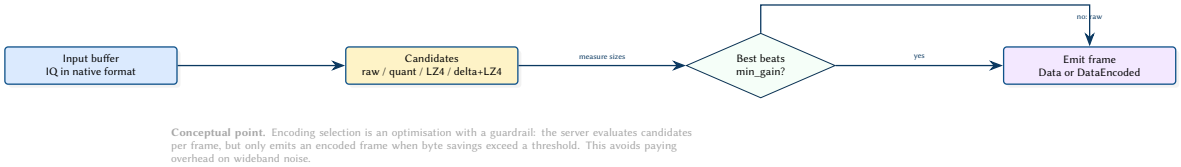


Figure 4. Adaptive compression as a decision DAG. Candidate encodings are evaluated per frame; a minimum-gain threshold prevents negative compression.

Encoded frames carry an 8-byte sub-header:

Table 4. Encoded payload sub-header (8 bytes).

Offset	Size	Field	Encoding
0	1	encoding_id	u8 (1=LZ4, 2=quant, 3=delta+LZ4)
1	1	decoded_format	u8 (SampleFormat)
2	1	flags	u8 (bit 0 = KEYFRAME)
3	1	reserved	zero
4	4	decoded_len	u32 LE (decompressed byte size)

3.5 UDP Transport

Two User Datagram Protocol (UDP) datagram formats complement the primary TCP stream for low-latency or lossy-link scenarios:

RTU1 (no FEC) 20-byte header: magic `"RTU1"` + stream ID + sequence number + payload. Payload chunked at 1200 bytes (MTU-safe). Socket send buffer expanded to 1 MB via `socket2`.

RTU2 (RaptorQ FEC) 32-byte header with RaptorQ [6, 7] Object Transmission Information. 64 KiB blocks with 8 repair packets per block, enabling reconstruction from any subset of $k + 8$ received packets.

Insight:
UDP transport always converts to u8 IQ regardless of the device's native format. This simplifies the receiver and keeps datagram sizes small—a single f32 sample pair is 8 bytes vs. 2 bytes in u8.

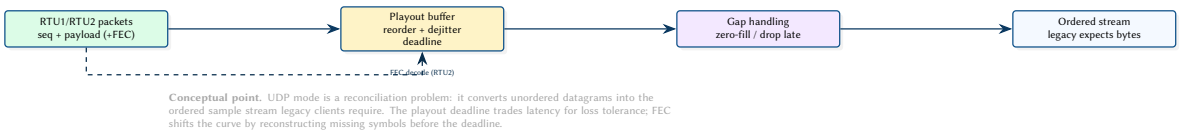


Figure 5. UDP receive path as a deadline-based reorderer. Playout buffering and (optional) FEC reconstruction convert an unreliable datagram stream into an ordered sample stream.

3.6 Extended Commands

Standard `rtl_tcp` commands (5-byte format) work in both modes. Extended-mode adds:

Table 5. Extended command set (client → server).

Byte	Command	Parameter
`0x01`	Set frequency	Hz (u32 BE)
`0x02`	Set sample rate	Hz (u32 BE)
`0x04`	Set gain	Device-specific
`0x80`	SET_LZ4	0=off, 1=on
`0x81`	SET_QUANTIZE_U8	0=off, 1=on
`0x82`	SET_MIN_GAIN	Minimum byte savings
`0x83`	SET_UDP_PORT	0=off, 1–65535=port
`0x84`	SET_UDP_FEC	0=off, 1=on
`0x85`	SET_DELTA_LZ4	0=off, 1=on

3.7 Legacy Conversion

When a standard client connects, higher-resolution samples are converted server-side to unsigned 8-bit:

$$\text{i16} \to \text{u8}: \quad \big((sample \gg 8) \text{ as u8}\big) \oplus \texttt{0x80} \tag{1}$$

$$\text{f32} \to \text{u8}: \quad \text{clamp}\big(sample \times 127.5 + 127.5,\ 0,\ 255\big) \tag{2}$$

Both are standard offset-binary mappings that preserve the DC bias point at sample value 127.

4 Coherent Multi-Device Mode

The coherent mode groups N RTL-SDR devices and runs an 8-state calibration FSM to synchronise their local oscillators and correct per-channel amplitude/phase offsets. The output is a multi-channel IQ frame compatible with the KrakenSDR passive radar receiver [3].

