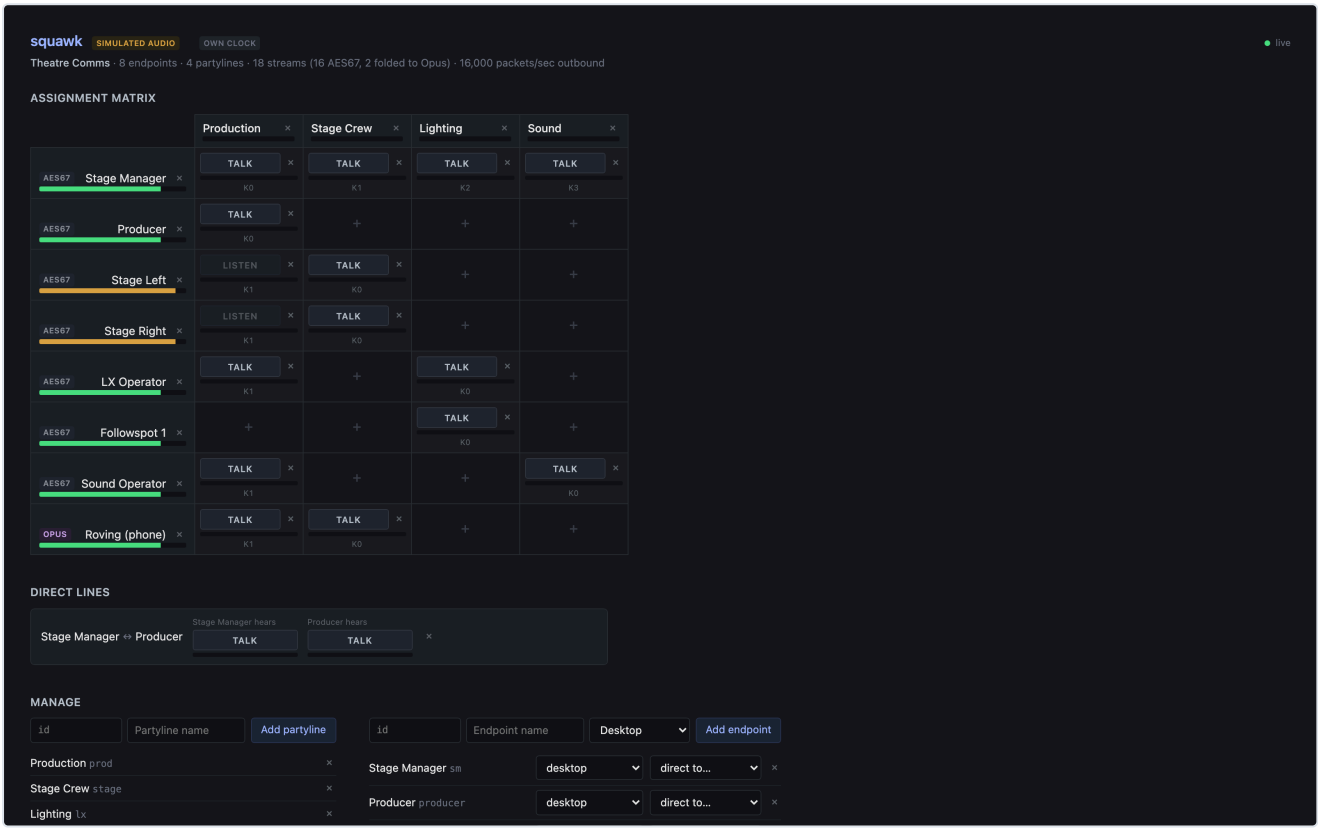
squawk user guide

squawk is an **open partyline intercom** — the software half of a Green-Go / Bolero / Clear-Com style comms rig. A server mixes; endpoints talk and listen.



The browser UI on the example show — eight endpoints across four partylines, each key with its own talk button and a meter of what it hears, and a direct line between Stage Manager and Producer. Running with no interface, so the audio is synthesised tones and nothing reaches the network, as the page's own badges say. The only rig it has mixed is this machine over loopback.

Each endpoint gets up to **10 keys**, and each key is its own AES67 stream carrying either a **partyline** (the bus, minus that endpoint's own voice) or a **direct** point-to-point path. Because the keys arrive separately, **a panel changes its own key levels, mutes and ear placement instantly and locally**, with no round trip to the server.

> **Before you rely on this:** audio goes in and out over real AES67 multicast with mix-minus asserted sample-exact through the round trip, and PTP drives the media clock.
>
> **But all of it has only ever run over the loopback interface, against a grandmaster this project wrote itself.** No SAP, no client, no hardware — nothing here has met another vendor's device, a real switch, or a microphone. **This is not a system to put on a show.**
>
> This codebase was created with AI assistance, directed and reviewed by a human author.

