vq-core ZeroMQ API

Sevana Ou PTE

68 Circular Road #02-01 Singapore 049422

vq-core ZeroMQ API

`vq-core` publishes every event it observes — capture statistics, live RTP streams, per-interval and final quality reports, SIP call signalling, optional in-band decoded audio, and process-level metrics — on a ZeroMQ PUB socket. This is the same feed that `vq-db` consumes to populate the database and the dashboard; any third-party tool can subscribe in parallel without affecting the existing consumers.

Overview

Property	Value
Transport	ZeroMQ PUB / SUB over TCP
Default endpoint	`tcp://127.0.0.1:9125` (configurable via `server.zeromq-bind` / `server.zeromq-port`)
Wire format	One Protocol Buffers `messages.Event` per ZMQ frame
Schema	`source/protobuf/capturemessage.proto` (proto3, package `messages`)
Authentication	None — front with stunnel / IP allow-lists / a CURVE proxy if you expose it off-host
Reliability	Best-effort. The publisher uses `ZMQ_DONTWAIT`; when the send-side HWM is reached the dropped frames are logged at ERROR level on the `system_bus` log subsystem

The publisher binds in `report_producer::open()` (see `source/vq_core/protobuf/message_bus.cpp`); the matching reference consumer is `message_bus_client` in `source/vq_db/message_bus_client.cpp`.

All events are serialized and sent from a single dedicated publish thread, not from the capture / DPDK poll cores. The capture-thread `Processor` runs network analysis inline (run-to-completion) and builds the interval report on the capture thread, but the actual ZMQ send is handed to the publish thread so a slow subscriber can never back-pressure a poll core (see `THREADING-REDESIGN.md`). Wire-format-wise this is invisible to subscribers; it only means event ordering is the publish thread's order, not strictly the capture order.

Configuring the bind

By default the publisher binds to `127.0.0.1` only — no off-host subscriber can connect. Two YAML keys under `server:` change this:

```yaml
server:
    zeromq-bind:  127.0.0.1     # default; "0.0.0.0" exposes the bus on all interfaces
    zeromq-port:  9125          # default
```

The bus has **no authentication** — anyone who can reach the socket can subscribe to every reported stream, which includes SIP call IDs and peer URIs. When `zeromq-bind` is anything other than `127.0.0.1` / `::1` / `localhost`, `vq-core` logs a startup banner reminding operators to restrict access at the firewall. Typical safe setups:

- Single-host install — leave at the default.
- Multi-host install — bind to a specific management-network address (e.g. `10.0.0.5`), and either firewall the port to known subscribers or terminate it behind a TCP-level proxy with mTLS.

Wire format

Each ZMQ message is exactly one serialized `messages.Event`. `Event` is a discriminated union — at most one of the eleven fields below will be set per message. Subscribers should look at which field is populated and dispatch accordingly (Python: `event.WhichOneof(...)` if you change the schema to `oneof`, or simply `event.HasField("stream_report")`).

```protobuf
message Event {
    Stream             stream_report     = 1;   // per-interval quality report
    StreamStart        stream_start      = 2;   // a new RTP stream was detected
    StreamFinish       stream_finish     = 3;   // a stream finished (final report inside)
    CaptureStats       capture_stats     = 4;   // per-interface capture totals
    StreamGhost        stream_ghost      = 5;   // a flow finished before reaching the analysis threshold
    InstanceStatistics instance_stats    = 6;   // process-level health and identity (timer-driven)

    SipCallStart       sip_call_start    = 7;   // SIP call established (setup response seen)
    SipCallEnd         sip_call_end      = 8;   // SIP call ended (BYE / timeout)
    SipReinvite        sip_reinvite      = 9;   // reINVITE / UPDATE on an existing call
    SipCallFailed      sip_call_failed   = 10;  // INVITE never reached a setup code

    StreamAudio        stream_audio      = 11;  // in-band decoded audio (only with publish-audio)
}
```

The first six members are the media/quality and process-health feed. Members 7–10 are the **SIP signalling** feed, emitted from the SIP processor as calls are established, end, are renegotiated, or fail. Member 11 carries **decoded audio in-band** and is only ever sent when the `publish-audio` option is enabled.

All numeric fields are proto3 scalars (default value = “unset”). Field numbers are stable; new event kinds were appended, so a subscriber that only switches on the six original members keeps working and silently ignores the rest. Producer-side cadence is summarized in *Event lifecycle*; the full schema is listed at the end.

