Sapsan Manual

Streaming media server

Flussonic

Table of contents

1. Products

2. Sapsan

2.0.1 Sapsan

Flussonic Sapsan (hereafter Sapsan) is a streaming media server. It ingests live video over industry-standard protocols, processes it (transcoding, multibitrate assembly, screenshots), records it into its own on-disk archive (DVR), and delivers it to viewers both live and from the archive.

Purpose: ingesting, processing, recording, and delivering live video in internet streaming, IPTV, and video surveillance projects. The detailed function list is on the functional characteristics page; the system design is on the technical architecture page.

Sapsan is driven by a single configuration file `sapsan.yaml`, reloads it on the fly without interrupting streaming, and can run either standalone or under control of an external management system (Flussonic Central).

Key features

- **Ingest**: RTSP (IP cameras), MPEG-TS (UDP and HTTP), SRT, RTMP, a channel from a file, synthetic test source.
- **Publishing**: RTMP publish, SRT publish, WebRTC WHIP.
- **Ingest failover**: multiple sources per stream with automatic switching (LSI), a fallback file.
- **Transcoding**: multibitrate (MBR) ladder preparation based on FFmpeg — h264/hevc/av1, aac/opus.
- **Playback**: HLS and LL-HLS (fMP4), DASH, MPEG-TS over HTTP (including CBR), SRT, RTSP, WebRTC WHEP.
- **Restreaming**: pushing a stream over RTMP, SRT, and UDP (multicast/unicast).
- **DVR**: append-only multi-disk archive, archive playback, rewind, frame previews, seamless playback of archives from other servers (remotes).
- **Screenshots**: JPEG previews of live streams and the archive.
- **Authorization**: play and publish tokens, external auth backends.
- **ONVIF**: camera discovery on the network, motion and detection events.
- **Management**: Admin API, external configuration (`config_external`), hot reload on SIGHUP, compatibility with Flussonic URLs and API.
- **Peeklio gateway (rproxy)**: access to cameras and devices behind NAT via agents.
- **Observability**: structured logs, Prometheus metrics, OpenTelemetry traces, ingest quality counters.

Quick start

- Install and run Sapsan
- Configure your first stream
- Play the stream over HLS

Documentation navigator

This documentation will help you:

- Install and run Sapsan
- Understand the configuration
- Ingest video:
  - from an IP camera over RTSP
  - from multicast or HTTP MPEG-TS
  - over SRT
  - over RTMP
  - make a channel from a file
  - run a test stream
  - accept RTMP/SRT/WHIP publishing
- Set up backup sources
- Assemble a multibitrate stream from several sources
- Discover ONVIF cameras and receive their events
- Transcode a stream
- Deliver video to viewers:
  - HLS and LL-HLS
  - DASH
  - MPEG-TS
  - SRT
  - RTSP
  - WebRTC
- Restream to other servers
- Record the archive and play it back
- Seamlessly play an archive from another server
- Get stream screenshots
- Protect playback and publishing
- Operate the server:
  - Admin API
  - External configuration management
  - Peeklio gateway and agents
  - Monitoring, logs and traces
  - Licensing
  - Flussonic compatibility

2.1 Functional characteristics

The complete function list of the Flussonic Sapsan streaming media server. Each function is described in the administrator guide (links inline).

2.1.1 Video ingest

- Capture from IP cameras and media servers over RTSP (RTP/AVP/TCP interleaved) with Basic and Digest authentication — details. Codecs: H264, HEVC, VP8, MJPEG video; AAC, G.711, Opus, MPEG-4 audio; ONVIF metadata.
- MPEG-TS ingest from UDP (unicast, multicast IPv4/IPv6) and over HTTP — details. Codecs: H264, HEVC, MPEG-2 video; AAC, AC3, EAC3, MPEG-2 audio; teletext, KLV, SCTE-35.
- SRT ingest in listener mode with AES encryption and streamid control — details.
- RTMP pull, including Adobe/FMS authentication and enhanced RTMP (HEVC) — details.
- A channel from a local MP4 file (multi-track MBR) — details.
- A built-in synthetic test signal generator with a machine-readable timecode in the picture — details.
- Publishing: RTMP publish, SRT publish, WebRTC WHIP; air hijack protection (one publisher per stream) — details.
- Source failover with automatic no-black-screen switching (LSI), a fallback file, input control via external flag files — details.
- Multibitrate stream assembly from several sources — details.
- ONVIF camera discovery, motion/detection event ingest, camera compliance checks — details.

2.1.2 Video processing

- FFmpeg-based transcoding: H264/HEVC/AV1 video, AAC/Opus audio, a multibitrate ladder with aligned keyframes, resizing (scale/fit/crop), GOP control — details.
- Periodic JPEG screenshots of a stream (from a video track or a camera snapshot URL) stored into the archive — details.

2.1.3 Video delivery

- HLS and LL-HLS (fMP4/CMAF, protocol v9): partial segments, blocking reload, playlist delta updates — details.
- MPEG-DASH (live and archive), multibitrate in one AdaptationSet — details.
- MPEG-TS over HTTP, including strict CBR with correct PCR and HRD handling — details.
- SRT output with encryption — details.
- An RTSP server (H264/HEVC/AAC) — details.
- WebRTC WHEP with minimal latency — details.
- Restreaming: push over RTMP (including HEVC/AV1), SRT, UDP multicast — details.

2.1.4 Recording and archive (DVR)

- Recording into its own append-only storage across several disks with automatic balancing — details.
- Retention policies by depth and size, protected episodes, automatic disk overflow protection.
- Self-healing of the archive catalog after failures and disk moves.
- Archive playback over HLS/DASH: rewind with seamless stitching to live, access by absolute time — details.
- Seamless playback of archives recorded on other servers, with lazy replication — details.

2.1.5 Management and security

- Single-YAML-file configuration with validation and hot reload without interrupting streaming — details.
- Configuration management from an external server (Flussonic Central) with fail-safe application — details.

- Playback and publish authorization: static tokens, external HTTP backends, session accounting — details.
- An Admin API (Flussonic API v3 compatible plus native v4) — details.
- Compatibility with Flussonic Media Server URL schemes — details.
- The Peeklio gateway for reaching cameras behind NAT via agents — details.
- Licensing with signed license files — details.

2.1.6 Observability

- Structured logs, Prometheus metrics, OpenTelemetry traces — details.
- Ingest quality counters (MPEG-TS, SRT), source, session, and DVR statistics.
- Built-in HLS and DASH validators for delivery diagnostics.

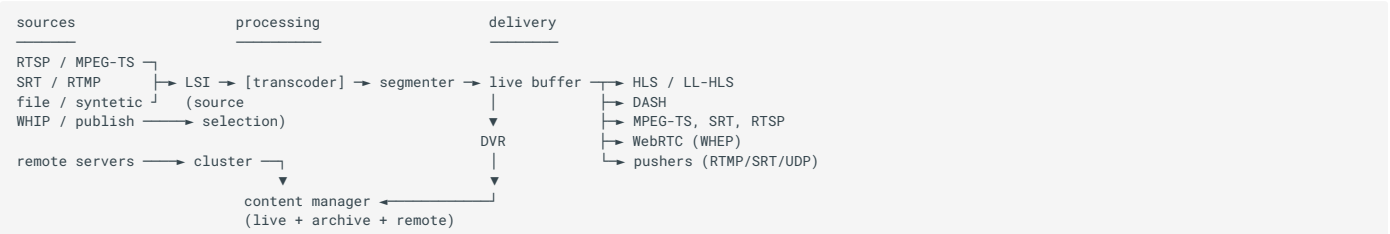
2.2 Technical architecture

Flussonic Sapsan is a server application written in Rust. All subsystems run inside a single `sapsan` process as tasks of the tokio asynchronous runtime and interact over typed message channels — with no shared mutable state between subsystems.