What exists today

Piece	State
Data model, config, validation	Built and tested
Mix-minus engine	Built and tested
Server and browser UI	Built and tested
AES67 packets, SDP, jitter buffer, sockets	Built and tested **over real UDP**
Transport wired into the server	**Working** — audio in and out over real multicast
PTP slave	Built; locks to a synthetic grandmaster
PTP driving the media clock	**Working** — RTP timestamps are PTP time
SAP discovery	Not started
Tray packaging	Not started
Desktop client station	Not started
Opus/WebRTC leg for phones and browsers	Not started
Hardware endpoint	Not started

Two transports, and why there have to be two

Wired clients and hardware panels get real **AES67**: L24 RTP at 1 ms packet time, PTP-locked, one stream per key.

Phones, browsers and anything off-LAN get **Opus over WebRTC** instead. **This is not a shortcut — it is forced.** No phone exposes PTP, and **wifi multicast is transmitted at the basic rate with no retries**, so AES67 over wifi does not degrade gracefully, it falls apart. The server is the bridge between the two domains.

The mix engine is deliberately unaware of any of this. It emits one stream per key; the fold-down to a stereo Opus mix happens downstream. **One mixing implementation, one set of tests.**

Mix-minus, and the two things that make it exact

Every member of a partyline hears the bus minus their own voice. Read literally that is a separate sum per member. It is not necessary:

```
contribution[key]  = mic × talk_ramp × bus_trim      (once per key)
bus[p]             = Σ contribution[keys on p]        (once per bus)
feed[key on p]     = bus[p] - contribution[key]       (once per key)
```

The subtraction is exact, not approximate, because it removes precisely the buffer that was added — same samples, same gain, same block. Two things follow, and both are load-bearing:

- **Every input must already be aligned into the server's clock domain.** An engine fed unaligned inputs **leaks the talker back into their own ear, quietly and unfixably.**
- **The bus trim is folded into each contribution, not applied to the summed bus.** Trimming after the sum would subtract an untrimmed buffer from a trimmed one and leave `(1 - trim)` of the talker in their own ear — **inaudible at unity gain, and a mystery at any other setting.**

The output limiter sits **per stream, after the subtraction**, for the same reason.

What it costs, and what the real constraint is

Measured on an M-series Mac, release build, 32 endpoints × 10 keys = 320 streams, every key talking at once — the worst case the engine can be put in:

```
1.000 s of audio mixed in 34.4 ms   (29x realtime, 3.4% of one core)
```

Mixing is not the constraint. Packet rate is. Those 320 streams at 1 ms are **320,000 packets per second outbound**, and unbatched sending measures at 292,000 packets/sec on one thread — just *under* what a 32-endpoint system implies.

That is the measurement that turns per-tick send batching from an optimisation into a requirement. Bandwidth is the lesser problem, at roughly 1.6 Mbit/s per stream.

Three transport details that fail quietly

The jitter buffer is indexed by RTP timestamp, not by sequence number. The timestamp *is* the media clock; the sequence number is only a counter. A sequence-indexed ring scatters audio into the wrong slots at the timestamp rollover — **once every ~24.8 hours at 48 kHz, so reliably mid-show.**

Multicast groups are allocated sequentially from one base. IPv4 multicast copies only the **low 23 bits** of the address into the Ethernet MAC, so `239.69.1.1` and `239.197.1.1` become the same MAC — and an IGMP-snooping switch delivers both to anyone who joined either. Sequential allocation keeps those bits distinct.

The buffer's capacity is not its latency. Depth sets the delay; capacity is headroom for the case where the receiving thread is descheduled and the socket then hands over 40 packets at once. A

ring sized to the depth throws away audio it had already received.

Addressing, and the one thing to fix before deploying

Multicast groups are derived from **(endpoint index, key slot)**, not from the engine's stream index. The engine numbers streams sequentially, so adding a key to endpoint 0 shifts every stream after it — and addressing off that number **would silently re-point every multicast group in the building each time somebody added a key in the UI.**

Deriving from identity means adding a key moves nothing else.

> **Reordering or deleting *endpoints* still shifts things.** The real fix is to persist an allocation per endpoint id, and **it is worth doing before anyone deploys this.**

If something is wrong

Symptom	Cause
A talker hears themselves	Inputs are not aligned into the server's clock domain, or a trim is being applied after the sum rather than into each contribution.
Audio scatters after about a day	A jitter buffer indexed by sequence number rather than timestamp.
Two unrelated streams arrive at one endpoint	Multicast addresses sharing their low 23 bits.
Bursts of audio go missing	Buffer capacity sized to the depth rather than to the burst.
It falls apart over wifi	AES67 over wifi does. That is what the Opus/WebRTC leg is for — and it is not built yet.
Endpoints re-point themselves after an edit	Endpoint reorder or delete. See above.