Endpoint

`vq-core` publishes on the port set by `server.zeromq-port` in the YAML config (default `9125`) and binds to the address set by `server.zeromq-bind` (default `127.0.0.1`). The resulting socket URL is:

```
tcp://<server.zeromq-bind>:<server.zeromq-port>
```

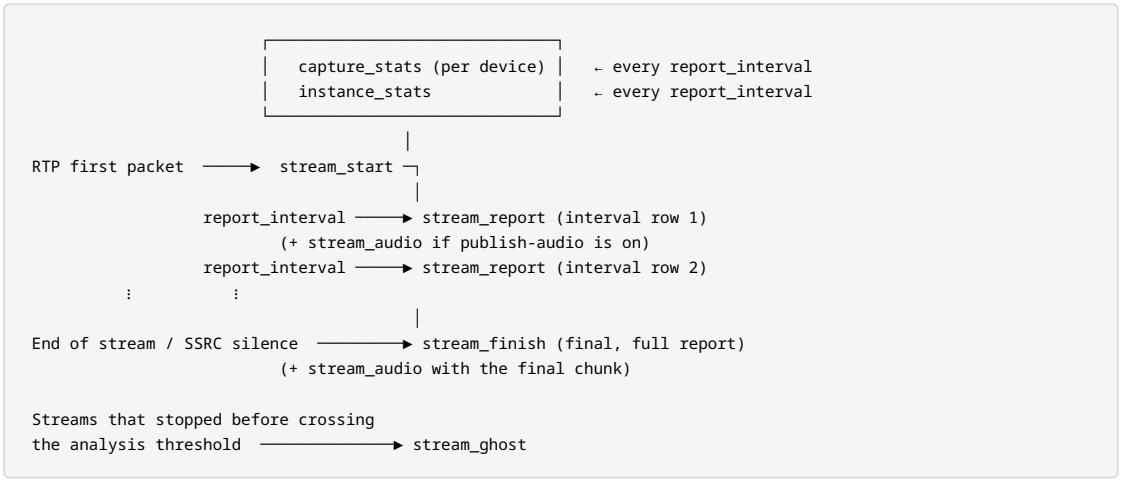
Subscribers connect with `ZMQ_SUB` and an empty subscription string (the publisher does not currently set a topic prefix), so subscribing to `""` receives every event:

```python
import zmq
ctx = zmq.Context.instance()
sub = ctx.socket(zmq.SUB)
sub.connect("tcp://127.0.0.1:9125")
sub.setsockopt_string(zmq.SUBSCRIBE, "")   # all events
```

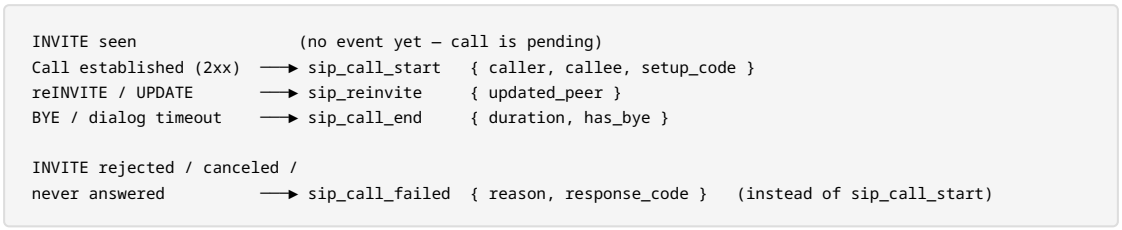
A 50 ms poll loop is what `vq-db` itself uses; see the sample subscriber under *Examples*.

Event lifecycle

For a typical RTP stream observed by `vq-core` the sequence of events on the bus is:



In parallel, the SIP signalling feed tracks the call the media belongs to:



`stream_report` carries the *current snapshot* of the rolling stream metrics (current MOS, current jitter, average/min/max so far). `stream_finish` carries the *final* version of the same `Stream` message after the stream ends.

`instance_stats` is emitted from a dedicated stats timer (`onIntervalElapsed`), so its cadence is `server.instance-report-interval` (default **1000 ms**) — **not** `media.report-interval`. `capture_stats` is emitted by the capturers (via the processing pool's `onCapturerStats`), independently of the report timer. The SIP events are emitted as the corresponding signalling is observed (not on any timer), so they are not tied to a report interval and may interleave with media events for the same call in either order.

Common types

`IpAddress`

```
family   IPv4 | IPv6        // enum
relayed  bool               // true for relayed/HEP-recovered addresses
ip       bytes              // 4 bytes for IPv4, 16 bytes for IPv6 (network byte order)
port     uint32
```

The `ip` field is **raw bytes** — clients must format it themselves. Python:

```
        import ipaddress
        addr = ipaddress.ip_address(bytes(msg.ip))   # → "192.168.1.10" or "fe80::1"
```

`StreamId`

The stable identifier the rest of the system keys on:

```
src   IpAddress     // sender address
dst   IpAddress     // receiver address
ssrc  uint32        // RTP SSRC
uuid  string        // textual LinkId (matches `stream_id` in the vq-db REST API)
```

`uuid` is the same value `vq-db`'s `/stats`, `/streamhistory` and `/report` endpoints expose as `stream_id` / `ref_id`. Use it to correlate live ZMQ events with later REST queries.

`Stream`

The voice-quality record. Used as the body of `stream_report` and embedded in `stream_finish.report`.