2.2.1 Components

Component	Purpose
Manager	configuration loading and validation, subsystem startup, network listener binding, hot reload, Admin API
Network listeners	HTTP (HLS, DASH, MPEG-TS, WHIP/WHEP, API), RTSP, RTMP, WebRTC (UDP), per-stream SRT ports
Streams manager	stream lifecycle: inputs and source switching (LSI), publishing, pushers, screenshots
Transcoder	decoding and encoding (FFmpeg): H264/HEVC/AV1, AAC/Opus, the multibitrate ladder
Segmenter and live buffer	slicing the stream into fMP4 fragments, the live-edge ring buffer
Content manager	the unified content surface: merges the live buffer, the local archive, and remote archives; all delivery protocols read from it
DVR	on-disk archive storage: hourly append-only blobs, a catalog on an embedded KV database, a background indexer (rebuild, metadata backfill, cleanup)
Cluster	reading archives from remote Sapsan servers over the internal streaming-api
Sessions and authorization	viewer/publisher session accounting, tokens, external auth backends
Peeklio gateway	a reverse proxy for devices behind NAT: agent registration, tunnels
Telemetry	structured logs, Prometheus metrics, OpenTelemetry traces

2.2.2 Data flows



- **Write path**: the stream's active input (selected by LSI) delivers frames; they optionally pass the transcoder; the segmenter assembles fMP4 fragments, which enter the live buffer and are appended to the DVR in parallel.
- **Read path**: a protocol module requests data from the content manager, which transparently picks the source — the live buffer, the local archive, or a remote server. Players never know where the data physically came from.
- **DVR read/write isolation**: each stream's writes are serialized in its own task; reads bypass it and go straight to the blobs through a shared cache — readers do not contend with the writer.

2.2.3 Content addressing

The unit of storage and delivery is a fragment addressed by absolute UTC time (DTS + timescale). The same fragment name is valid in the live buffer, in the archive, and on every cluster server. This property underpins the seamless stitching of live to archive and the merging of archives across servers. The URL contract (routes, playlist URIs, fragment references) is defined in a single streaming-api module shared by the server and cluster clients.

2.2.4 Configuration lifecycle

1. The configuration is read from `sapsan.yaml` and validated as a whole; an invalid configuration is not applied.
2. On SIGHUP the configuration is re-read: streams, authorization, and DVR are reconfigured in place; listeners are rebound only when a port changes.
3. With `config_external` enabled, the stream list is periodically fetched from an external server; partially invalid updates are applied without the broken streams, fully invalid ones are discarded keeping the working configuration.

2.2.5 Technology stack

Layer	Technology
Language	Rust
Async runtime	tokio (network I/O, tasks), rayon (CPU parallelism)
Allocator	jemalloc
Transcoding	FFmpeg
DVR catalog	embedded key-value database (LSM)
Observability	OpenTelemetry (OTLP), Prometheus

3. Administrator

3.1 Starting

3.1.1 Installing and running Sapsan

This page describes how to install Sapsan, run it with a minimal configuration, and verify that the server works.

Before installation

> **Note**
>
> TODO: system requirements (OS, architectures, CPU/RAM, DVR disks).

Installing from the deb package

Sapsan is distributed as the `sapsan` deb package from the Flussonic repository:

```
echo "deb http://apt.flussonic.com binary/" > /etc/apt/sources.list.d/flussonic.list
apt update
apt install sapsan
```

Running in Docker

The official image is `flussonic/sapsan`. Map the ports and the configuration file into the container:

```
docker run -d --name sapsan \
  -p 80:80 -p 554:554 -p 1935:1935 \
  -v /etc/sapsan/sapsan.yaml:/etc/sapsan/sapsan.yaml \
  -v /storage:/storage \
  flussonic/sapsan
```

> **Note**
>
> TODO: exact config path inside the image, environment variables, UDP ports for SRT/WebRTC.

Minimal configuration

Sapsan reads its configuration from `sapsan.yaml` (the path can be overridden with the `CONFIG_PATH` environment variable). A minimal config is one HTTP port and one stream:

```
listeners:
  http:
    - port: 80

streams:
  demo:
    inputs:
    - syntetic: {}
    transcoder:
      output:
      - source: !content video
        codec: !set h264
      - source: !content audio
        codec: !set aac
```

Starting and stopping

```
sapsan
# or with an explicit config path:
CONFIG_PATH=/etc/sapsan/sapsan.yaml sapsan
```

- Reload the configuration without interrupting streaming: `kill -HUP <pid>`.
- Gracefully stop the server: `kill -TERM <pid>`.

> **Note**
>
> TODO: systemd unit, default logs, where to check status.

Health check

Open the test stream playlist:

```
curl http://localhost/streaming/v/demo/index.m3u8
```

If the server returns a playlist, Sapsan is installed and running. Next, configure real video ingest.

3.1.2 Sapsan configuration

All of Sapsan's configuration lives in a single YAML file, `sapsan.yaml`. This page describes its sections and reload rules.

File structure

```
listeners:        # ports the server listens on
  http:
    - port: 80
  rtsp:
    - port: 554
  rtmp:
    - port: 1935
  webrtc:
    - port: 5005

streams:          # named streams
  cam1:
    inputs:
    - rtsp:
        url: rtsp://admin:password@10.0.0.5/stream0
    dvr: {}

dvr:              # shared archive storage
  root: /storage
  disks:
  - path: d1

auth: {}            # playback and publish authorization
config_external: {} # external configuration management
rproxy: {}          # Peeklio gateway
api_auth:           # Admin API login/password
  login: admin
  password: secret
```

Section	What it configures	Details
`listeners`	HTTP, RTSP, RTMP, and WebRTC server ports	this page
`streams`	streams: sources, transcoder, DVR, pushes, screenshots	ingest
`dvr`	on-disk archive storage	DVR recording
`auth`	tokens and auth backends	authorization
`config_external`	fetching configuration from an external server	external configuration
`rproxy`	Peeklio gateway for agents	Peeklio gateway
`api_auth`	Admin API access	Admin API

Validation

Sapsan validates the configuration on load and refuses to start with an invalid one:

- unknown fields are an error (protects against typos);
- every stream input must contain exactly one protocol;
- ports cannot be reused across listeners and streams;
- if a stream has `dvr` enabled, the global `dvr` section must exist.

Hot reload

On `SIGHUP` Sapsan re-reads `sapsan.yaml` and applies changes without restarting the process:

- streams, authorization, and DVR are reconfigured in place;
- listeners are rebound only when a port changes;
- the rproxy gateway keeps agent connections across reloads (but changing `streampoint_key` / `endpoint_auth_url` requires a process restart).

```
kill -HUP $(pidof sapsan)
```

3.2 Ingest

3.2.1 RTSP ingest

Sapsan pulls video from IP cameras and other RTSP sources in client (pull) mode.

Configuration

```
streams:
  cam1:
    inputs:
    - rtsp:
        url: rtsp://admin:password@10.0.0.5:554/stream0
        peer_timeout_ms: 5000
```

Parameter	Description
`url`	camera address; login and password go in the URL
`peer_timeout_ms`	source inactivity timeout after which the input is considered down
`via_agent`	ingest through a Peeklio agent (`agent://ID`) when the camera is behind NAT — see Peeklio gateway

Authentication

Basic and Digest are supported (with automatic fallback to Basic when the camera rejects Digest). Credentials come from the URL and are not sent to the camera in the clear when Digest is used.

Codecs and transport

Transport is RTP/AVP/TCP (interleaved), robust against NAT and firewalls. Decoded: H264, HEVC, VP8, MJPEG video; AAC, G.711 (PCMA/PCMU), Opus, MPEG-4 audio; ONVIF metadata (camera events) straight from the RTP stream. RTP timestamp wraparound and rollback are handled.

The SDP parser is battle-tested against real cameras: Axis, Hikvision, Dahua, Bosch, Panasonic, Uniview, Beward, and others — including cameras with malformed SDP (missing PPS, broken SEI).

> **Note**
>
> G.711 audio does not play over HLS — add a transcoder that re-encodes audio to AAC.