4.1 Calibration State Machine

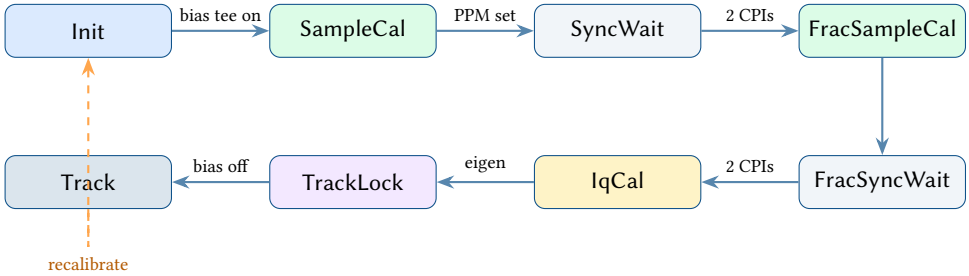


Figure 6. Calibration FSM state transitions.

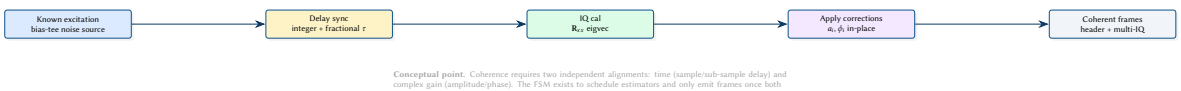


Figure 7. Coherent calibration as a two-stage estimation problem: delay alignment followed by complex gain (IQ) alignment.

Table 6. Calibration FSM actions per state.

State	Action
Init	Enable bias tee on reference device (drives noise source)
SampleCal	FFT cross-correlation per channel → integer sample delays → PPM corrections
SyncWait	Discard 2 CPIs while hardware settles
FracSampleCal	Block-FFT spectral phase slope → fractional delay refinement
FracSyncWait	Discard 2 CPIs
IqCal	Spatial correlation matrix → eigendecomposition → amplitude/phase corrections
TrackLock	Apply corrections in-place; disable noise source
Track	Steady-state: apply corrections, build IqHeader, output frames

4.2 DSP Algorithms

The calibration pipeline implements four core algorithms, all ported from HeIMDALL's `delay_sync.py` [2].

4.2.1 Integer Delay via Cross-Correlation

Given reference signal $x_{\text{ref}}[n]$ and channel signal $x_{\text{ch}}[n]$, the integer sample delay is estimated by:

$$\hat{\tau} = \arg\max_k \left| \mathcal{F}^{-1}\{X^*_{\text{ref}}(f) \cdot X_{\text{ch}}(f)\}[k] \right| \tag{3}$$

where $X(f) = \mathcal{F}\{x[n]\}$ is the DFT, computed after zero-padding both signals to $2N$ (next power of two) to avoid circular aliasing [8]. The circular index is unwrapped to a signed lag.

4.2.2 Fractional Delay via Spectral Phase Slope

For sub-sample precision, the signal is divided into blocks of size B (default 1024). Per block, the transfer function $H(f) = X_{\text{ch}}(f)/X_{\text{ref}}(f)$ is computed (bins below -60 dB magnitude are masked). The unwrapped phase $\angle H(f)$ is fit via least-squares to extract the slope [9]:

$$\hat{\tau}_{\text{frac}} = -\frac{\text{slope} \cdot B}{2\pi} \tag{4}$$

The per-block estimates are averaged across all valid blocks in the CPI.

4.2.3 Spatial Correlation Matrix

The $N \times N$ Hermitian correlation matrix is:

$$\mathbf{R}_{xx} = \frac{1}{M}\mathbf{X}\mathbf{X}^H \tag{5}$$

where $\mathbf{X} \in \mathbb{C}^{N \times M}$ contains N channels of M samples each. The implementation uses `nalgebra` with `DMatrix<Complex<f64>>`.

4.2.4 IQ Calibration via Eigendecomposition

The dominant eigenvector of $\mathbf{R}_{xx}$ encodes the per-channel amplitude and phase offsets relative to the reference channel. For each channel i:

$$a_i = |v_i|/|v_0| \qquad \text{(amplitude correction)} \tag{6}$$

$$\phi_i = \angle v_i - \angle v_0 \qquad \text{(phase correction)} \tag{7}$$

where $\mathbf{v}$ is the eigenvector corresponding to the largest eigenvalue. Corrections are applied in-place: $s[n] \leftarrow s[n] \cdot a_i \cdot e^{j\phi_i}$.