Field	Type	Notes
`stream_id`	StreamId	
`rtp_packet_counter`	uint64	RTP packets observed in the stream
`rtp_illegal_packet_counter`	uint64	malformed / non-decodable RTP
`rtp_lost_packet_counter`	uint64	gaps inferred from sequence numbers
`jitter`	MetricStats	Milliseconds (the producer scales seconds × 1000). See `MetricStats` below: `current` is the latest interval value, `min` / `max` / `avg` cover the whole stream so far.
`codecname`	string	Name of the most-used codec, empty if unknown
`rtt`	MetricStats	Milliseconds. Round-trip delay. `current` is the latest value; `min` / `max` / `avg` cover the whole stream so far. (Was the flat `rttdelay*` scalars before the MetricStats refactor.)
`start_timestamp` / `finish_timestamp`	uint64	Unix milliseconds for **this report's window**: in an interval (`stream_report`) it is the chunk `[start, finish]` ; in the final `stream_finish.report` it spans the whole stream. The window duration is `finish_timestamp - start_timestamp` (there is no separate `duration` field).
`start_timestamp_stream`	uint64	Unix milliseconds of the whole-stream start (first media packet). Constant across every interval report for a stream; equals `start_timestamp` in the final report.
`duration_audio`	float	Seconds of audio actually decoded (≤ the stream's wall-clock duration)
`mos_network`	MetricStats	Network MOS (G.107 E-model) — `current` plus running `min` / `max` / `avg`
`mos_sevana`	MetricStats	Sevana MOS (PVQA) — only populated when PVQA actually ran on this stream
`sevana_rfactor`	float	0–100, percent of "good" intervals as scored by PVQA
`interval_report`	string	CSV-style PVQA detector dump for the interval (only set when Sevana MOS is)
`user_data`	string	Free-form extension field (e.g. decoded-audio path when audio dumping is enabled)
`sip`	SipInfo	

Field	Type	Notes
		SIP correlation (`call_id` / `peer1` / `peer2`). Sub-message unset if the call was not seen on SIP. See `SipInfo` below.
`echolist[]`	EchoPeak	Optional list of detected echo peaks: `{ time, level, length }` (seconds / dB / seconds)
`dtx_info`	DtxInfo	AMR-NB/WB and EVS frame counters derived from the RTP payload header without decoding. Unset for non-AMR/EVS codecs and for SRTP. In `stream_report` the counts are for the latest interval; the embedded `stream_finish.report` carries whole-stream totals.

Jitter, RTT and both MOS scores share the `MetricStats` shape (see below). The `min` / `max` / `avg` members are the range/average since the start of the stream; `current` is the latest interval value.

`MetricStats`

Aggregate of a single metric tracked over the stream's lifetime, reused for `jitter`, `rtt`, `mos_network` and `mos_sevana`. Left at proto3 defaults (all zero, sub-message unset) when the underlying metric was never initialized.

Field	Type	Notes
`min`	float	minimum since stream start
`max`	float	maximum since stream start
`avg`	float	average since stream start
`current`	float	latest / instantaneous value

`SipInfo`

SIP correlation, sub-message of `Stream.sip`. Unset (proto3 default) when the stream was not matched to a SIP dialog.

Field	Type	Notes
`call_id`	string	SIP Call-ID
`peer1`	string	first party AOR (`mSip.mPeerA`)
`peer2`	string	second party AOR (`mSip.mPeerB`)

`DtxInfo`

AMR/EVS frame statistics, sub-message of `Stream.dtx_info`. Derived from the RTP payload header (not the decoded audio), so it is populated even for network-MOS-only streams; it stays unset for SRTP (encrypted payload) and non-AMR/EVS codecs.

Field	Type	Notes
`count_sid`	uint64	SID / comfort-noise frames
`count_dtx`	uint64	NO_DATA frames (DTX silence, nothing transmitted)
`count_total`	uint64	total AMR/EVS frames inspected (any class)

If the stream was carried over SRTP with `media.use-srtp: true`, only the network-side fields (`*_packet_counter`, `jitter`, `rtt`, `mos_network`, `codecname`, `sip_*`) are populated; Sevana MOS / R-factor / detector report and `dtx_info` stay at their proto3 defaults (zero / empty / unset).

`CaptureStats`

Per-interface counters, emitted by the capturers (not on the report timer).

```
device                  string   // capture interface ("eth0", "lo", "hep", or "dpdk:<port>")
total_packet_counter    uint64   // packets seen since process start
rtp_packet_counter      uint64   // packets classified as RTP
sip_packet_counter      uint64   // packets classified as SIP
truncated_packet_counter uint64  // packets dropped because the snaplen cut them short
src_port_range          string   // configured filter, "" if unset
dst_port_range          string   // configured filter, "" if unset
dropped_hw_packet_counter uint64 // DPDK only: packets the NIC dropped (RX queues full); 0 elsewhere
erroneous_packet_counter  uint64 // DPDK only: erroneous packets (ierrors); 0 elsewhere
mbuf_alloc_failed_counter uint64 // DPDK only: RX mbuf allocation failures; 0 elsewhere
```

Which fields are populated depends on the source event. The standalone `capture_stats` event (`onCapturerStats`) sets only `device`, `total_packet_counter`, `rtp_packet_counter`, `sip_packet_counter`, and `truncated_packet_counter`; the remaining fields (`src_port_range`, `dst_port_range`, and the three NIC-level counters below) stay at their proto3 defaults there. The port-range and NIC-loss fields are populated on the `InstanceStatistics.capturers` path instead.