Applying changes

Changing `url` or `via_agent` restarts the source; changing only `peer_timeout_ms` applies in place.

Verification

Open `http://server/streaming/v/cam1/index.m3u8` in a player or request a screenshot at `http://server/streaming/live-preview-jpeg/cam1`.

Next steps

- Add a backup source
- Combine two camera qualities into one MBR stream
- Discover cameras via ONVIF
- Record the stream to the archive

3.2.2 MPEG-TS ingest

Sapsan ingests SPTS MPEG-TS streams in two ways: by listening on UDP (unicast and multicast, IPv4 and IPv6) and by pulling over HTTP.

UDP ingest

```
streams:
  tv1:
    inputs:
    - udp:
        host: 239.0.0.1
        port: 5000
        peer_timeout_ms: 5000
```

If `host` is a multicast group, Sapsan joins it itself (IGMP/MLD). `peer_timeout_ms` sets the silence timeout after which the input is considered down and LSI switches to a backup.

HTTP ingest

```
streams:
  tv2:
    inputs:
    - tshttp:
        url: http://origin.example.com/tv2/mpegts
        peer_timeout_ms: 5000
```

Chunked transfer is supported. A dropped connection and a silent source are distinguished (`PeerClosed` / `PeerTimeout`) and visible in the input status.

What is decoded

H264, HEVC, MPEG-2 video; AAC, AC3, EAC3, MPEG-2 audio; DVB teletext, KLV metadata, SCTE-35 markers; track language descriptors and DVB subtitle descriptors from the PMT.

Applying changes

On config reload, changing `host` / `port` (UDP) or `url` (HTTP) restarts the source; changing only `peer_timeout_ms` applies in place without a disruption.

Quality counters

Per-PID reception quality counters are collected: packets and frames, CC errors (continuity counter violations, including across interval boundaries), TEI errors, scrambled packets, PSI CRC errors, broken PES, decoder buffer (HRD) issues. See the Admin API and monitoring.

Next steps

- Set up a backup source
- Serve the stream back as SPTS
- Push the stream to multicast

3.2.3 SRT ingest

Sapsan ingests an SRT stream by listening on a dedicated UDP port (listener mode). Each stream gets its own port.

Configuration

```
streams:
  event:
    inputs:
    - srt:
        host: 0.0.0.0
        port: 9000
        passphrase: verysecretpass
```

Parameter	Description
`host`, `port`	address and UDP port to wait for the SRT connection on
`passphrase`	encryption key (AES); a mismatching key gets the connection rejected
`stream_id`	expected client streamid; the Flussonic-style `#!::r=<name>` form is passed through verbatim
`handshake_timeout_ms`, `peer_timeout_ms`	handshake and inactivity timeouts

You can send a stream to Sapsan like this:

```
ffmpeg -re -i input.mp4 -c copy -f mpegts \
  "srt://server:9000?passphrase=verysecretpass"
```

Delivery reliability

The full SRT ARQ machinery is implemented: acknowledgements (ACK/ACKACK), retransmission requests for lost packets (NAK), a latency buffer with in-order delivery, too-late packet drop, and keepalives for idle connections. The NAK period follows the measured RTT.

Applying changes

Changing the address, port, `passphrase`, or `stream_id` restarts the source; changing only the timeouts applies in place.

Counters

Per connection: received packets and bytes, RTT and its variance, lost and retransmitted packets in both directions, buffer sizes, keepalive timeouts. See the Admin API.

Next steps

- Serve the stream over SRT to viewers
- Restream over SRT to another server

3.2.4 RTMP ingest

Sapsan can pull an RTMP stream from another server in client (pull) mode. For accepting incoming RTMP publishing (push into Sapsan), see publishing.

Configuration

```
streams:
  relay:
    inputs:
    - rtmp:
        url: rtmp://origin.example.com/live/streamkey
        peer_timeout_ms: 5000
```

If the server requires authentication, put the login and password in the URL: `rtmp://user:password@origin.example.com/live/streamkey` — the multi-stage Adobe/FMS authentication (`authmod=adobe`) is supported.

Codecs

H264 + AAC, HEVC (enhanced RTMP), audio-only streams. Broken metadata (`onMetaData`) does not break ingest.

Error behavior

Server rejection codes are distinguished and visible in the input status: stream not found, access denied, stream stopped, protocol error. A dropped connection and a silent server (`peer_timeout_ms`) are also distinguished. LSI switches to a backup input on any of these.

Applying changes

Changing `url` restarts the source; changing only `peer_timeout_ms` applies in place.

Next steps

- Set up a backup source
- Restream further over RTMP

3.2.5 Channel from a file

The `file` input turns a local MP4 file into a proper live stream: the file plays in a loop. This is a production tool for filler channels: a "we'll be right back" technical channel, a promo channel, and a fallback when sources go down.

Configuration

```
streams:
  promo:
    inputs:
    - file:
        path: /storage/promo.mp4
    dvr: {}
```

The file can be multi-track: several video qualities and several audio tracks in one MP4 give a full MBR stream with language selection — with no transcoder on the server.

Preparing the file with ffmpeg

For an MBR file to play seamlessly, all qualities must be perfectly aligned: identical timestamps and keyframes in the same places. This is achieved with a single ffmpeg run using the `split` filter:

```
ffmpeg -y -i source.mkv \
  -filter_complex "\
[0:v:0]split=3[vhi][vme][vlo]; \
[vhi]scale=1920:1080:force_original_aspect_ratio=decrease,pad=1920:1080:(ow-iw)/2:(oh-ih)/2,setsar=1,fps=25,setpts=PTS-STARTPTS[v1]; \
[vme]scale=1280:720:force_original_aspect_ratio=decrease,pad=1280:720:(ow-iw)/2:(oh-ih)/2,setsar=1,fps=25,setpts=PTS-STARTPTS[v2]; \
[vlo]scale=960:540:force_original_aspect_ratio=decrease,pad=960:540:(ow-iw)/2:(oh-ih)/2,setsar=1,fps=25,setpts=PTS-STARTPTS[v3]" \
  -map "[v1]" -c:v:0 libx264 -b:v:0 6000k -g:v:0 100 -keyint_min:v:0 100 \
    -sc_threshold:v:0 0 -x264-params:v:0 "scenecut=0:open_gop=0" \
  -map "[v2]" -c:v:1 libx264 -b:v:1 3000k -g:v:1 100 -keyint_min:v:1 100 \
    -sc_threshold:v:1 0 -x264-params:v:1 "scenecut=0:open_gop=0" \
  -map "[v3]" -c:v:2 libx264 -b:v:2 1200k -g:v:2 100 -keyint_min:v:2 100 \
    -sc_threshold:v:2 0 -x264-params:v:2 "scenecut=0:open_gop=0" \
  -map 0:a:0 -c:a:0 aac -b:a:0 192k -ac 2 -ar 48000 \
  -metadata:s:v:0 title="1080p" -metadata:s:v:1 title="720p" -metadata:s:v:2 title="540p" \
  -metadata:s:a:0 language=eng \
  -movflags +faststart \
  promo.mp4
```

What matters here:

- **one run, one source,** `split` — all qualities get identical timestamps;
- `setpts=PTS-STARTPTS` — each quality's timeline starts at zero;
- the same `fps` in all qualities;
- **rigid GOP**: `-g` = `-keyint_min`, `-sc_threshold 0` and `scenecut=0:open_gop=0` — keyframes strictly every N frames in the same places; the encoder may not insert them on scene changes;
- AAC audio at 48 kHz stereo;
- `-movflags +faststart` — the index at the front of the file;
- `-metadata:s:v title=...` / `language=...` — quality labels and track languages end up in the playlists.

A complete working example with multiple audio tracks lives in the sapsan repository: `storage/king.sh`.

Verification

Open `http://server/streaming/v/promo/index.m3u8` — the master playlist must list all qualities, and switching between them must not stutter.