4.3 IQ Frame Output

The coherent output uses a 1024-byte binary header matching HeIMDALL's `iq_header.py` format, followed by per-channel complex-float data:

$$\text{Frame} = \underbrace{[\text{IqHeader}]}_{1024\,\text{B}} \underbrace{[\text{ch}_0][\text{ch}_1]\cdots[\text{ch}_{N-1}]}_{N \times \text{CPI} \times 8\,\text{B}} \tag{8}$$

For a 5-channel KrakenSDR with CPI = 262,144, each frame is $1024 + 5 \times 262{,}144 \times 8 \approx 10$ MB.

: Wire Compatibility with HeIMDALL

The IqHeader must be byte-for-byte compatible with `iq_header.py` to allow `krakenSDR_receiver.py` to consume the stream in Ethernet mode without modification. All fields use little-endian encoding to match Python's `struct.pack` default on ARM/x86.

Key header fields:

Table 7. Selected IqHeader fields (1024 bytes total).

Offset	Type	Field	Notes
0	u32	sync_word	`0x2BF7B95A`
4	u32	frame_type	0=Data, 1=Dummy, 3=Cal
28	u32	active_ant_chs	Number of channels
60	u32	cpi_length	Samples per channel
228	u32	delay_sync_flag	1 when sample-aligned
232	u32	iq_sync_flag	1 when IQ-calibrated
236	u32	sync_state	CalState as u32
240	u32	noise_source	1 when bias tee active

5 HTTP API and Terminal UI

5.1 REST API

The HTTP API is built on Axum [10] and serves JSON responses. All state is accessed through an `Arc<ApiState>` injected via Axum's `State` extractor.

Table 8. HTTP API endpoints.

Method	Path	Description
GET	`/api/v1/health`	Liveness check
GET	`/api/v1/server`	Version, uptime, device count
GET	`/api/v1/devices`	All devices with status
GET	`/api/v1/devices/{n}`	Single device (404 if absent)
GET	`/api/v1/devices/{n}/capabilities`	Hardware caps (503 if loading)
GET	`/api/v1/system`	CPU, memory, temperature
GET	`/api/v1/coherent`	List coherent groups
POST	`/api/v1/coherent`	Create coherent group
DELETE	`/api/v1/coherent/{n}`	Destroy group
POST	`/api/v1/coherent/{n}/recalibrate`	Restart FSM

5.2 Terminal UI

The TUI [11] provides two tabs—**Devices** (real-time status, inline parameter input) and **Logs** (scrolling output). Device commands flow over a `broadcast::channel` so all device server tasks see them simultaneously; each ignores commands not addressed to it.

Design Decision 4: C Library stderr Capture

C driver libraries print USB discovery messages directly to file descriptor 2. In raw terminal mode these lack carriage returns, causing "staircase" rendering. The server captures stderr around driver calls via POSIX `pipe()`/`dup2()`, drains the pipe, and routes captured output through the TUI log channel.

6 macOS Bridge (`mac_tunnel`)

Many popular macOS SDR applications—CubicSDR, GQRX, SDR# (via Mono)—speak only the legacy RTL0 `rtl_tcp` protocol: a 12-byte handshake followed by an unsigned 8-bit IQ stream. The `mac_tunnel` binary acts as a transparent protocol bridge: it connects upstream to an `rtltcp-rust` server using the extended RTE1 protocol (receiving native sample depth and adaptive compression), then re-serves the data locally as a standard RTL0 server that any legacy application can connect to without modification.