The three `*_hw_*` / `erroneous` / `mbuf_alloc_failed` counters are NIC-level capture-loss counters, populated by the DPDK capturer from `rte_eth_stats` and `0` on the libpcap / HEP paths (which have no equivalent). They are the

signal that the capture path is overwhelmed — packets lost before vq-core's pipeline ever saw them. Fields 8–10 were added additively, so older consumers that don't read them are unaffected.

`StreamStart` / `StreamFinish` / `StreamGhost`

```
StreamStart  { stream_id }
StreamFinish { stream_id, report }                // report is a full Stream
StreamGhost  { stream_id, packet_counter }        // stream finished without crossing analysis threshold
```

`StreamGhost.packet_counter` is the total number of RTP packets ever seen for the flow before it was dropped as a ghost — i.e. a flow whose packet count stayed below `RTP_STREAM_DETECT_LIMIT` and never graduated to full reporting. It lets a subscriber distinguish a one-or-two-packet probe from a near-threshold flow.

`StreamAudio`

Decoded audio for a stream, delivered **in-band** so a subscriber never needs filesystem access to `vq-core`. It is emitted **only** when the `publish-audio` option is enabled; one `StreamAudio` message accompanies each interval/final `Stream` report that actually produced audio (it follows the matching `stream_report` / `stream_finish` on the wire). Network-MOS-only streams and offline file analysis carry no audio, so no `StreamAudio` is sent for them.

Field	Type	Notes
`stream_id`	StreamId	same id as the accompanying `Stream` report
`wav`	bytes	a complete, self-contained WAV file (canonical PCM16 mono header + samples)
`start_timestamp`	uint64	Unix milliseconds — start of this audio chunk (matches the report window)
`finish_timestamp`	uint64	Unix milliseconds — end of this audio chunk

Because `wav` is a full WAV (header included), a subscriber can write the bytes straight to a `.wav` file or feed them to a player without any extra metadata. Each message is one interval's worth of audio; collecting the chunks reconstructs the call. Audio frames are typically much larger than the other events, so size the subscriber's receive HWM accordingly.

`InstanceStatistics`

Process-level snapshot of `vq-core` itself, emitted every `server.instance-report-interval` (default 1000 ms).

```
id                            string   // server.instance.id
name                          string   // server.instance.name
server_time                   string   // millisecond-precision local timestamp
pvqa_instance_counter         uint64   // current PVQA instances in flight
pvqa_processed_seconds        uint64   // accumulated audio seconds analyzed by PVQA
uptime_seconds                uint64
version                       string   // "Server vM.m.s , build number BBBB" (and PVQA version)
active_decoder_counter        uint64   // live StreamDecoder objects right now
total_decoder_counter         uint64   // total decoders created since startup
active_audio_decoder_counter  uint64   // streams doing audio decode + PVQA right now (subset of
active_decoder_counter)
capturers[]                   CaptureStats
sip_call_counter              uint64   // live SipManager Call objects right now

// reSIProcate object counts (leak indicators)
resip_message_counter         uint64   // every resip::Message-derived object alive
resip_sip_message_counter     uint64   // the SIP-specific subset

// Allocator (tcmalloc) memory figures — 0 when not built against tcmalloc
mem_allocated_bytes           uint64   // live application usage
mem_heap_size                 uint64   // total mapped from the OS
mem_pageheap_free_bytes       uint64   // free-but-mapped memory the allocator retains

// Allocator load — 0 unless track-allocations is enabled on a tcmalloc build
mem_alloc_count               uint64   // cumulative allocations since startup
mem_free_count                uint64   // cumulative frees since startup
mem_allocs_per_sec            double   // per-second allocation rate over the last interval
mem_frees_per_sec             double   // per-second free rate over the last interval

// Process CPU usage (whole process, all threads), from getrusage(RUSAGE_SELF)
cpu_user_seconds              double   // cumulative user-space CPU time
cpu_system_seconds            double   // cumulative kernel CPU time on the process's behalf
cpu_usage_percent             double   // busy fraction over the last interval, % of a SINGLE core (can exceed
100)
```

`active_audio_decoder_counter` is the count of streams currently running the expensive audio-decode + PVQA chain — a subset of `active_decoder_counter`. It shows how full the audio set is relative to the SIP track list and the `max-streams-audio` cap, and reads `0` in the network-MOS-only default where no stream is decoded.

`sip_call_counter` is the number of `Call` objects `vq-core` currently holds. It grows with call volume and only drops as calls are garbage-collected, so a value that keeps climbing while `active_decoder_counter` stays flat is a sign that `Call` objects are being retained (e.g. calls that never received a BYE). The two `resip_*_counter` fields are the analogous leak indicators for parsed reSIProcate messages.

The `mem_*` figures come from tcmalloc and are `0` on a non-tcmalloc build; the `mem_alloc*` / `mem_free*` load figures are additionally `0` unless allocation tracking (`track-allocations`) is on. `cpu_usage_percent` is expressed top-

style as a percentage of a single core, so e.g. `350` means ~3.5 cores busy; divide by the core count for a 0–100 machine-wide figure. It reads `0` on the first report (no prior interval to diff against).

SIP signalling events

Independent of the media feed, `vq-core` publishes the SIP call lifecycle it reconstructs from signalling. These are emitted by the SIP processor as the events happen (not on the report timer), so they can arrive before, between, or after the media events for the same call. Correlate them with media via `Stream.sip.call_id` (the `SipInfo.call_id` on the quality report) — the SIP events key on the same `call_id`.