Next steps

- Use a file as a fallback when a source goes down
- Synthetic test source

3.2.6 Synthetic source

The `syntetic` input is a built-in test signal generator: SMPTE bars with a running clock and a tone. It needs no external data and serves install verification, debugging, and load testing.

Configuration

The generator emits uncompressed frames (raw YUV and PCM). This is deliberate: the raw signal can be sent losslessly straight to SDI. To serve the stream over network protocols (and get the codec/quality variants you need), pair the generator with a transcoder:

```
streams:
  syn:
    inputs:
    - syntetic: {}
    transcoder:
      output:
      - source: !content video
        codec: !set h264
      - source: !content audio
        codec: !set aac
    segments: 6
```

Without a `transcoder` section the stream carries raw frames: a regular player cannot play it, but this is exactly the stream you need for SDI output.

Generator parameters (all optional, `syntetic: {}` gives a demo signal):

```
    - syntetic:
        video: !video
          rate:
            numerator: 30000     # fractional frame rates (NTSC) are supported
            denominator: 1001
        audio: !audio
          sample_rate: 48000
          channels: 2
```

Timecode in the picture

Each frame's PTS is drawn into the luma band of the image and can be machine-read back. This makes it possible to measure end-to-end latency and find lost frames from the picture itself — handy when debugging the transcoder and protocols.

Next steps

- Transcoding
- Channel from a file

3.2.7 Accepting publications

Publishing means the source connects to Sapsan and sends the stream itself (as opposed to ingest, where Sapsan connects to the source). Sapsan accepts publications over RTMP, SRT, and WebRTC WHIP.

Stream configuration

A stream is declared with a `publish` input, which means "wait until somebody publishes":

```
listeners:
  rtmp:
    - port: 1935
  webrtc:
    - port: 5005

streams:
  studio:
    inputs:
    - publish: {}
```

RTMP publishing

Publish from OBS or ffmpeg to:

```
rtmp://server:1935/static/studio
```

> **Note**
>
> TODO: clarify the RTMP URL scheme (application/stream key) and streamkey-based publish authorization.

WebRTC publishing (WHIP)

Standard WHIP endpoint:

```
http://server/streaming/whip/studio
```

You can publish from a browser or any WHIP-compatible client (e.g. OBS 30+). Codecs are H264 and Opus.

One publisher per stream

While a publication is active, a second publication into the same stream is rejected — the air cannot be hijacked. If the stream also has regular inputs, a publication does not kick out a working source: LSI switches to the publisher under the usual readiness rules (keyframe), and with a fallback file configured the publication may yield to the filler.

SRT publishing

SRT publishing is configured as an SRT input with a dedicated port per stream.

Publish authorization

Publications are protected with `publish_tokens` — see authorization.

Next steps

- Play the published stream
- Record the publication to the archive

3.2.8 Source failover

A stream can have multiple sources. Sapsan monitors the active source, automatically switches to the next one when it degrades, and switches back when the primary recovers. This logic is called LSI (Live Source Input).

Multiple inputs

Inputs are listed in priority order — the first one is the primary:

```
streams:
  tv1:
    inputs:
    - udp:
        host: 239.0.0.1
        port: 5000
    - tshttp:
        url: http://backup-origin.example.com/tv1/mpegts
```

How switching works

Key properties of the algorithm that matter on air:

- **A source becomes active only when ready**: it must deliver stream parameters (MediaInfo) and the first keyframe. Until then viewers keep receiving the current source.
- **Failback without a black screen**: a recovered higher-priority input is probed in the background, and the switch happens only once it has delivered a keyframe.
- A failed input reconnects with growing delays (`reconnect_initial_delay` → `reconnect_delay_step` → `reconnect_max_delay`); when all inputs are exhausted, rotation wraps back to the first.
- A source whose bitrate exceeds `max_bitrate` is forcibly stopped.
- Separate no-video and no-audio timeouts: lost audio with live video also counts as degradation.

Switching policy

The `lsi` section sets the stream-wide policy; `source_options` on a specific input overrides it:

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    - udp:
        host: 239.0.0.2
        port: 5000
      source_options:
        source_timeout: 10
    lsi:
      source_timeout: 5
      video_timeout: 3
      audio_timeout: 3
```

Timeouts are in seconds. Main parameters (`lsi` and/or `source_options`):

Parameter	Description
`source_timeout` , `video_timeout` , `audio_timeout`	no-data/no-video/no-audio timeouts
`max_bitrate`	source bitrate limit
`reconnect_initial_delay` , `reconnect_delay_step` , `reconnect_max_delay`	reconnection policy
`retry_limit` , `max_retry_timeout`	retry limits
`recheck_secondary_inputs_interval`	how often to check whether a higher-priority input came back
`allow_if` , `deny_if` (`source_options` only)	enable/disable an input via an external flag file

`allow_if` / `deny_if` let you control inputs externally without editing the config: an input is used only while the flag file allows it.

Fallback file (backup)

If all sources are down, a file is shown instead of a black screen (see how to prepare the file):

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    lsi:
      backup:
        file: /storage/technical-difficulties.mp4
        timeout: 5
```

The fallback kicks in `timeout` seconds after the inputs fail (immediately if there are no inputs at all) and has its own `audio_timeout` / `video_timeout`.

Observability

Per stream: time on the primary and backup inputs, time with no data, retry counts, backup input validity, and a **parameter divergence detector** — if a backup input delivers different video parameters (making a seamless switch impossible), you see it in advance, before the incident. Per input: open/connect/start times and the last error. See monitoring.

Next steps

- Prepare a fallback file
- Monitor source switching

3.2.9 Multibitrate ingest

Many cameras and encoders output several qualities as separate streams. The `mbr` input merges them into one multibitrate stream: the viewer gets a single playlist with all qualities and adaptive switching.

Configuration

```
streams:
  cam1:
    inputs:
    - mbr:
        inputs:
        - rtsp:
            url: rtsp://admin:password@10.0.0.5/stream0   # high quality
        - rtsp:
            url: rtsp://admin:password@10.0.0.5/stream1   # low quality
```

Inside `mbr.inputs` the same protocols are allowed as in regular `inputs`.

> **Note**
>
> TODO: track alignment rules (timestamps, GOP), quality ordering in the playlist, behavior when one quality goes down.

Alternative: transcoder

If there is a single source, multibitrate is prepared by the transcoder.

Next steps

- Serve the MBR stream over HLS
- Record all qualities to the archive

3.2.10 ONVIF cameras

Sapsan can discover ONVIF cameras on the network, receive their events (motion, human and vehicle detection), and check a camera for compliance.

> **Warning**
>
> The ONVIF subsystem is under active development — the feature set and configuration may change.

Camera discovery

Sapsan discovers cameras via WS-Discovery (multicast ProbeMatch) and additionally understands Hikvision's proprietary discovery protocol. For each camera it collects addresses (XAddrs), name, and model; repeated observations of the same camera are merged.

When working through NAT, the addresses a camera reports about itself (snapshot URL, RTSP URL) are automatically rewritten to the actually reachable device address.

Camera events

Events from ONVIF metadata become episodes with open and close:

- motion (CellMotion `IsMotion`, MotionAlarm) — `true` opens an episode, `false` closes it;
- human and vehicle detection;
- digital input triggers;
- stuck episodes are closed by an inactivity timeout.

Multiple motion sources on one camera are tracked independently.

Compliance check

The built-in camera check produces a structured report: device information, authentication (HTTP Digest), camera clock (time skew is handled automatically), event classification, plus an RTSP stream probe with an error summary (packet loss, broken payload, desync) and keyframe statistics.

> **Note**
>
> TODO: how discovery and the compliance check are launched (config/API/CLI), camera configuration via ONVIF, snapshot retrieval.

Next steps

- RTSP camera ingest

3.3 Transcode

3.3.1 Transcoding

The Sapsan transcoder (based on FFmpeg) re-encodes the input stream: changes codecs, bitrate, resolution, and prepares a multibitrate (MBR) ladder from a single source.