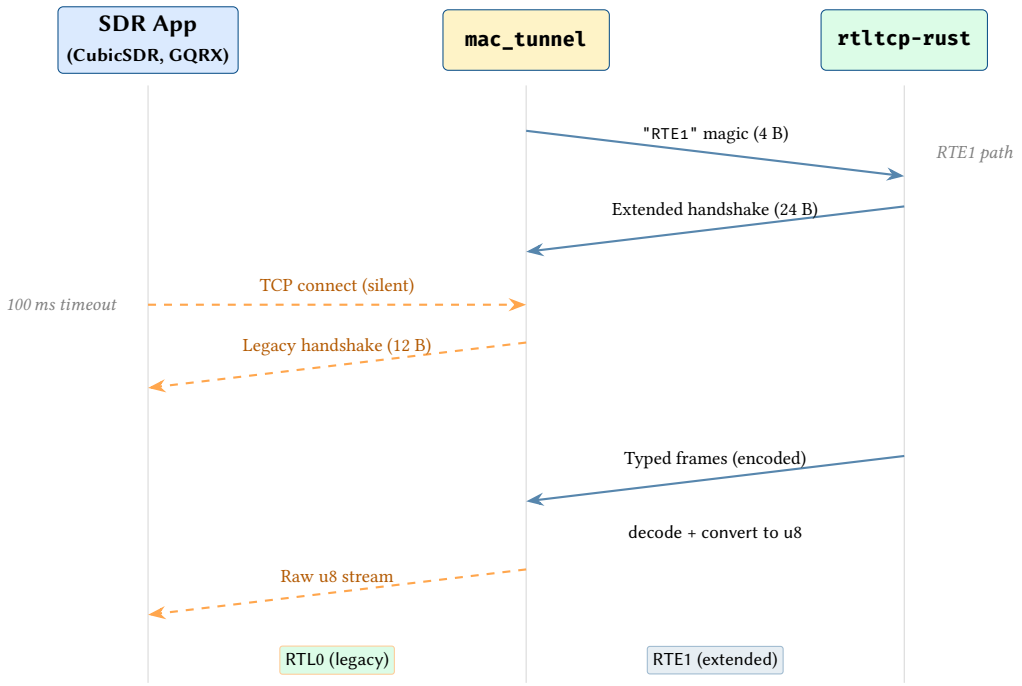


Figure 8. Protocol bridge: `mac_tunnel` translates between RTE1 and RTL0.

6.1 Encoding Decode Paths

When the upstream server sends encoded frames (`0x02` DataEncoded), the tunnel must decode them before converting to raw u8 for the legacy client. The decode path depends on the `encoding_id` in the 8-byte sub-header:

Table 9. Tunnel decode paths by encoding type.

ID	Encoding	Decode Path
1	LZ4	Decompress via `lz4_flex`; convert samples to u8
2	Quantize-u8	Pass through (payload is already unsigned 8-bit)
3	Delta+LZ4	Decompress via `lz4_flex`; apply delta to reference frame; update reference; convert to u8

Design Decision 5: Delta+LZ4 Keyframe State Machine

The delta+LZ4 encoding requires the tunnel to maintain a reference frame for each active stream. The `flags` byte in the encoded payload sub-header (bit 0 = `KEYFRAME`) drives a two-state machine:

- **Keyframe received** (`flags & 0x01`): decompress the LZ4 payload and store the result as the new reference frame. Output this frame directly.
- **Delta frame** (no keyframe flag): decompress the LZ4 payload to obtain the delta buffer, then apply wrapping addition to the current reference frame: ref[i] += delta[i]. Output the updated reference frame.

The client *must* track reference frame state; a missed keyframe causes all subsequent delta frames to produce corrupt output until the next keyframe arrives.

6.2 UDP Mode

In addition to TCP, `mac_tunnel` can receive samples over UDP using the RTU1 and RTU2 datagram formats. A playout buffer reorders and dejitters incoming datagrams before feeding the decode pipeline.

RTU1 (no FEC) Simple UDP datagrams with 20-byte header: magic `"RTU1"` + stream ID + sequence number + payload. Datagrams are inserted into the playout buffer keyed by sequence number. Out-of-order packets are reordered; gaps beyond the playout window are treated as losses and zero-filled.

RTU2 (RaptorQ FEC) 32-byte header carrying RaptorQ [6, 7] Object Transmission Information. The playout buffer collects encoding symbols per source block and attempts reconstruction once k or more symbols arrive. With 8 repair symbols per block, the receiver tolerates up to 8 lost datagrams per 64 KiB block.

The playout buffer is parameterised by a target depth (default 20 ms), expressed as a number of sequence slots. Packets arriving later than the playout deadline are silently discarded. The buffer emits samples in strict sequence order, bridging the unreliable UDP transport into the ordered byte stream expected by legacy RTL0 clients.

Insight: *UDP mode is particularly useful for macOS clients connected over Wi-Fi, where the combination of RTU2 FEC and the playout buffer masks transient packet loss that would otherwise cause TCP retransmission stalls.*

7 Cross-Compilation and Deployment

The server targets Raspberry Pi via the `cross` tool with custom Docker images that cross-compile `librtlsdr`, `libairspyhf`, and `libairspy` from source for the target architecture.

Table 10. Cross-compilation targets.

Target triple	Pi model	Docker image
`aarch64-unknown-linux-gnu`	Pi 3/4/5 (64-bit)	`rtltcp-cross:aarch64`
`armv7-unknown-linux-gnueabihf`	Pi 2/3 (32-bit)	`rtltcp-cross:armv7`
`arm-unknown-linux-gnueabihf`	Pi Zero/1	`rtltcp-cross:arm`

The `deploy-to-pi.sh` script handles the full pipeline: cross-build, optional strip, `scp` to Pi, and USBFS buffer configuration (256 MB recommended for SDR streaming).