All timestamps below are **Unix milliseconds** (the producer divides the internal microsecond clock by 1000), matching `Stream.start_timestamp`.

`SipPeer`

One party of a SIP dialog, embedded in the call events as `caller` / `callee` / `updated_peer`.

Field	Type	Notes
`aor`	string	From/To AOR, e.g. `sip:alice@example.com`
`user_agent`	string	`User-Agent` / `Server` header if present
`rtp_addresses`	IpAddress[]	media addresses advertised in SDP (`c=` / `m=`)
`codecs`	string	resolved codec settings, one line (cf. `Stream.codecname`)
`ssrc`	uint32	SSRC this peer is expected to send (`0` if unknown)

`SipDirection`

Enum on the end / reINVITE events: `SIP_DIR_UNKNOWN = 0`, `SIP_DIR_CALLER_TO_CALLEE = 1`, `SIP_DIR_CALLEE_TO_CALLER = 2`.

`SipCallStart`

Emitted once a call is established (a 2xx setup response is seen). Mutually exclusive with `SipCallFailed` for a given Call-ID.

Field	Type	Notes
`call_id`	string	SIP Call-ID
`timestamp`	uint64	establishment time (ms)
`caller`	SipPeer	originator (`mPeers.first`)
`callee`	SipPeer	answerer (`mPeers.second`)
`setup_code`	uint32	response code that established the call (default `200`)
`invite_timestamp`	uint64	first INVITE observed (ms) — subtract from `timestamp` for post-dial delay

`SipCallEnd`

Emitted on BYE (or when the dialog is reaped without one).

Field	Type	Notes
`call_id`	string	SIP Call-ID
`timestamp`	uint64	BYE time (ms)
`bye_direction`	SipDirection	which party sent the BYE
`response_codes`	uint32[]	all response codes seen over the dialog
`duration`	uint64	milliseconds, `BYE - setup` (set only when both are known)
`has_bye`	bool	`false` ⇒ ended by timeout / dangling, not an actual BYE

`SipReinvite`

A reINVITE (also covers UPDATE — same code path), e.g. a hold/resume or codec renegotiation.

Field	Type	Notes
`call_id`	string	SIP Call-ID
`timestamp`	uint64	event time (ms)
`is_request`	bool	request vs response
`direction`	SipDirection	
`updated_peer`	SipPeer	the peer whose SDP/media was renegotiated
`raw_message`	string	optional raw SIP text

`SipCallFailed`

Emitted when an INVITE never reaches a setup code. Mutually exclusive with `SipCallStart` for a Call-ID.

Field	Type	Notes
`call_id`	string	SIP Call-ID
`timestamp`	uint64	when the failure was determined (ms)
`caller`	SipPeer	originator
`callee`	SipPeer	answerer
`invite_timestamp`	uint64	first INVITE observed (ms)
`reason`	enum	`REASON_UNKNOWN/REJECTED/CANCELED/TIMEOUT` — see below
`response_code`	uint32	final SIP response code (e.g. `486`, `603`); `0` if none
`reason_phrase`	string	optional reason phrase from the status line

`reason` is `REASON_REJECTED` for a final non-2xx (4xx/5xx/6xx), `REASON_CANCELED` when the caller sent CANCEL before answer, and `REASON_TIMEOUT` is reserved (dialog reaped by gc with no final response — not emitted yet).

Examples

Quick check — print everything from the bus

This is the bundled sample (`workdir/zeromq_monitor.py`; also at `scripts/test/zeromq_monitor.py`) — pipe it through `less` or grep for the message kind you care about:

```
python3 workdir/zeromq_monitor.py \
    --endpoint tcp://127.0.0.1:9125 --format text
```

It prints each event in pretty-printed protobuf text form, with raw `IpAddress.ip` bytes rendered as a dotted (IPv4) or colon (IPv6) string.