Configuration

The `transcoder.output` section describes output tracks. Each track takes a source (`!content video` or `!content audio`) and sets a codec:

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    transcoder:
      output:
      - source: !content video          # 1080p — top of the ladder
        codec: !set h264
        bitrate: 2800000
        params: !video
          gop_size: 28
          gop_structure: ipppb
      - source: !content video          # 720p
        codec: !set h264
        bitrate: 1200000
        params: !video
          gop_size: 28
          gop_structure: ipppb
          resize: !fit
            height: 720
      - source: !content audio
        codec: !set aac
```

Source track selection

`source`	What it takes
`!content video` / `!content audio`	a track by content type
`!exact v1`	a specific track by id
`!best_quality video`	the best available quality
`!language_codec [eng, aac]`	a track by language and codec

The `if_missing` parameter sets the behavior when the track is absent: `drop` (default — the output is not created), `blank` (empty signal), or `substitute` .

Codecs and parameters

- `codec: !set <name>` — re-encode (h264, hevc, av1, aac, opus); `codec: same` — pass through without re-encoding (e.g. only repackage audio).
- Outputs of different codecs from one source keep aligned timestamps and keyframes — an h264/hevc/av1 ladder stays consistent.
- `gop_size` , `gop_structure` (e.g. `ipppb`) — GOP structure control, important for aligned MBR segments.
- Audio: G.711 → AAC/Opus 48 kHz re-encoding with timeline preservation — the typical IP camera case.
- JPEG screenshot tracks pass through the transcoder untouched.

Resizing

`resize`	What it does
`!scale`	exact scaling to the given dimensions
`!fit`	fit preserving aspect ratio (with padding, background color configurable)
`!crop`	crop to the given dimensions

> **Note**
>
> TODO: hardware acceleration (Nvenc/Vulkan), deinterlacing, overlays/logos, audio parameter reference.

Verification

Open the master playlist `http://server/streaming/v/tv1/index.m3u8` — all output qualities should appear in it.

Next steps

- Serve the transcoded stream
- Record the ladder to the archive

3.4 Playback

3.4.1 HLS and LL-HLS playback

Every Sapsan stream is immediately available over HLS (fMP4/CMAF) without extra configuration. Live streams get LL-HLS with partial segments by default.

Playback URLs

What	URL
Master playlist (all qualities)	`http://server/streaming/v/<stream>/index.m3u8`
Single track playlist	`http://server/streaming/v/<stream>/variant/<track>/index.m3u8`
Rewind	`http://server/streaming/v/<stream>/rewind/<seconds-ago>/index.m3u8`
Archive by absolute time	`http://server/streaming/v/<stream>/index.m3u8?from=<utc_ms>&to=<utc_ms>`

The master playlist carries qualities with `RESOLUTION`, `FRAME-RATE`, and `CODECS`; audio goes into rendition groups (`EXT-X-MEDIA`). G.711 audio (PCMA/PCMU) is not served over HLS — transcode such streams to AAC first.

LL-HLS

For live streams Sapsan serves full LL-HLS (protocol version 9):

- `EXT-X-PART` partial segments at the live edge and `EXT-X-PRELOAD-HINT` for the next part;
- blocking playlist reload: the client parameters `_HLS_msn` and `_HLS_part` hold the request until the requested segment/part appears;
- playlist delta updates: `_HLS_skip=YES` shortens the playlist with an `EXT-X-SKIP` tag;
- `EXT-X-RENDITION-REPORT` for fast quality switching;
- `PART-HOLD-BACK`, `CAN-SKIP-UNTIL`, `HOLD-BACK` are derived automatically from segment and part durations.

LL-HLS can be disabled per client with `?llhls=false` — the master playlist propagates it into all variant URLs, and the player gets classic HLS (version 7).

Tokens in playlists

If playback is token-protected, `?token=` is automatically appended to every nested playlist URL: variants, init segments, segments, parts, and preload hints. The player only needs to open the master playlist with the token.

Segment count

The live playlist length is set by the stream's `segments` parameter:

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    segments: 6
```

Diagnostics

Sapsan ships an HLS playlist validator: it estimates the expected latency and points at configuration problems with codes like `LL-DISABLED`, `TARGETDURATION-INFLATED`, `PART-TARGET-LARGE`, `SPARSE-INDEPENDENT-PARTS` (GOP too long for LL-HLS).

Note

TODO: how to run the validator (CLI/API).

Next steps

- Protect playback with a token
- Play the archive

3.4.2 DASH playback

Every Sapsan stream is available over MPEG-DASH: both live and archive.

Playback URLs

What	URL
Live manifest	`http://server/streaming/v/<stream>/Manifest.mpd`
Archive by absolute time	`http://server/streaming/v/<stream>/Manifest.mpd?from=<utc_ms>&to=<utc_ms>`

Live manifest

A dynamic MPD (`isoff-live` profile): all qualities of an MBR stream live in one video AdaptationSet as separate Representations sorted by ascending bandwidth — players build the ABR ladder correctly. The `timeShiftBufferDepth` window and `suggestedPresentationDelay` are derived automatically (the delay is about two segments behind the live edge).

Archive manifest

The archive MPD is static. Recording gaps become separate `<Period>` s, and the timeline is selectable with the `?timeline=` parameter:

- `compact` (default) — gaps are collapsed, the archive plays back continuously;
- `wallclock` — gaps are preserved, the timeline matches real time.

Each Representation's `bandwidth` is measured from the actual data of the requested range. If the archive has a JPEG screenshot track, it is exposed as a standard thumbnail tile AdaptationSet.

Diagnostics

The built-in DASH validator checks a manifest: start-time alignment of ABR ladder tracks, gaps and overlaps in the SegmentTimeline between refreshes, audio/video live-edge drift. The mode (static/live) is auto-detected from the manifest `@type`.

> **Note**
>
> TODO: how to run the validator, verified players (dash.js, ExoPlayer).

Next steps

- Protect playback with a token
- Play the archive

3.4.3 MPEG-TS output

Sapsan muxes any stream into SPTS MPEG-TS and serves it over HTTP — convenient for set-top boxes, transcoders, and legacy systems.

Playback URL

```
http://server/streaming/mpegts/<stream>
```

What is produced

- Correct PAT/PMT with continuous continuity counters; PIDs can be set explicitly, conflicting and reserved ones are reassigned automatically.
- H264/HEVC video, AAC/AC3/EAC3 audio, track language descriptors, teletext (teletext descriptor with pages and languages).
- JPEG screenshot tracks never enter the TS.

CBR

Sapsan can output a strictly constant bitrate — a requirement of professional receivers and modulators:

- the target rate is derived from track bitrates (with ~5% growth headroom and a PSI budget);
- sparse spots are filled with null packets (PID 0x1FFF);
- PCR is placed from a drift-free global clock — PCR-to-DTS divergence does not accumulate even on hours-long looped content;
- the decoder buffer model (HRD) is respected — overruns and underruns are tracked.

> **Note**
>
> TODO: how to enable CBR and set the bitrate in the config (currently the rate is derived from track bandwidth).

Sending over UDP

For sending MPEG-TS to multicast over UDP, see restreaming.

Next steps

- Ingest MPEG-TS
- Restream

3.4.4 SRT output

Sapsan serves a stream over SRT: each stream gets a dedicated UDP port that SRT clients connect to in caller mode.