All DSP dependencies (`rustfft`, `ndarray`, `nalgebra`, `num-complex`) are pure Rust and cross-compile without issue.

8 Test Infrastructure

8.1 Unit Tests

The protocol module (`protocol.rs`) contains 30+ inline tests covering:

- `SampleBuffer` serialisation/deserialisation roundtrips for all three formats
- Misalignment rejection (odd bytes for I16, non-multiple-of-4 for F32)
- Handshake build/parse roundtrips (both extended and legacy)
- Frame header encoding/decoding
- Encoded payload roundtrips (LZ4, quantise, delta+LZ4)
- UDP RTU1 and RTU2 datagram build/parse with bad-magic and too-short rejection

The calibration module (`calibration.rs`) tests all DSP functions with synthetic signals, verifying cross-correlation delay detection, fractional delay estimation accuracy (< 0.05 samples), spatial correlation matrix properties, and IQ calibration recovery of known amplitude/phase offsets.

8.2 Integration Tests

Integration tests in `tests/` use a `MockDevice` implementing `SDRDevice` to test the coherent FSM without hardware. The end-to-end test spawns the full `run_coherent_group()` task, feeds synthesised tone data via broadcast channels, connects a TCP client, reads IQ frames, and asserts the FSM reaches the Track state.

> **Design Decision 6: Library Target for Testing**
>
> The crate is primarily a binary, but `lib.rs` re-exports the modules needed by integration tests. This allows `cargo test` to compile tests against the library target without coupling to `main.rs` specifics or requiring a running server.

9 Design Summary

Table 11. Key design decisions and their rationale.

Decision	Rationale
Broadcast channel fan-out	$O(1)$ fan-out with per-subscriber lag detection; lagged clients get `RecvError::Lagged` rather than blocking faster consumers
Two-stage mpsc → broadcast	C callbacks cannot be async; mpsc bridges the synchronous C world into Tokio
100 ms timeout negotiation	Standard clients never speak first, so timeout naturally selects legacy mode
Per-frame encoding selection	Adapts to varying signal conditions without session-level commitment
LE sample data, BE control fields	LE avoids byte-swap on ARM/x86 data path; BE matches original protocol convention
stderr capture via dup2	Prevents C library writes from corrupting TUI raw terminal rendering
Manual FFI for AirSpy	No maintained Rust bindings exist; manual declarations avoid unmaintained dependency

References

[1] S. Markgraf, D. Stolnikov, and Hoernchen, "librtlsdr: Turns your Realtek RTL2832 based DVB dongle into a SDR receiver," in *osmocom project*, 2012. [Online]. Available: https://github.com/osmocom/rtl-sdr

[2] KrakenRF, *HeIMDALL DAQ Firmware*, 2023. [Online]. Available: https://github.com/krakenrf/heimdall_daq_fw

[3] C. Laufer et al., "KrakenSDR: Five channel coherent RTL-SDR receiver," in *Crowd Supply*, 2022. [Online]. Available: https://www.crowdsupply.com/krakenrf/krakensdr

[4] Tokio Contributors, *Tokio: An asynchronous runtime for Rust*, 2024. [Online]. Available: https://tokio.rs

[5] Y. Collet, "LZ4: Extremely fast compression," 2011. [Online]. Available: https://lz4.org

[6] A. Shokrollahi, "Raptor codes," in *IEEE Transactions on Information Theory*, vol. 52, 2006, pp. 2551–2567.

[7] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder, "RaptorQ forward error correction scheme for object delivery," IETF, RFC 6330, 2011.

[8] G. C. Carter, "Coherence and time delay estimation," *Proceedings of the IEEE*, vol. 75, no. 2, pp. 236–255, 1987.

[9] R. C. Williamson, "Delay estimation using narrow-band processes," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 3, pp. 577–581, 1981.

[10] Tokio Contributors, *Axum: Ergonomic and modular web framework built with Tokio*, 2024. [Online]. Available: https://github.com/tokio-rs/axum

[11] Ratatui Contributors, *Ratatui: A Rust library for cooking up terminal user interfaces*, 2024. [Online]. Available: https://ratatui.rs

List of Figures

List of Tables

List of Design Decisions & Rules

Revision History

Version	Date	Changes
0.1.0	March 2026	Initial architecture report