Minimal subscriber

```
import ipaddress
import zmq
from capturemessage_pb2 import Event   # protoc-generated module

ctx = zmq.Context.instance()
sub = ctx.socket(zmq.SUB)
sub.connect("tcp://127.0.0.1:9125")
sub.setsockopt_string(zmq.SUBSCRIBE, "")  # subscribe to everything

while True:
    raw = sub.recv()
    event = Event()
    event.ParseFromString(raw)

    if event.HasField("stream_report"):
        s = event.stream_report
        src = ipaddress.ip_address(bytes(s.stream_id.src.ip))
        dst = ipaddress.ip_address(bytes(s.stream_id.dst.ip))
        print(f"[interval] {s.stream_id.uuid} {src}:{s.stream_id.src.port} -> "
              f"{dst}:{s.stream_id.dst.port}  codec={s.codecname or '?'}  "
              f"mos_net={s.mos_network.current:.2f}  mos_sevana={s.mos_sevana.current:.2f}  "
              f"jitter_ms={s.jitter.current:.2f}")
    elif event.HasField("stream_start"):
        print(f"[start]    {event.stream_start.stream_id.uuid}")
    elif event.HasField("stream_finish"):
        s = event.stream_finish.report
        print(f"[finish]   {s.stream_id.uuid}  "
              f"loss={s.rtp_lost_packet_counter}/{s.rtp_packet_counter}  "
              f"r_factor={s.sevana_rfactor}")
    elif event.HasField("capture_stats"):
        c = event.capture_stats
        print(f"[capture]  {c.device}: rtp={c.rtp_packet_counter} "
              f"sip={c.sip_packet_counter} truncated={c.truncated_packet_counter}")
    elif event.HasField("instance_stats"):
        i = event.instance_stats
        print(f"[instance] {i.id}/{i.name}  uptime={i.uptime_seconds}s  "
              f"active_decoders={i.active_decoder_counter}  "
              f"audio_decoders={i.active_audio_decoder_counter}  "
              f"cpu={i.cpu_usage_percent:.0f}%")
    elif event.HasField("sip_call_start"):
        c = event.sip_call_start
        print(f"[sip-start]  {c.call_id}  {c.caller.aor} -> {c.callee.aor}  "
              f"code={c.setup_code}")
    elif event.HasField("sip_call_end"):
        c = event.sip_call_end
        print(f"[sip-end]    {c.call_id}  duration_ms={c.duration}  has_bye={c.has_bye}")
    elif event.HasField("sip_call_failed"):
        c = event.sip_call_failed
        print(f"[sip-fail]   {c.call_id}  reason={c.reason}  code={c.response_code}")
    elif event.HasField("sip_reinvite"):
        c = event.sip_reinvite
        print(f"[sip-reinv]  {c.call_id}  peer={c.updated_peer.aor}")
    elif event.HasField("stream_audio"):
        a = event.stream_audio
        # `a.wav` is a complete WAV file — write it straight to disk.
        with open(f"{a.stream_id.uuid}_{a.start_timestamp}.wav", "wb") as f:
            f.write(a.wav)
        print(f"[audio]    {a.stream_id.uuid}  {len(a.wav)} bytes")
```

Generating the protobuf bindings

The schema is in `source/protobuf/capturemessage.proto` (proto3, package `messages`). To produce a Python module for your subscriber:

```
protoc --python_out=. --proto_path=source/protobuf \
       source/protobuf/capturemessage.proto
# → capturemessage_pb2.py
```

Equivalent invocations exist for C++, Go, Java, etc. — see the Protobuf documentation for the right generator flag.

Bridging the bus to another transport

A simple use case is rebroadcasting the feed to a Kafka topic or to a syslog collector. Because each ZMQ frame is one fully-formed protobuf message, you can forward `raw` directly without re-serializing:

```
producer.send("vq-core.events", value=raw)   # Kafka client
```

Multi-agent fan-in

In a multi-probe deployment several `vq-core` instances publish their own buses on distinct hosts; `vq-db` (or any aggregator) connects to each of them as a subscriber and demuxes by `instance_stats.id` / `Stream.stream_id.uuid`. The `server.instance.id` configured in YAML on each agent is what appears in every published `instance_stats` and in the JSON returned by `vq-db`'s `/instance_list`.

Operational notes

- **HWM drops.** When a subscriber is too slow (or paused), ZMQ buffers up to its high-water mark on the publisher side and then drops new frames. The publisher logs `Dropped <N>-byte report: ZMQ send HWM reached` at ERROR level on the `system_bus` log subsystem; alert on this to catch silent data loss.
- **Topic prefix.** None today — the publisher sends frames without a topic envelope. Every subscriber receives every event. If you need to filter on the wire, the cheapest option is to do it client-side after `ParseFromString`.
- **Schema evolution.** The current schema is proto3, so adding new fields is always backward-compatible. Removing or repurposing field numbers is not — coordinate with anyone consuming the bus.