Configuration

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    srt_play:
      port: 4010
      passphrase: verysecretpass
```

`passphrase` enables encryption; without it the stream is served unencrypted.

Playback

```
ffplay "srt://server:4010?passphrase=verysecretpass"
```

Next steps

- Ingest a stream over SRT
- Push a stream over SRT to another server

3.4.5 RTSP output

Sapsan acts as an RTSP server: any stream can be pulled over RTSP — the standard way to feed video into NVRs, analytics systems, and other media servers.

Configuration

Enabling the RTSP listener is enough:

```
listeners:
  rtsp:
    - port: 554
```

Playback

```
ffplay rtsp://server:554/cam1
```

What the server supports:

- OPTIONS, DESCRIBE, SETUP, PLAY, TEARDOWN methods;
- RTP/AVP/TCP (interleaved) transport — works over a single TCP port, firewall-friendly;
- an RTCP Sender Report before the first RTP packet of each track — the client gets absolute time anchoring right away;
- H264 and HEVC video, AAC audio (RFC 3640);
- original stream timestamps are preserved.

> **Note**
>
> TODO: exact RTSP URL scheme (path = stream name?), UDP transport, RTSP playback authorization.

Next steps

- Ingest a stream over RTSP

3.4.6 WebRTC playback (WHEP)

For minimal-latency playback Sapsan serves streams over WebRTC via the standard WHEP protocol.

Configuration

WebRTC needs a UDP port on the server:

```
listeners:
  webrtc:
    - port: 5005
```

Playback URL

The stream's WHEP endpoint:

```
http://server/streaming/whep/<stream>
```

Any WHEP-compatible player works.

Requirements and behavior

- The stream must have an **H264** video track; **Opus** audio is optional. A stream with another video codec is not served over WHEP (re-encode with the transcoder). Streams with G.711 audio also go through the transcoder to Opus.
- The server answers PLI/NACK (a fresh keyframe on player request) and paces packet output instead of flooding the client.
- WebRTC uses the fixed UDP port set from `listeners.webrtc` — easy to open on a firewall; the server's external address is detected automatically.

> **Note**
>
> TODO: an HTML player example, limitations (no MBR playback, simulcast, TWCC yet).

Next steps

- Accept WHIP publishing

3.4.7 Screenshots

Sapsan can periodically capture JPEG frames from a stream: show a current preview in UIs and store frames in the archive alongside the video.

Configuration

A stream can have several screenshot generators. Each takes frames from one of two sources:

- `http_preview` — fetch a ready JPEG over HTTP (e.g. a camera's snapshot URL);
- `video_preview` — render a frame from the stream's own video track.

```
streams:
  cam1:
    inputs:
    - rtsp:
        url: rtsp://admin:password@10.0.0.5/stream0
    thumbnails:
    - http_preview:
        url: http://10.0.0.5/snapshot.jpg
        timeout: 5      # request timeout, s (default 5)
      interval: 10       # capture period, s (default 10)
      dvr: true          # store frames in the archive (default false)
    - video_preview:
        track: v1        # source video track
        width: 320       # default 320, minimum 32
        height: 180
      interval: 30
```

Validation rules: `interval` and `timeout` must be greater than 0; frame dimensions must be at least 32; `timeout` is only allowed on `http_preview`, and `width` / `height` only on `video_preview`.

Garbage protection: a response that is not a JPEG or exceeds the size limit is discarded — the preview is not updated.

Getting screenshots

What	URL
Current stream frame	`http://server/streaming/live-preview-jpeg/<stream>`
Frame of a specific track	`http://server/streaming/track-preview-jpeg/<stream>/<track>`
Frame from the archive	`http://server/streaming/dvr-preview-jpeg/<stream>/<track>/image/<ref>`

Screenshots in the archive

With `dvr: true` frames are written into the archive as a separate JPEG track with real frame dimensions. In the archive's DASH manifest it is exposed as a standard thumbnail tile AdaptationSet (`http://dashif.org/guidelines/thumbnail_tile`) — players that support timeline thumbnails pick it up automatically.

Next steps

- Configure DVR recording

3.4.8 Authorization

Sapsan protects stream playback and publishing. There are two mechanisms: static tokens in the config and external auth backends the server delegates decisions to.

How the token is passed

A token is accepted in two ways (the header takes priority):

- the `Authorization: Bearer <token>` header;
- the `?token=<token>` URL parameter.

For HLS it is enough to open the master playlist with the token — Sapsan appends `?token=` to every nested URL itself (variants, segments, parts, init).

Static tokens

```
auth:
  play_tokens:
  - viewer-secret-1
  publish_tokens:
  - publisher-secret-1
```

- `play_tokens` — playback tokens;
- `publish_tokens` — publishing tokens.

A request without a valid token is denied. If the `auth` section exists but neither tokens nor backends matched, access is denied.

External auth backends

Decisions can be delegated to HTTP backends:

```
auth:
  upstreams:
    main:
      url: http://backend.example.com/auth
      timeout_ms: 3000
```

Sapsan makes a `POST` to the backend's `url` with a JSON body:

```
{
  "kind": "play",                  // play or publish
  "stream_name": "cam1",
  "proto": "hls",                  // protocol: hls, dash, rtsp, m4f, ...
  "user_agent": "Mozilla/5.0 ...", // may be absent
  "token": "viewer-secret-1",      // may be absent
  "session_id": "..."              // stable session identifier
}
```

The decision is made from the HTTP response status (the body is not parsed):

- `2xx` — allow;
- `401` or `403` — deny (the denial is cached — an identical repeat request does not hit the backend);
- any other status, unreachability, or exceeding `timeout_ms` (default 1000 ms) marks the backend as down;
- **multiple backends are queried in parallel**: one "allow" is enough; only denials → deny; all backends down → a "backend down" error (not a silent deny).

Sessions

A viewer session is identified by the quadruple: stream, kind (play/publish), IP, token — repeated requests from the same viewer do not spawn new sessions. Long-lived connections (RTSP, WebRTC) are periodically re-authorized: if the backend revokes access, the session closes. Session counters (opened, denied, authorized) are available in monitoring.

Admin API access

The Admin API is protected by a separate `api_auth` section — see Admin API.

Next steps

- Configure publishing

3.5 Restream

3.5.1 Restreaming (push)

Sapsan can actively send a stream onward: to CDNs and streaming platforms over RTMP, to another server over SRT, or into the network over UDP (multicast/unicast). A stream can have several pushes at once.

Configuration

Each push has a unique name and exactly one protocol:

```
streams:
  tv1:
    inputs:
    - udp: {host: 239.0.0.1, port: 5000}
    pushes:
    - name: youtube
      rtmp:
        url: rtmp://a.rtmp.youtube.com/live2/xxxx-xxxx
    - name: backup-dc
      srt:
        host: 203.0.113.10
        port: 9000
        passphrase: verysecretpass
    - name: local-multicast
      udp:
        host: 239.1.1.1
        port: 5500
```

Protocol	Parameters
`rtmp`	`url`, `connect_timeout_ms`
`srt`	`host`, `port`, `passphrase`, `stream_id`, timeouts
`udp`	`host`, `port` — MPEG-TS to multicast or unicast

Capabilities and behavior

- RTMP push handles H264+AAC, HEVC, and AV1 (enhanced RTMP), and audio-only streams; timestamps start at zero on push, as CDNs expect.
- Push errors are distinguished and visible in status: connection refused, connection timeout, publish rejection (`NetStream.Publish.BadName` — key taken or wrong), a stalled server (frame write timeout).
- An SRT push with a mismatching `passphrase` gets a handshake rejection; a silent receiver (no ACKs) is registered as a peer timeout.

> **Note**
>
> TODO: push reconnection policy, track selection for a push (which quality goes out).

Next steps

- Monitor pushes

3.6 DVR

3.6.1 DVR recording

Sapsan's DVR is its own on-disk storage: a stream is written append-only into hourly files, per track, across several disks at once. Archive reads do not interfere with writes.

Storage

The global `dvr` section describes the whole storage:

```
dvr:
  root: /storage            # archive root
  catalog: /storage/catalog # blob catalog (defaults to <root>/catalog)
  disks:
  - path: d1                # disks relative to root
  - path: d2
  - path: d3
```

Disk parameters:

Parameter	Description
`path`	disk path relative to `root`
`mode`	`Active` (default) or `Degraded` — the disk is read but not written
`min_free_bytes`	minimum free space reserve

`check_mount` verifies that the disk path is actually a mount point (protects against writing into an empty directory when a disk drops off).

Enabling recording for a stream