Schema reference

The current schema is reproduced verbatim from `source/protobuf/capturemessage.proto`:

```
syntax="proto3";
package messages;

message IpAddress {
    enum Family { IPv4 = 0; IPv6 = 1; }
    Family family   = 1;
    bool   relayed  = 2;
    bytes  ip       = 3;
    uint32 port     = 4;
}

message StreamId {
    IpAddress src   = 1;
    IpAddress dst   = 2;
    uint32    ssrc  = 3;
    string    uuid  = 4;
}

message CaptureStats {
    string device                    = 1;
    uint64 total_packet_counter      = 2;
    uint64 rtp_packet_counter        = 3;
    uint64 sip_packet_counter        = 4;
    uint64 truncated_packet_counter  = 5;
    string src_port_range            = 6;
    string dst_port_range            = 7;
    uint64 dropped_hw_packet_counter = 8;   // DPDK: dropped by NIC HW (RX queues full); 0 elsewhere
    uint64 erroneous_packet_counter  = 9;   // DPDK: erroneous packets (ierrors); 0 elsewhere
    uint64 mbuf_alloc_failed_counter = 10;  // DPDK: RX mbuf allocation failures; 0 elsewhere
}

message MetricStats {
    float min     = 1;
    float max     = 2;
    float avg     = 3;
    float current = 4;   // last / instantaneous value
}

message Stream {
    StreamId stream_id                  = 1;

    uint64 rtp_packet_counter           = 2;
    uint64 rtp_illegal_packet_counter   = 3;
    uint64 rtp_lost_packet_counter      = 4;
    uint64 rtp_duplicated_counter       = 5;

    string codecname                    = 6;

    uint64 start_timestamp              = 7;   // this report's window start (chunk start for intervals)
    uint64 finish_timestamp             = 8;   // window duration = finish_timestamp - start_timestamp
    float  duration_audio               = 9;
    uint64 start_timestamp_stream       = 20;  // whole-stream start; constant across intervals

    string user_data                    = 10;
    float  sevana_rfactor               = 11;
    string interval_report              = 12;

    message SipInfo { string call_id = 1; string peer1 = 2; string peer2 = 3; }
    SipInfo     sip                     = 13;

    message EchoPeak { float time = 1; float level = 2; float length = 3; }
    repeated EchoPeak echolist          = 14;

    message DtxInfo { uint64 count_sid = 1; uint64 count_dtx = 2; uint64 count_total = 3; }
    DtxInfo     dtx_info                = 15;

    MetricStats jitter                  = 16;   // milliseconds
    MetricStats rtt                     = 17;   // milliseconds
    MetricStats mos_network             = 18;
    MetricStats mos_sevana              = 19;
}

message StreamStart  { StreamId stream_id = 1; }
message StreamFinish { StreamId stream_id = 1; Stream   report    = 2; }
message StreamGhost  { StreamId stream_id = 1; uint64   packet_counter = 2; }  // packets seen before drop
```

```
// Decoded audio for a stream, in-band; only sent when publish-audio is enabled.
message StreamAudio {
    StreamId stream_id      = 1;
    bytes    wav            = 2;   // complete WAV (header + PCM16 mono)
    uint64   start_timestamp = 3;  // UNIX ms of this chunk
    uint64   finish_timestamp = 4; // UNIX ms of this chunk
}

message InstanceStatistics {
    string id                           = 1;
    string name                         = 2;
    string server_time                  = 3;
    uint64 pvqa_instance_counter        = 4;
    uint64 pvqa_processed_seconds       = 5;
    uint64 uptime_seconds               = 6;
    string version                      = 7;
    uint64 active_decoder_counter       = 8;
    uint64 total_decoder_counter        = 9;
    uint64 active_audio_decoder_counter = 24;  // streams decoding audio+PVQA now (subset of #8)

    repeated CaptureStats   capturers   = 10;

    uint64 sip_call_counter             = 11;  // live SipManager Call objects

    uint64 resip_message_counter        = 12;  // every resip::Message-derived object alive
    uint64 resip_sip_message_counter    = 13;  // SIP-specific subset

    uint64 mem_allocated_bytes          = 14;  // tcmalloc: live usage (0 if not tcmalloc)
    uint64 mem_heap_size                = 15;
    uint64 mem_pageheap_free_bytes      = 16;
    uint64 mem_alloc_count              = 17;  // 0 unless track-allocations is on
    uint64 mem_free_count               = 18;
    double mem_allocs_per_sec           = 19;
    double mem_frees_per_sec            = 20;

    double cpu_user_seconds             = 21;  // getrusage(RUSAGE_SELF)
    double cpu_system_seconds           = 22;
    double cpu_usage_percent            = 23;  // % of a SINGLE core over last interval (can exceed 100)
}

// ---- SIP signalling feed ----

enum SipDirection {
    SIP_DIR_UNKNOWN          = 0;
    SIP_DIR_CALLER_TO_CALLEE = 1;
    SIP_DIR_CALLEE_TO_CALLER = 2;
}

message SipPeer {
    string aor                      = 1;   // e.g. sip:alice@example.com
    string user_agent               = 2;
    repeated IpAddress rtp_addresses = 3;   // media addresses from SDP (c=/m=)
    string codecs                   = 4;
    uint32 ssrc                     = 5;   // 0 if unknown
}

// All SIP-event timestamps are UNIX milliseconds.
message SipCallStart {
    string  call_id          = 1;
    uint64  timestamp        = 2;   // establishment time
    SipPeer caller           = 3;
    SipPeer callee           = 4;
    uint32  setup_code       = 5;   // response code that established the call (default 200)
    uint64  invite_timestamp = 6;   // first INVITE observed (for post-dial delay)
}

message SipCallEnd {
    string  call_id           = 1;
    uint64  timestamp         = 2;   // BYE time
    SipDirection bye_direction = 3;
    repeated uint32 response_codes = 4;
    uint64  duration          = 5;   // ms, BYE - setup (if both known)
    bool    has_bye           = 6;   // false => ended by timeout, not BYE
}

message SipReinvite {                // also covers UPDATE
    string  call_id          = 1;
```

```
    uint64  timestamp        = 2;
    bool    is_request       = 3;
    SipDirection direction   = 4;
    SipPeer updated_peer     = 5;
    string  raw_message      = 6;   // optional raw SIP text
}

message SipCallFailed {               // mutually exclusive with SipCallStart per Call-ID
    string  call_id          = 1;
    uint64  timestamp        = 2;
    SipPeer caller           = 3;
    SipPeer callee           = 4;
    uint64  invite_timestamp = 5;

    enum Reason {
        REASON_UNKNOWN  = 0;
        REASON_REJECTED = 1;   // final non-2xx (4xx/5xx/6xx)
        REASON_CANCELED = 2;   // caller CANCELed before answer
        REASON_TIMEOUT  = 3;   // reserved: reaped by gc, not yet emitted
    }
    Reason  reason           = 6;
    uint32  response_code    = 7;   // final SIP code (e.g. 486, 603); 0 if none
    string  reason_phrase    = 8;
}

message Event {
    Stream             stream_report     = 1;
    StreamStart        stream_start      = 2;
    StreamFinish       stream_finish     = 3;
    CaptureStats       capture_stats     = 4;
    StreamGhost        stream_ghost      = 5;
    InstanceStatistics instance_stats    = 6;

    SipCallStart       sip_call_start    = 7;
    SipCallEnd         sip_call_end      = 8;
    SipReinvite        sip_reinvite      = 9;
    SipCallFailed      sip_call_failed   = 10;

    StreamAudio        stream_audio      = 11;
}
```