```
streams:
  cam1:
    inputs:
    - rtsp:
        url: rtsp://admin:password@10.0.0.5/stream0
    dvr:
      max_depth: 168        # archive depth, hours
      max_bytes: 500000000000
```

`dvr: {}` enables recording with default settings.

Write distribution across disks

Sapsan balances recording across active disks on its own:

- consecutive hours of one stream alternate between disks;
- neighboring streams are spread evenly across disks;
- tracks of a multibitrate stream are written to different disks in parallel.

A disk in `degraded` mode keeps serving reads but receives no new data — this is how you decommission a disk without losing the archive.

Cleanup and retention

Cleanup runs once an hour (5 minutes after the hour boundary):

- `max_depth` is hour-granular: only fully elapsed hours are deleted;
- `max_bytes` is counted from newest blobs to oldest: everything older than the first blob exceeding the limit is deleted;
- when both limits are set, the tighter one wins;
- blobs with protected episodes are never deleted by retention;
- the blob of the current (actively written) hour is never deleted.

Disk-overflow protection works separately: when free space drops below `min_free_bytes` (default 1% of disk capacity), Sapsan deletes the oldest blobs **regardless of retention** until enough space is freed.

How storage works

The archive is laid out on disks as `<disk>/<hash>/<stream>/<Y>/<M>/<D>/<hour>.mp4`. Files are only appended, never rewritten; identical init segments are deduplicated within a blob.

A background indexer continuously reconciles the catalog with the disks and heals divergence without administrator involvement:

- blobs that appeared on disk behind the catalog's back (e.g. after moving disks from another server) are added to the catalog;
- records of missing files are removed;
- a corrupt blob sidecar index is rebuilt by scanning the blob itself;
- blob byte sizes and bloom filters are backfilled in the background (blobs younger than one hour are left alone).

Reads are failure-tolerant: if the catalog points at the wrong disk, the fragment is searched on all disks of that hour; orphaned catalog records are skipped without an error.

Observability

DVR reports statistics per disk (free/total space, read/write op and byte counters, blob counts, bytes used by the archive) and for the catalog — see the Admin API and monitoring.

Next steps

- Play the archive
- Get screenshots from the archive

3.6.2 Archive playback

The recorded archive is available over the same protocols as live: HLS and DASH. There is also an API for querying recorded ranges and previews.

Archive playback works only for streams with recording enabled (`dvr`) — otherwise the request is rejected with a clear error.

Rewind

A playlist rewound N seconds back from now:

```
http://server/streaming/v/<stream>/rewind/<seconds>/index.m3u8
```

For example, `rewind/3600/` is a one-hour timeshift. Rewind seamlessly stitches the archive to the live edge: the playlist starts in the archive and continues with live, with no duplicates or gaps at the seam; LL-HLS (blocking reload, delta updates) keeps working. Recording gaps become a standard `EXT-X-DISCONTINUITY` .

Playback by absolute time

The archive interval is set with `from` / `to` parameters (UTC in milliseconds) on the regular URLs:

```
http://server/streaming/v/<stream>/index.m3u8?from=<utc_ms>&to=<utc_ms>
http://server/streaming/v/<stream>/Manifest.mpd?from=<utc_ms>&to=<utc_ms>
```

The archive HLS playlist is a closed VOD playlist. In DASH, recording gaps become Periods, and the timeline is selected with `?timeline=compact|wallclock` — see DASH.

Which ranges are recorded

Recorded intervals are served by the HTTP API:

```
http://server/streaming/dvr-range/<stream>
http://server/streaming/dvr-ranges/<stream>
```

> **Note**
>
> TODO: dvr-range / dvr-ranges response format.

Archive frame previews

- A JPEG frame from the screenshot track: `http://server/streaming/dvr-preview-jpeg/<stream>/<track>/image/<utc_ms>.jpg`
- An MP4 fragment of an archive moment: `http://server/streaming/dvr-preview-mp4/<stream>/<utc_ms>` — a short fragment from a keyframe; the server picks the lowest-resolution track for the preview itself.

Next steps

- Configure recording
- Play an archive from another server
- Protect the archive with authorization

3.6.3 Archive from another server (remotes)

Sapsan can seamlessly play an archive that is recorded (or used to be recorded) on another server. A stream lists remote servers — and their archives become an extension of the local one: the viewer sees one continuous archive and never knows where the data physically comes from.

Typical tasks this solves:

- migrating to a new server without losing history — the new one records fresh archive, the old one keeps the past;
- reading the archive from replica servers;
- recovering after a disk replacement.

Configuration

```
streams:
  cam1:
    inputs:
    - rtsp:
        url: rtsp://10.0.0.5/stream0
    dvr: {}
    remotes:
    - url: http://old-server:5080
      timeout_ms: 5000
```

Parameter	Description
`url`	address of the remote Sapsan server that has this stream's archive
`timeout_ms`	timeout for requests to the remote server

There can be several remote servers — Sapsan queries them all.

How seamlessness works

On every archive access Sapsan merges local and remote data:

- **Archive bounds** — the union of all servers' ranges: the earliest start and the latest end. In UIs and the API the stream looks like one continuous archive.
- **Playlists** — fragment lists from the local DVR and all remote servers are merged, sorted, and deduplicated. There is no seam, gap, or duplicate at the junction of two archives.
- **Fragments** — read in a cascade: live buffer → local DVR → remote server. The fragment URL is the same for the player regardless of where the data lives.

This is possible thanks to Sapsan's absolute fragment addressing: a fragment's name is its UTC time, so the same fragment has the same name on every server, and merging archives requires no timeline remapping.

Lazy replication

A fragment read from a remote server is automatically written into the local archive. The local DVR gradually "pulls in" the parts of foreign history that viewers actually watch — and over time stops fetching them over the network. A failure of such a write never interferes with serving the viewer.

Failure behavior

- If a remote server is unreachable but the data exists locally, playback continues; the problem is only logged.
- Sapsan remembers which ranges are absent on a remote (negative cache) and does not re-query it for missing data.
- Init fragments of remote streams are cached.

Note

TODO: inter-server request authorization, timeout recommendations, limits (how many remotes are reasonable), interaction with `config_external`.

Next steps

- Configure DVR recording
- Play the archive

3.7 Administration

3.7.1 Admin API

Sapsan exposes server state over HTTP. The API is format-compatible with the Flussonic Streamer API v3, so existing integrations keep working.

Access

API access is protected by a login and password in the `api_auth` section:

```
api_auth:
  login: admin
  password: secret
```

Endpoints

The base prefix is `/streamer/api/v3`:

Method and path	Returns
`GET /streamer/api/v3/streams`	list of streams and their state
`GET /streamer/api/v3/config`	current server configuration
`GET /streamer/api/v3/dvrs`	list of DVR storages
`GET /streamer/api/v3/dvrs/{name}`	state of a specific DVR
`GET /streamer/api/v3/rproxy`	Peeklio gateway and agent state
`GET /streamer/api/v3/monitoring/readiness`	readiness probe for orchestrators

Native API v4

The evolving native API lives under the `/streamer/api-v4` prefix:

Path	Returns
`GET /listeners`	listener list
`GET /streams`, `GET /streams/{name}`	streams and their state
`GET /streams/stats`, `GET /streams-stats`	stream statistics
`GET /sessions/stats`	session statistics
`GET /live-metrics`, `GET /sessions-metrics`	live and session metrics
`GET /dvr`, `GET /dvr/catalog`, `GET /dvr/metrics`	DVR storage and catalog state
`POST /dvr/rebuild-index`, `GET` (status), `DELETE` (stop)	catalog rebuild from disks
`POST /dvr/cleanup`, `GET /dvr/cleanup`	start archive cleanup / status
`GET /runtime/metrics`	process metrics in Prometheus format
`GET /auth`	authorization state
`GET /license/status`	license state
`GET /rproxy/streampoint-agents`	Peeklio gateway agents

> **Note**
>
> TODO: response examples, stream management via API (once writes appear).

Next steps

- External configuration management
- Monitoring

3.7.2 External configuration management

In cluster installations Sapsan streams are managed by an external system (e.g. Flussonic Central): the server periodically fetches the stream list from an external URL and applies it.

Configuration