Control channel — the SIP “to track” list

By default `vq-core` computes only the cheap **network MOS** for every RTP stream. The expensive audio decode + PVQA (Sevana MOS) chain is started for a stream **only when one of its SIP peers (caller or callee AOR) matches an entry in an in-memory “to track” list.** That list is mutated at runtime over a second ZeroMQ socket and is **not persisted** — it starts empty on every launch. (Offline `.pcap` analysis is unaffected: it always decodes every stream.)

Property	Value
Transport	ZeroMQ REQ / REP over TCP (request/reply, one `CommandAck` per `Command`)
Default endpoint	`tcp://127.0.0.1:9127` (configurable via `server.zeromq-control-bind` / `server.zeromq-control-port`)
Wire format	One `messages.Command` request → one `messages.CommandAck` reply
Authentication	None — same caveats as the publisher; loopback by default

The socket is bound in `command_consumer` (`source/vq_core/protobuf/command-consumer.cpp`) and is only started when capture is active.

Matching semantics

- A pattern matches **case-insensitively as a substring** of either peer's AOR. `alice`, `sip:alice@host`, and `1001` are all valid patterns.
- A stream is tracked if **either** caller **or** callee matches.
- Adding a pattern (`TRACK_ADD` / `TRACK_REPLACE`) re-evaluates streams that are **already in progress**, so audio decoding begins mid-call from that point forward (`CommandAck.affected_streams` reports how many started).
- `TRACK_REMOVE` only stops *new* streams from being tracked; it does **not** tear down a decode already running.

```
message TrackEntry { string sip_pattern = 1; }

message Command {
    enum Op { TRACK_ADD = 0; TRACK_REMOVE = 1; TRACK_REPLACE = 2; TRACK_QUERY = 3; }
    Op op                       = 1;
    repeated TrackEntry entries = 2;   // empty for TRACK_QUERY
}

message CommandAck {
    bool ok                     = 1;
    string error                = 2;   // set when !ok
    repeated TrackEntry current = 3;   // full list after the op
    uint32 affected_streams     = 4;   // live streams that began decoding as a result
}
```

Driving it from the test tool

`scripts/test/zeromq_monitor.py` doubles as a control client:

```
zeromq_monitor.py track add     sip:alice@example.com 1001   # start tracking
zeromq_monitor.py track list                                 # show current list
zeromq_monitor.py track remove  1001
zeromq_monitor.py track replace sip:bob@example.com          # swap the whole list
# --control tcp://HOST:PORT  to target a non-default endpoint
```

Interactive TUI

The same tool has a terminal UI that combines the live event feed with track-list management — the event log on the left, the tracked-address list on the right (patterns matching a live call are highlighted with a `●`):

```
zeromq_monitor.py tui [--endpoint tcp://127.0.0.1:9125] [--control tcp://127.0.0.1:9127]
```

Keys: `a` add a pattern, `d` / `Del` remove the selected one, `c` clear all, `r` refresh, `p` pause autoscroll, `q` quit. The panel also auto-refreshes (`--refresh`, default 5 s) so it reflects changes made by other clients. Requires the `textual` package (`pip install textual`); `monitor` and `track` do not.

Configuring the bind

```
server:
    zeromq-control-bind:  127.0.0.1     # default; same no-auth caveat as zeromq-bind
    zeromq-control-port:  9127          # default (HEP listener is disabled by default: hep-port 0)
```

Driving it from vq-db (REST)

`vq-db` proxies the track list over its dashboard HTTP port, so the web frontend (or `curl`) can manage it without speaking ZeroMQ. `vq-db` connects to vq-core's control socket at `tcp://localhost:<zeromq-control-port>` (single-host scope). All endpoints use `GET` and return JSON `{ok, error, current:[...], affected_streams}`:

```
curl  http://HOST:9126/track                                # current list (TRACK_QUERY)
curl "http://HOST:9126/track/add?pattern=alice&pattern=1001"
curl "http://HOST:9126/track/remove?pattern=1001"
curl "http://HOST:9126/track/replace?pattern=sip:bob@x"     # no pattern => clear
```

`?pattern=` is repeatable. A missing pattern on add/remove returns `400`; if vq-core's control socket is unreachable the endpoint returns `503`.