```
config_external:
  url: http://central.example.com/central/api/v3/streamers/streamer1.example.com
  bearer: <access token>
  client_host: streamer1.example.com
  interval_secs: 5
  timeout_ms: 30000
```

Parameter	Description
`url`	where to fetch the configuration from
`bearer`	authorization token for the external server
`client_host`	the name this server identifies itself with to the management system
`interval_secs`	polling period (5 s by default)
`timeout_ms`	request timeout (30 s by default)
`fetch_runtime_config`	whether to also fetch the runtime configuration (`true` by default)

Important

When `config_external` is enabled, the local `streams` section in `sapsan.yaml` is ignored — the external server becomes the source of truth. The other sections (`listeners` , `dvr` , `auth` , `api_auth`) stay local unless the external server supplies its own.

How polling works

- Sapsan requests `GET <url>/streams` every `interval_secs` ; the list is fetched page by page via the `next` cursor. Every request carries `Authorization: Bearer` , an `x-originator: sapsan` header, and the `client_host` .
- The runtime configuration (`GET <url>/config` — listeners, DVR, auth) is fetched only when the server announces it has one.
- If the configuration has not changed, the re-apply is skipped.

Failure tolerance

- **A partially invalid list**: broken streams are dropped (reporting which field failed), valid ones are applied — status `Partial` .
- **A fully invalid configuration** (e.g. a port conflict): nothing is applied, the previous configuration keeps running — status `Error` .
- External server unavailability, broken JSON, and HTTP errors are distinguished in the status (`network` / `malformed_json` / `http<code>`).

Next steps

- Admin API

3.7.3 Peeklio gateway (rproxy)

Peeklio is a reverse-proxy gateway embedded in Sapsan. A lightweight agent installed next to a camera (behind NAT) connects out to Sapsan and maintains a tunnel; through this tunnel Sapsan pulls the device's video and traffic as if it were on the local network.

Gateway configuration

```
rproxy:
  streampoint_key: <key for streampoint connections>
  endpoint_auth_url: http://backend.example.com/agent-auth
  agents:
  - agent_id: a1b2c3
    password: <agent password>
    salt: <salt>
    streampoint_url: http://this-server:5080
```

Agents can be authorized in two ways: a local `agents` list in the config, or an external `endpoint_auth_url` backend. Without `endpoint_auth_url` the gateway runs in "streampoint-only" mode — it accepts tunnels but does not register new agents.

On the agent side you set the server address, `agent_id`, password, ping interval, and a **host ACL** — the list of addresses that may be reached through this agent at all; everything else is rejected.

> **Warning**
>
> Gateway state survives configuration reloads (agents do not drop on SIGHUP), but changing `streampoint_key` / `endpoint_auth_url` requires a process restart.

Ingesting a stream through an agent

A stream input specifies `via_agent` with the agent identifier:

```
streams:
  cam-behind-nat:
    inputs:
    - rtsp:
        url: rtsp://admin:password@192.168.1.10/stream0
        via_agent: "agent://a1b2c3"
```

The address in `url` is the camera's local address in the agent's network. The RTSP handshake and the stream go through the tunnel.

Error behavior

Connection errors through an agent are distinguishable: closed port (connection refused), host not in the agent's ACL (permission denied), unresolvable name. The server can remotely send an agent `reset` and `reboot` commands.

Agent monitoring

`GET /streamer/api-v4/rproxy/streampoint-agents` returns per-agent counters: pings, bytes both ways, attempted/opened/current connections. See the Admin API.

> **Note**
>
> TODO: installing and configuring the agent itself (rproxy-agent package, agent.toml), key issuance.

3.7.4 Monitoring

Sapsan writes structured logs, serves Prometheus-format metrics, and exports traces via OpenTelemetry (OTLP) — it fits the standard observability stack: Prometheus/Grafana, Jaeger/Tempo, Loki.

Logs

The log level is controlled by the `RUST_LOG` environment variable:

```
RUST_LOG=info sapsan
RUST_LOG=debug,hls=trace sapsan   # more detail for a specific module
```

> **Note**
>
> TODO: log format, default destinations in deb/Docker installations.

Prometheus metrics

`GET /streamer/api-v4/runtime/metrics` serves process metrics in Prometheus format (including the jemalloc allocator). Next to it — live, session, and DVR metrics: `/live-metrics`, `/sessions-metrics`, `/dvr/metrics` (see the Admin API).

Quality counters

What is available for alerting:

- **MPEG-TS ingest**: per PID — CC errors, TEI, scrambled packets, PSI CRC errors, broken PES, decoder buffer (HRD) state.
- **SRT**: RTT, loss and retransmits both ways, buffer sizes, keepalive timeouts.
- **Sources (LSI)**: time on primary/backup input, time with no data, retries, backup validity, parameter divergence between inputs.
- **Sessions**: opened, denied, authorized, re-auth denials.
- **DVR**: disk space, op counters, blob counts, bytes used by the archive, catalog rebuild progress.
- **Pushes and Peeklio agents**: connection state, bytes, errors.

Traces

Sapsan exports traces over the OTLP protocol.

> **Note**
>
> TODO: OTLP exporter environment variables (endpoint, sampling), what is covered by traces.

Server health

- Readiness probe: `GET /streamer/api/v3/monitoring/readiness` — for Kubernetes and load balancers.
- Stream state: `GET /streamer/api/v3/streams`.

Diagnostic tools

The built-in HLS and DASH validators actively check your own (or someone else's) delivery: LL-HLS latency, timeline integrity, ABR ladder alignment.

3.7.5 Licensing

Sapsan is licensed with a license file issued to the customer at purchase.

The license is verified online: the server needs internet access to the Flussonic licensing servers. Without internet access Sapsan will not work.

The Flussonic licensing servers are distributed across continents and regions — the failure of a single server or the unavailability of a whole region does not affect license verification.

> **Note**
>
> Sapsan never updates itself. There is no forced-update functionality, and there never will be: license verification does not change the installed software, and a version upgrade is always an explicit administrator action through the package manager.

Files

Three files live in the license directory:

File	Contents
`license.txt`	the license key issued at purchase
`server.id`	unique server identifier (UUID)
`activation-<version>.json`	activation for a specific product version

States

State	When	What works
`Licensed`	the license is verified	everything
`Restricted`	the license is absent, expired, or failed verification	the admin interface stays available, streaming is limited

The current license status is served by the API: `GET /streamer/api-v4/license/status`.

Next steps

- Admin API

3.7.6 Flussonic compatibility

Sapsan answers on the familiar Flussonic Media Server URLs so existing players, integrations, and monitoring keep working during migration. Compatibility is on by default and can be disabled in the config:

```
flussonic:
  streaming: false   # default true
```

Supported legacy URLs

Root-level addresses of the following forms are recognized (GET/HEAD/OPTIONS only):

Legacy URL	What it does
`/<stream>/variant/v1/index.m3u8`	track playlist
`/<stream>/info.json`	stream information
`/<stream>/ranges.json`	recorded archive ranges
`/<stream>/<utc>-preview.mp4`	MP4 preview of an archive moment
`/<stream>/index-<from>-<duration>.fmp4.m3u8`	archive HLS playlist; redirects (302) to `/streaming/v/<stream>/index.m3u8?from=&to=`, converting seconds to milliseconds

`ranges.json` parameters: `closed_at_gte`, `opened_at_gte`, `opened_at_lte`, `resolution`, `limit`, `cursor`.

Admin API v3

The API under the `/streamer/api/v3` prefix responds in the Flussonic Streamer API v3 format — see Admin API. When converting a v3 configuration, some fields are dropped by design (`static`, `comment`, `title`, `segment_duration`, `https` listeners, and others) — a loss report is available during conversion.

Tokens

As in Flussonic, a token is accepted both as `?token=` in the URL and as an `Authorization: Bearer` header (the header takes priority).

Next steps

- Admin API
- Authorization